

CHAPTER 1. 신경망 복습

1.1 수학과 파이썬 복습

1.1.1 벡터와 행렬

`w = np.array([[1, 2, 3], [4, 5, 6]])`

`w.shape → (2, 3)` 마치 원 배열의 행

`w.ndim → 2` 차원수

1.1.2 행렬의 원소별 연산

1.1.3 브로드캐스트

1.1.4 벡터의 내적, 행렬의 곱

$$x \cdot y = x_1 y_1 + x_2 y_2 + \dots + x_n y_n$$

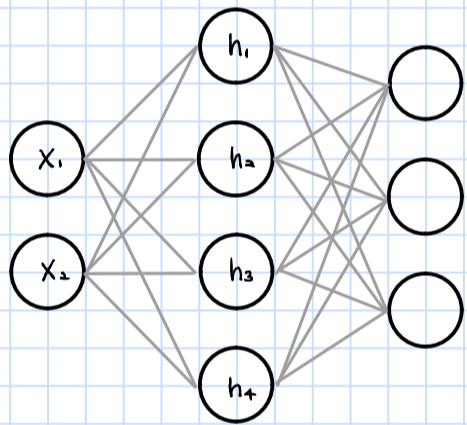
- 행렬의 곱: 왼쪽 행렬의 행벡터와 오른쪽 행렬의 열벡터의 내적.

1.1.5 행렬 예상 학습

1.2 신경망의 추론

- "학습"과 "추론"

1.2.1 신경망 추론 전체 그림



$$- h_i = x_1 w_{i1} + x_2 w_{i2} + b_i$$

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

(1, 4) (1, 2) (2, 4)

↓
간소화

$$- h = xW + b$$

- 원 배열

$$\begin{array}{ccc} x & w & = h \\ \begin{matrix} N \times 2 \\ \text{일치} \end{matrix} & \begin{matrix} 2 \times 4 \\ \text{일치} \end{matrix} & \begin{matrix} N \times 4 \\ \text{일치} \end{matrix} \end{array}$$

- '비선형' 결과를 부여하기 위해 활성화함수 사용.

예) 시그모이드

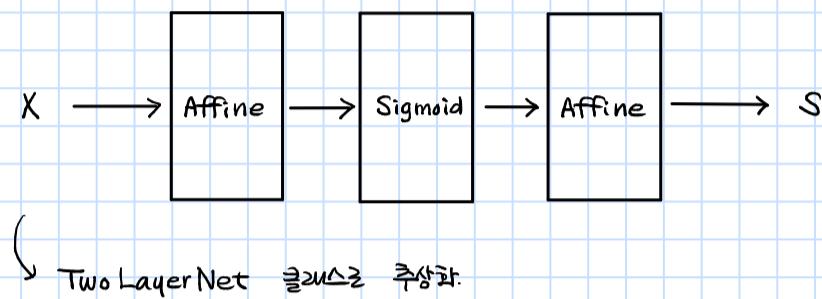
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

1. 2. 2. 계층으로 클래스화 및 순전파 구현

- Affine : 완전연결 계층에 의한 변환
- Sigmoid : 시그모이드 함수에 의한 변환

* 구현 규칙

- 모든 계층은 forward() 와 backward() 메서드를 가진다.
- 모든 계층은 이스터스 변수인 params 와 grads를 가진다.
 - params : 가중치, 편향 등을 담는 리스트
 - grads : params에 저장된 각 매개변수에 대응하여, 해당 매개변수의 기울기를 보관하는 리스트



```
X = np.random.randn(10, 2)
```

```
model = TwoLayerNet(2, 4, 3)
```

```
s = model.predict(X)
```

1. 3 신경망의 학습.

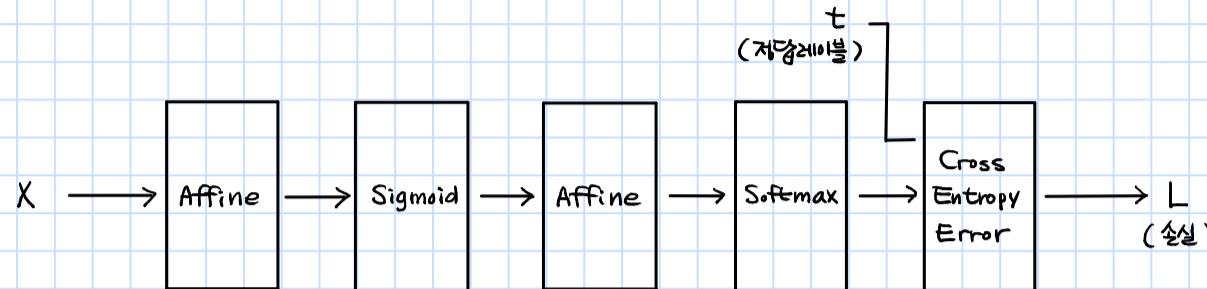
- 최적의 매개변수의 값을 찾는 작업.

1. 3. 1 솔실함수

- 솔실: 학습데이터(학습 시 주어진 정답데이터)와 신경망이 예측한 결과를 비교,

예측이 얼마나 나쁠지를 산출한 단일값 (스칼라)

- 솔실함수: 대중의 경우 Cross Entropy Error



- 소프트맥스 함수

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

(출제가 중 n개일 때,
K번째 출제 y_k 를 구하는 계산식)
 y_k 는 K번째 클래스에 해당하는 소프트맥스 함수의 출력.

: 지수 s_k 의 지수 함수

: 모든 입력신호의 지수 함수의 총합

↓
소프트맥스의 출력 (= 확률)이 교차 엔트로피 오차에 적용됨.

- 교차 엔트로피의 수식

$$L = -\sum_k t_k \log y_k \rightarrow \text{정답레이블이 1인 원소에 해당하는 출력의 자연로그를 계산.}$$

: K번째 클래스에 해당하는 정답레이블

$t = [0, 0, 1]$ 과 같이 원핫벡터로 표기

- 미니배치 고려

$$L = -\frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk}$$

: n 번째 데이터의 k 차원 째 값 의미, 정답레이블

: 신경망의 출력.

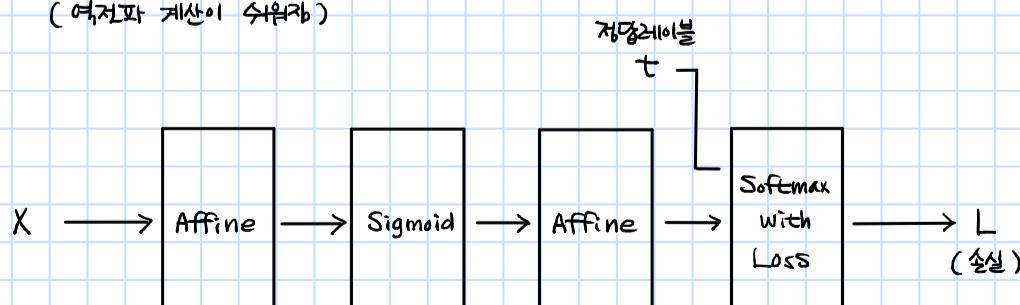
- N개짜리로 확장.

- N으로 나눠서 '평균 솔실 함수'를 구함.

→ 미니배치의 크기에 관계없이 항상 일관된 척도를 얻을 수 있음.

- 소프트맥스와 교차 엔트로피 오차를 Softmax with Loss 계층으로 구현

(여전파 계산이 수월해)



1.3.2 미분과 기울기

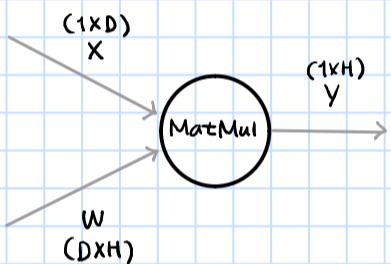
- x 에 관한 y 의 미분 : $\frac{dy}{dx}$
· x 의 값을 조금 변화시켰을 때 y 값이 변하는 정도
- 기울기 (gradient) : 베クトル의 각 원소에 대한 미분을 정의.
- w 가 $m \times n$ 행렬이라면 $L = g(w)$ 함수의 기울기

$$\frac{\partial L}{\partial w} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \dots & \frac{\partial L}{\partial w_{1n}} \\ \vdots & \ddots & \\ \frac{\partial L}{\partial w_{m1}} & \dots & \frac{\partial L}{\partial w_{mn}} \end{pmatrix}$$

1.3.3 연쇄법칙

1.3.4 계산그래프

- 텁새노드
- 꼽새노드
- 분기노드
- Repeat 노드
- Sum 노드
- MatMul 노드



· x 의 i 번째 원소에 대한 미분 $\frac{\partial L}{\partial x_i}$

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

$$\frac{\partial y_j}{\partial x_i} = W_{ij} \text{ 가 성립}$$

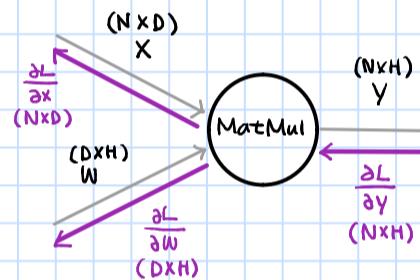
$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} W_{ij}$$

$\frac{\partial L}{\partial x_i}$ 을 벡터 $\frac{\partial L}{\partial y}$ 과 W 의 i 행 벡터의

내적으로 구해지는 것을 알 수 있음

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

$$(1 \times D) \quad (1 \times H) \quad (H \times D)$$



- 학습의 여러 단계에서도 순차 단계 간의 영향을 서로 바꾼 학습을 사용.

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

$$(N \times D) \quad (N \times H) \quad (H \times D)$$

$$\frac{\partial L}{\partial w} = X^T \frac{\partial L}{\partial y}$$

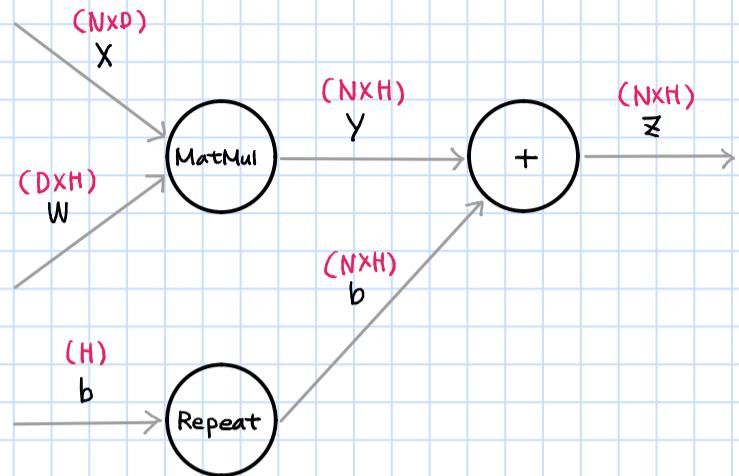
$$(D \times H) \quad (D \times N) \quad (N \times H)$$

1.3.5 기울기 토출과 역전파 구현

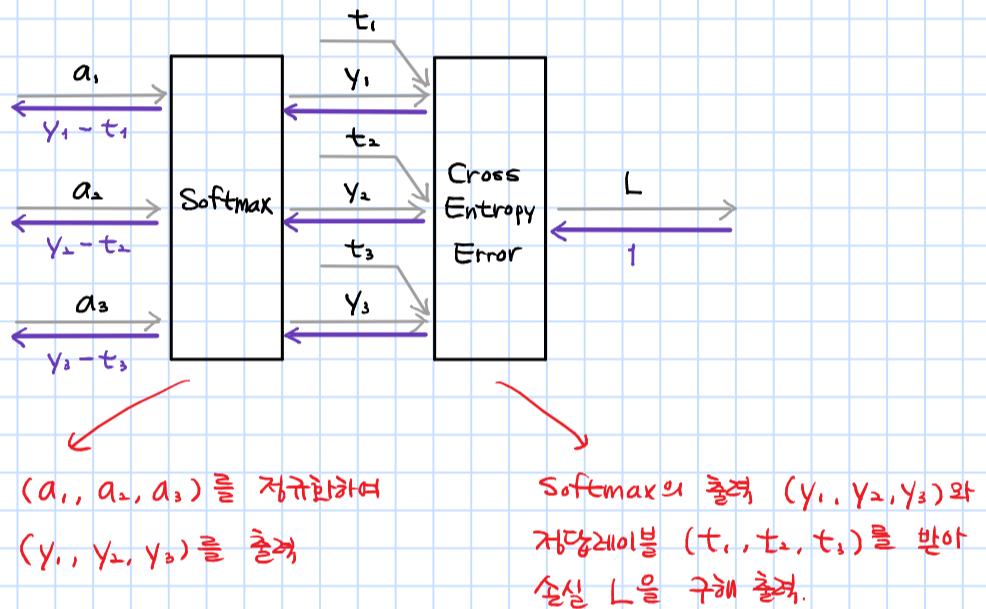
- Sigmoid 계층

- Sigmoid의 미분 : $\frac{\partial y}{\partial x} = y(1-y)$
- $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x} = \frac{\partial L}{\partial y} \cdot y(1-y)$

- Affine 계층



- Softmax with Loss



1. 3. 6 가중치 개선

- SGD: 현재의 가중치를 기울기 방향으로 일정한 거리만큼 개선

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

: "에타", 학습률

: W 에 대한 손실함수의 기울기

1. 4. 신경망으로 문제를 풀다.

1. 4. 1 스파이럴 데이터셋

1. 4. 2 신경망 구현

1. 4. 3 학습용 코드

학습의 4단계

1단계 미니배치

- 훈련데이터에서 무작위로 다수의 데이터를 끌어내기

2단계 기울기 계산

- 오차역전파법으로 각 가중치 매개변수에 대한 손실함수의 기울기를 구함

3단계 매개변수 개선

- 기울기를 사용하여 가중치 매개변수 개선

4단계 반복

1.4.3 학습 코드

```
import sys
sys.path.append('..')
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 = []

for epoch in range(max_epoch):
    # 데이터 뒤섞기
    idx = np.random.permutation(data_size)
    x = x[idx]          데이터 인덱스 뒤섞기
    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 개선
    loss = model.forward(batch_x, batch_t)
    model.backward()
    optimizer.update(model.params, model.grads)

    total_loss += loss
    loss_count += 1

    if loss_count % 10 == 0:
        print("Epoch %d, Iter %d, Loss: %f" % (epoch, iters, loss))

print("Final loss: %f" % total_loss)
```

→ 예전 단위를 데이터를 뒤섞고
뒤섞은 데이터 중 앞에서부터 순서대로
뽑아내는 방식 사용.



```
total_loss += loss
```

```
loss_count += 1
```

초기적으로 학습결과 출력

```
if (iters+1) % 10 == 0: → 10 번째 반복마다 솔실평균을 구해 loss_list 변수에 추가
```

```
avg_loss = total_loss / loss_count
```

```
print ('| 에폭 %d | 반복 %d / %d | 솔실 %.2f' % (epoch+1, iters+1, max_iters, avg_loss))
```

```
loss_list.append (avg_loss)
```

```
total_loss, loss_count = 0, 0
```

앞에서 예제에 의한 변화의 디버깅하기 버전

```
import numpy as np
```

```
W1 = np.random.randn(2, 4)      # 가중치  
b1 = np.random.randn(4)        # 편향  
x = np.random.randn(10, 2)     # 입력, : 각 샘플데이터  
h = np.matmul(x, W1) + b1      # 브로드캐스트 됨.
```

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

```
a = sigmoid(h)
```

종합:

```
import numpy as np
```

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

```
x = np.random.randn(10, 2)      # 입력, : 각 샘플데이터  
W1 = np.random.randn(2, 4)      # 가중치  
b1 = np.random.randn(4)        # 편향  
W2 = np.random.randn(4, 3)      → 원래는 뉴런 4개, 출력은 뉴런 3개이므로 4x3 행상으로 설정.  
b2 = np.random.randn(3)
```

```
h = np.matmul(x, W1) + b1
```

```
a = sigmoid(h)
```

```
s = np.matmul(a, W2) + b2  
↓ 최종 출력, (10, 3)
```

: 10개의 데이터가 한 개씩에 처리,
각 데이터는 3차원 데이터를 변환함.

```
import numpy as np

class Sigmoid:
```

```
    def __init__(self):
```

```
        self.params = []
```

```
        self.out = None
```

```
    def forward(self, x):
```

```
        out = 1 / (1 + np.exp(-x))
```

```
        self.out = out
```

```
        return out
```

출처는 이스토스 변수 out에 저장되고
역전파를 계산할 때 out 변수를 사용.

```
    def backward(self, dout):
```

```
        dx = dout * (1.0 - self.out) * self.out
```

```
        return dx
```

class Affine:

```
def __init__(self, W, b):
    self.params = [W, b] → 초기화될 때 가중치와 편향을 받음.
    (신경망이 학습될 때 수시로 개선)
    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) → 해결의 허상을 잘 살펴보고
                           어느 쪽으로 힘을 구할지 명시.
self.grads[0][...] = dW
self.grads[1][...] = db
return dx
```

```

import sys
sys.path.append('..') → 디렉토리 추가 (absolute path to parent directory)

import numpy as np

from common.layers import Affine, Sigmoid, SoftmaxWithLoss

class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        I, H, O = input_size, hidden_size, output_size
        self.params, self.grads = [], []
        self.layers = [Affine(W1, b1),
                      Sigmoid(),
                      Affine(W2, b2)]
        self.loss_layer = SoftmaxWithLoss()

    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

```

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

↳ deep copy, '덮어쓰기' 수행
변수의 메모리 주소 고정 → 인스턴스 변수 grads를 더 다루기 쉬워짐

```
class SGD:
```

```
    def __init__(self, lr=0.01):
```

```
        self.lr = lr
```

```
    def update(self, params, grads): → 매개변수 개선을 처리
```

```
        for i in range(len(params)):
```

```
            params[i] -= self.lr * grads[i]
```

2.1 자연어처리학

2.1.1 단어의 의미

2.2 시소러스 (thesaurus, 유의어 사전)

2.2.1 WordNet

2.3 통계 기반 기법

2.3.2 단어의 분산표현

2.3.3 분포 개선

- "단어의 의미는 주변 단어에 의해 형성된다"

2.3.4 동시 발생 해결

2.3.5 베이지안 유사도

- 코사인 유사도 : 두 베이지언 가리키는 방향이 얼마나 비슷한가?

$$\text{Similarity}(x, y) = \frac{x \cdot y}{\|x\| \|y\|} = \frac{x_1 y_1 + \dots + x_n y_n}{\sqrt{x_1^2 + \dots + x_n^2} \sqrt{y_1^2 + \dots + y_n^2}} \rightarrow L_2 \text{ norm}$$

☞ 해석 : 베이지언을正规화하고 내적을 구함

2.4 통계 기반 기법 개선하기

2.4.1 상호정보량

- 점별 상호정보량 (Pointwise Mutual Information)

$$\text{PMI}(x, y) = \log_2 \frac{P(x, y)}{P(x)P(y)}$$

- ● : x 가 일어날 확률
- ● : y 가 일어날 확률
- ● : x, y 가 동시에 일어날 확률

↓ 동시발생 해결을 사용하여 식을 다시 씁

$$\text{PMI}(x, y) = \log_2 \frac{C(x, y)}{P(x)P(y)} = \log_2 \frac{\frac{C(x, y)}{N}}{\frac{C(x)}{N} \frac{C(y)}{N}} = \log_2 \frac{C(x, y) \cdot N}{C(x)C(y)}$$

- ● : 말뭉치 단어 수
- ● : 동시 발생하는 횟수

N : 10,000

"the": 1,000 번, "car": 20 번, "drive": 10 번이라고 가정

"the", "car" 동시발생 10회, "car", "drive" : 5회

$$\text{PMI}(\text{"the"}, \text{"car"}) = \log_2 \frac{10 \cdot 1000}{1000 \cdot 20} \approx 2.32$$

why? 단어가 단독으로 출현하는 횟수가 고려되었기 때문.

$$\text{PMI}(\text{"car"}, \text{"drive"}) = \log_2 \frac{5 \cdot 1000}{20 \cdot 10} \approx 7.97$$

↓ 두 단어의 동시발생 횟수가 0이면

$\log_2 0 = -\infty$... 양의 상호정보량 사용

$$\text{PPMI}(x, y) = \max(0, \text{PMI}(x, y))$$

2.4.2 차원감소

- 특이값 분해 (Singular Value Decomposition, SVD)

$$X = U \Sigma V^T$$

2.4.3 SVD에 의한 차원 감소

```
import sys
sys.path.append('..')
import numpy as np
import matplotlib.pyplot as plt
from common.util import preprocess, create_co_matrix, ppmi

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(id_to_word)
C = create_co_matrix(corpus, vocab_size, window_size=1)
W = ppmi(C)

# SVD
U, S, V = np.linalg.svd(W)
변환된 밀집벡터 저장
```

```
def preprocess(text):  
    text = text.lower()  
    text = text.replace('!', '!')  
    words = text.split(' ')  
    word_to_id = {}  
    id_to_word = {}  
  
    for word in words:  
        if word not in word_to_id:  
            new_id = len(word_to_id)  
            word_to_id[word] = new_id  
            id_to_word[new_id] = word  
  
    corpus = np.array([word_to_id[w] for w in words]) → 단어 목록에서 단어 ID 목록으로 변환  
  
    return corpus, word_to_id, id_to_word  
          단어 ID 목록
```

```
def create_co_matrix (corpus, vocab_size, window_size = 1):
    단어 ID 리스트   어휘수   윈도우 크기
    corpus_size = len(corpus)

    co_matrix = np.zeros ((vocab_size, vocab_size), dtype = np.int32)

    for idx, word_id in enumerate (corpus):
        for i in range (1, window_size + 1):
            left_idx = idx - i
            right_idx = idx + i

            if left_idx >= 0:
                left_word_id = corpus [left_idx]
                co_matrix [word_id, left_word_id] += 1

            if right_idx < corpus_size:
                right_word_id = corpus [right_idx]
                co_matrix [word_id, right_word_id] += 1

    return co_matrix
```

```
def cos_similarity (x, y, eps=1e-8):  
    np 배열이라고 가정.  
    nx = x / (np.sqrt(np.sum(x**2)) + eps) # x의 정규화  
    ny = y / (np.sqrt(np.sum(y**2)) + eps) # y의 정규화  
  
    return np.dot(nx, ny) → 두 벡터의 내적을 구함
```

```

def most_similar (query, word_to_id, id_to_word, word_matrix, top=5):
    # 정색어를 끄낸다.

    if query not in word_to_id:
        print ('%s (을)를 찾을 수 없습니다.' % query)
        return

    print ('\n[query] ' + query)
    query_id = word_to_id [query]

    query_vec = word_matrix [query_id]
    단어벡터들을 한 데 모으는 해결.
    기 해에는 대응하는 단어의 벡터가 저장되어 있다고 가정

    # 코사인 유사도 계산

    vocab_size = len (id_to_word)
    similarity = np.zeros (vocab_size)

    for i in range (vocab_size):
        similarity [i] = cos_similarity (word_matrix [i], query_vec)

    # 코사인 유사도를 기준으로 내림차순으로 출력

    count = 0
    for i in (-1 * similarity).argsort ():      → similarity 배열에 뒤로 부모의 인덱스를
                                                내림차순으로 정렬한 후 상위 원소들을 출력
        if id_to_word [i] == query:
            continue
        print ('%s: %s' % (id_to_word [i], similarity [i]))
        count += 1
        if count >= top:
            return

```

```
def ppmi (C, verbose = False, eps = 1e-8):
    동시발생 케널      기계상황 출현여부
    M = np.zeros_like (C, dtype = np.float32)
    N = np.sum (C)
    S = np.sum (C, axis = 0)
    total = C.shape[0] * C.shape[1]
    Cnt = 0

    for i in range (C.shape[0]):
        for j in range (C.shape[1]):
            pmi = np.log2 (C[i,j] * N / (S[j] * S[i]) + eps)
            M[i,j] = max (0, pmi)

            if verbose:
                Cnt += 1
                if Cnt % (total // 100) == 0:
                    print ('%.1f%% 완료' % (100 * Cnt / total))

    return M
```

CHAPTER 3. Word 2 vec

3. 1 추론 기반 기법과 신경망

- 분포가설

3. 1. 1 통계 기반 기법의 문제점

3. 1. 2 추론기반 기법 개요

3. 1. 3 신경망에서의 단어처리

3. 2 단순한 word2vec

- CBOW
- skip-gram

) word2vec에 사용되는 신경망

3. 2. 1 CBOW 모델의 추론 처리

3. 2. 2 CBOW 모델의 학습

3. 3. 2 워드 표현으로 변환

- 매개변수 초기화: $(6, 2) \rightarrow (6, 2, 7)$ 로 변환

```
import sys
sys.path.append('..')
from common.util import preprocess, create_contexts_target, convert_one_hot

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)

context, target = create_contexts_target(corpus, window_size=1)

vocab_size = len(word_to_id)
target = convert_one_hot(target, vocab_size)
contexts = convert_one_hot(contexts, vocab_size)
```

3.4.1 학습코드 구현

```
import sys
sys.path.append('..')

from common.trainer import Trainer
from common.optimizer import Adam
from simple_cbow import SimpleCBOW
from common.util import preprocess, create_contexts_target, convert_one_hot

window_size = 1
hidden_size = 5
batch_size = 3
max_epoch = 1000

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)

vocab_size = len(word_to_id)
contexts, target = create_contexts_target(corpus, window_size)
target = convert_one_hot(target, vocab_size)
contexts = convert_one_hot(contexts, vocab_size)

model = SimpleCBOW(vocab_size, hidden_size)
optimizer = Adam()
trainer = Trainer(model, optimizer)

trainer.fit(contexts, target, max_epoch, batch_size)
trainer.plot()

word_vecs = model.word_vecs
for word_id, word in id_to_word.items():
    print(word, word_vecs[word_id])
```

3.5 word2vec 보충

3.5.1 CBOW 모델과 손실함수

- w_{t-1} 과 w_{t+1} 이 일어난 후 w_t 가 일어날 확률

$$P(w_t | w_{t-1}, w_{t+1})$$

- 고차 페트로피 모자

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

정답레이블
(원핫벡터)
 w_t 에 해당하는 원소만 1



$$L = -\log P(w_t | w_{t-1}, w_{t+1})$$

\rightarrow "음의 로그 가능도 (negative log likelihood)



샘플 하나에 대한 손실함수를 매크로치 전체로 확장

$$L = -\frac{1}{T} \sum_{t=1}^T \log P(w_t | w_{t-1}, w_{t+1})$$

3.5.2 skip-gram 모델

$$- P(w_{t-1}, w_{t+1} | w_t)$$

$$= P(w_{t-1} | w_t) P(w_{t+1} | w_t)$$

↓ 손실함수 유도

$$L = -\log P(w_{t-1}, w_{t+1} | w_t)$$

$$= -\log P(w_{t-1} | w) P(w_{t+1} | w_t)$$

$$= -(\log P(w_{t-1} | w_t) + \log P(w_{t+1} | w_t))$$

$$\log xy = \log x + \log y$$

\rightarrow 매각별 손실함수를 구한 다음 모두 더함.

↓ 샘플 레이터 하나짜리를 매크로치 전체로 확장

$$L = -\frac{1}{T} \sum_{t=1}^T (\log P(w_{t-1} | w_t) + \log P(w_{t+1} | w_t))$$

완전연결계층에 의한 변화

```
import numpy as np  
  
c = np.array ([[1, 0, 0, 0, 0, 0, 0]])      # 입력  
W = np.random.randn(7,3)                    # 가중치  
h = np.matmul(c, W)                         # 중간노드  
print(h)
```

CBOW 모델 축출 처리

```
import sys  
sys.path.append('..')  
  
import numpy as np  
  
from common.layers import MatMul  
  
# 샘플 매개 변수  
c0 = np.array([[1, 0, 0, 0, 0, 0]])  
c1 = np.array([[0, 1, 0, 0, 0, 0]])  
  
# 가중치 초기화  
W_in = np.random.randn(7, 3)  
W_out = np.random.randn(3, 7)  
  
# 계층 생성  
in_layer0 = MatMul(W_in)  
in_layer1 = MatMul(W_in)  
out_layer = MatMul(W_out)  
  
# 순전파  
h0 = in_layer0.forward(c0) # 중간 레이어 계산  
h1 = in_layer1.forward(c1)  
h = 0.5 * (h0 + h1)  
s = out_layer.forward(h) → 각 단어의 저수 구함  
  
print(s)
```

```
def create_contexts_target (corpus, window_size = 1) :  
    target = corpus [window_size : -window_size]  
    contexts = []  
  
    for idx in range (window_size, len (corpus) - window_size) :  
        cs = []  
        for t in range (-window_size, window_size + 1) :  
            if t == 0 :  
                continue  
            cs. append (corpus [idx + t])  
        contexts. append (cs)  
  
    return np.array (contexts), np.array (target)
```

```

import sys
sys.path.append('..')
import numpy as np
from common.layers import MatMul, SoftmaxWithLoss

class SimpleCBOW:
    def __init__(self, vocab_size, hidden_size):
        V, H = vocab_size, hidden_size

        # 가중치 초기화
        W_in = 0.01 * np.random.randn(V, H).astype('f')
        W_out = 0.01 * np.random.randn(H, V).astype('f')

        # 계층 생성
        self.in_layer0 = MatMul(W_in)
        self.in_layer1 = MatMul(W_in)
        self.out_layer = MatMul(W_out)
        self.loss_layer = SoftmaxWithLoss()

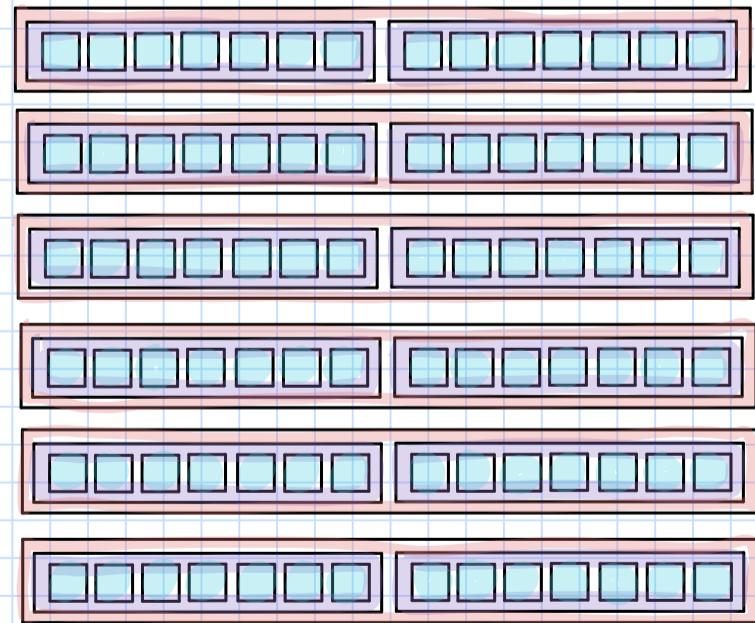
        # 모든 가중치와 기울기를 리스트에 넣는다.
        layers = [self.in_layer0, self.in_layer1, self.out_layer]
        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads

        # 인스턴스 변수에 단어의 분산표현을 저장한다.
        self.word_vecs = W_in

    def forward(self, contexts, target):
        h0 = self.in_layer0.forward(contexts[:, 0])
        h1 = self.in_layer1.forward(contexts[:, 1])
        h = (h0 + h1) * 0.5
        score = self.out_layer.forward(h)
        loss = self.loss_layer.forward(score, target)
        return loss

    def backward(self, dout=1):
        ds = self.loss_layer.backward(dout)
        da = self.out_layer.backward(ds)
        da *= 0.5
        self.in_layer1.backward(da)
        self.in_layer0.backward(da)
        return None

```



$[0]$ $[1]$

$[:, 0]$

$[:, 1]$

4.1 Word2vec 개선 1

4.1.1 Embedding 계층

4.1.2 Embedding 계층 구현

```
import numpy as np
```

```
W = np.arange(21).reshape(7,3)
```

4.2 word2vec 개선 2

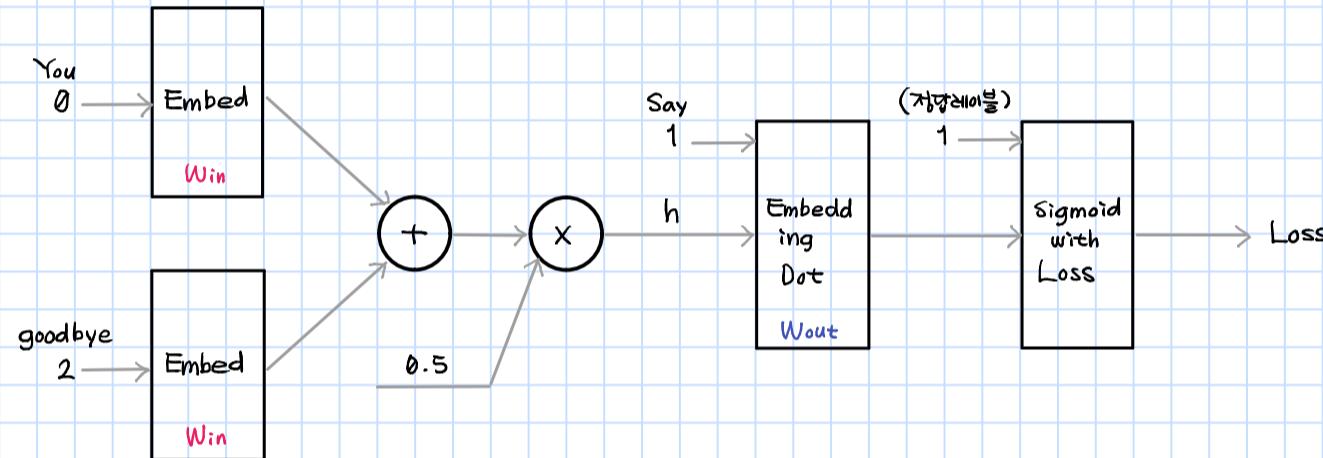
- 은닉층 이후의 처리 (해결법과 Softmax 계층의 계산)

- 네거티브 샘플링

4.2.1 은닉층 이후 계산의 문제점.

4.2.2 대주 분류에서 이진 분류로

4.2.3 시그모이드 함수와 교차 엔트로피 오차



4.2.5 네거티브 샘플링

- 적은 수의 부정적 예를 샘플링해 사용

4.2.6 네거티브 샘플링의 샘플링 기법

- 말뭉치의 등장 빈도 라이트를 기준으로 샘플링

(자주 등장하는 단어를 많이 추출, 드물게 등장하는 단어를 적게 추출)

$$- P'(w_i) = \frac{P(w_i)^{0.75}}{\sum_j^n P(w_j)^{0.75}}$$

→ 수제 후에도 확률 축합은 1이 되어야 하므로
분모로는 '수제 후 확률분포의 총합'이 필요

4.2.7 네거티브 샘플링 구현

4.3 개선판 Word2vec 학습

4.3.1 CBOW 모델 구현

4.3.2 CBOW 모델 학습 코드

```
import sys
sys.path.append('..')
import numpy as np
from common import config
import pickle
from common.trainer import Trainer
from common.optimizer import Adam
from cbow import CBOW
from common.util import create_contexts_target, to_cpu, to_gpu
from dataset import ptb

# 하이퍼파라미터 설정
window_size = 5
hidden_size = 100
batch_size = 100
max_epoch = 10

# 테이터 읽기
corpus, word_to_id, id_to_word = ptb.load_data('train')
vocab_size = len(word_to_id)

contexts, target = create_contexts_target(corpus, window_size)
if config.GPU:
    contexts, target = to_gpu(contexts), to_gpu(target)

# 모델 등 생성
model = CBOW(vocab_size, hidden_size, window_size, corpus)
optimizer = Adam()
trainer = Trainer(model, optimizer)

# 학습 시작
trainer.fit(contexts, target, max_epoch, batch_size)
trainer.plot()
```

원도우 크기 2 ~ 10개,
온갖 뉴스 수 50 ~ 500개 차도면 좋은 결과를 얻음.

나중에 사용할 수 있도록 필요한 데이터 저장

```
word_vecs = model.word_vecs  
if config.GPU:  
    word_vecs = to_cpu(word_vecs)  
params = {}  
params['word_vecs'] = word_vecs.astype(np.float16)  
params['word_to_id'] = word_to_id  
params['id_to_word'] = id_to_word  
pkl_file = 'cbow_params.pkl'  
with open(pkl_file, 'wb') as f:  
    pickle.dump(params, f, -1)
```

4.4 word2vec 낭을 주제

4.4.1 word2vec을 사용한 애플리케이션의 예

4.4.2 단어 벡터 평가방법

class Embedding:

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

def forward(self, idx):
    W, = self.params
    self.idx = idx      → 추출하는 해의 인덱스(단어 ID)를 배열로 저장
    out = W[idx]
    return out

def backward(self, dout):
    dW, = self.grads
    dW[:] = 0          → dW의 계상을 유지한 채 원소들을 0으로 덮어씀.

    for i, word_id in enumerate(self.idx):
        dW[word_id] += dout[i]

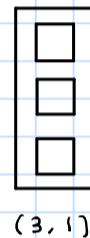
    # 혹은 np.add.at(dW, self.idx, dout)
    ↗ dout을 dW의 self.idx 위치에 더해줌.
```

class EmbeddingDot:

```
def __init__(self, W):  
    self.embed = Embedding(W)  
    self.params = self.embed.params  
    self.grads = self.embed.grads  
    self.cache = None  
  
def forward(self, h, idx):  
    target_W = self.embed.forward(idx)  
    out = np.sum(target_W * h, axis=1) → 내적 계산  
    ← 원소별 곱  
    self.cache = (h, target_W)  
    return out
```

def backward(self, dout):

```
h, target_W = self.cache  
dout = dout.reshape(dout.shape[0], 1) →  $\begin{matrix} \square & \square & \square \end{matrix} \Rightarrow \begin{matrix} \square \\ \square \\ \square \end{matrix}$   
dtarget_W = dout * h  
self.embed.backward(dttarget_W)  
dh = dout * target_W  
return dh
```



Class NegativeSamplingLoss :

```
def __init__(self, W, corpus, power = 0.75, sample_size = 5):  
    self.sample_size = sample_size  
  
    self.sampler = UnigramSampler(corpus, power, sample_size)  
  
    self.loss_layers = [SigmoidWithLoss() for _ in range(sample_size + 1)]  
    self.embed_dot_layers = [EmbeddingDot(W) for _ in range(sample_size + 1)]  
  
    self.params, self.grads = [], []  
  
    for layer in self.embed_dot_layers:  
        self.params += layer.params  
        self.grads += layer.grads
```

```
def forward(self, h, target):  
    # 긍정적 예의 태그  
    batch_size = target.shape[0]  
  
    negative_sample = self.sampler.get_negative_sample(target)
```

```
# 긍정적 예 순전파  
  
score = self.embed_dot_layers[0].forward(h, target)  
correct_label = np.ones(batch_size, dtype = np.int32)  
loss = self.loss_layers[0].forward(score, correct_label)
```

```
# 부정적 예 순전파  
  
negative_label = np.zeros(batch_size, dtype = np.int32)  
  
for i in range(self.sample_size):  
    negative_target = negative_sample[:, i]  
  
    score = self.embed_dot_layers[1+i].forward(h, negative_target)  
    loss += self.loss_layers[1+i].forward(score, negative_label)
```

```
return loss
```

```
def backward(self, dout = 1):  
    dh = 0  
  
    for l0, l1 in zip(self.loss_layers, self.embed_dot_layers):  
        dscore = l0.backward(dout)  
        dh += l1.backward(dscore)  
  
    return dh
```

) 부정적 예를 다루는 계층이
sample-size 개
+
긍정적 예를 다루는 계층 1개
· loss_layers[0]
· embed_dot_layers[0]

```

import sys
sys.path.append('..')
import numpy as np

from common.layers import Embedding
from ch04.negative_sampling_layer import NegativeSamplingLoss

class SimpleCBOW:

    def __init__(self, vocab_size, hidden_size, window_size, corpus):
        V, H = vocab_size, hidden_size

        # 가중치 초기화
        W_in = 0.01 * np.random.randn(V, H).astype('f')
        W_out = 0.01 * np.random.randn(H, V).astype('f')

        # 계층 생성
        self.in_layers = []
        for i in range(2 * window_size):
            layer = Embedding(W_in)      # Embedding 계층 사용
            self.in_layers.append(layer)

        self.ns_loss = NegativeSamplingLoss(W_out, corpus, power=0.75, sample_size=5)

        # 모든 가중치와 기울기를 리스트에 넣운다.
        layers = self.in_layers + [self.ns_loss]
        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads

        # 인스턴스 변수에 단어의 분산표현을 저장한다.
        self.word_vecs = W_in

```

```
def forward (self, contexts, target):  
    h = 0  
  
    for i, layer in enumerate (self.in_layers):  
        h += layer.forward (contexts[:, i])  
  
    h *= 1 / len(self.in_layers)  
  
    loss = self.ns_loss.forward (h, target)  
  
    return loss  
  
def backward (self, dout=1):  
    dout = self.ns_loss.backward (dout)  
  
    dout *= 1 / len (self.in_layers)  
  
    for layer in self.in_layers:  
        layer.backward (dout)  
  
    return None
```

5.1 확률과 언어모델

5.1.1 word2vec을 확률 관점에서 바라보다.

5.1.2 언어모델

- w_1, \dots, w_m 이라는 m 개 단어로 된 문장에서단어가 w_1, \dots, w_m 이라는 순서로 출현할 확률: $P(w_1, \dots, w_m) \rightarrow$ 동시확률

↓
사후확률을 이용하여 분해 (공세b정리를 이용)

$$P(w_m | w_1, \dots, w_{m-1}) P(w_{m-1} | w_1, \dots, w_{m-2})$$

$$\dots P(w_3 | w_1, w_2) P(w_2 | w_1) P(w_1)$$

$$= \prod_{t=1}^m P(w_t | w_1, \dots, w_{t-1})$$

● : '총곱'을 의미

$$P(A, B) = P(A|B) P(B)$$

$$P(w_1, \dots, w_{m-1}, w_m) = \underbrace{P(A, w_m)}_{A} = P(w_m | A) P(A)$$

↓
A(w_1, \dots, w_{m-1})에 대해서 다시 서 번역

$$P(A) = \underbrace{P(w_1, \dots, w_{m-1})}_{A'} = P(A', w_{m-1}) = P(w_{m-1} | A') P(A')$$

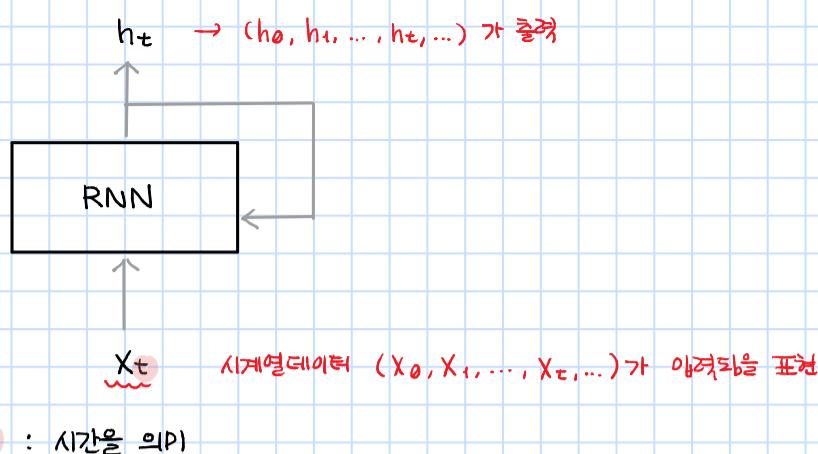
5.1.3 CBOW 모델을 언어 모델로?

$$P(w_1, \dots, w_m) = \prod_{t=1}^m P(w_t | w_1, \dots, w_{t-1}) \approx \prod_{t=1}^m P(w_t | w_{t-2}, w_{t-1})$$

- 한계: 예전 단의 단어 순서가 무시됨

5.2 RNN 이란

5.2.1 순환하는 신경망



5.2.2 순환구조 평가

$$- h_t = \tanh(h_{t-1}W_h + x_tW_x + b)$$

- W_x : 이전 x 를 출력 h 로 변환하기 위한 가중치
- W_h : RNN 출력을 다음시간의 출력으로 변환하기 위한 가중치

5.2.3 BPTT

— 시계열 데이터의 시간 크기가 커지는 것에 비례하여 BPTT가 소비하는 컴퓨팅 자원도 증가

— 시간 크기가 커지면 역전파 시의 기울기가 불안정해짐

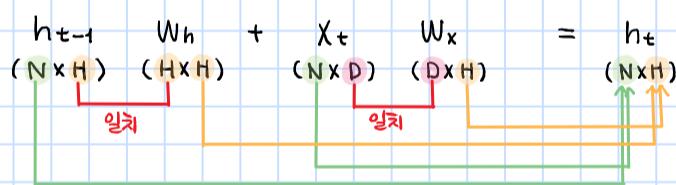
5.2.4 Truncated BPTT

5.2.5 Truncated BPTT의 미니배치 학습

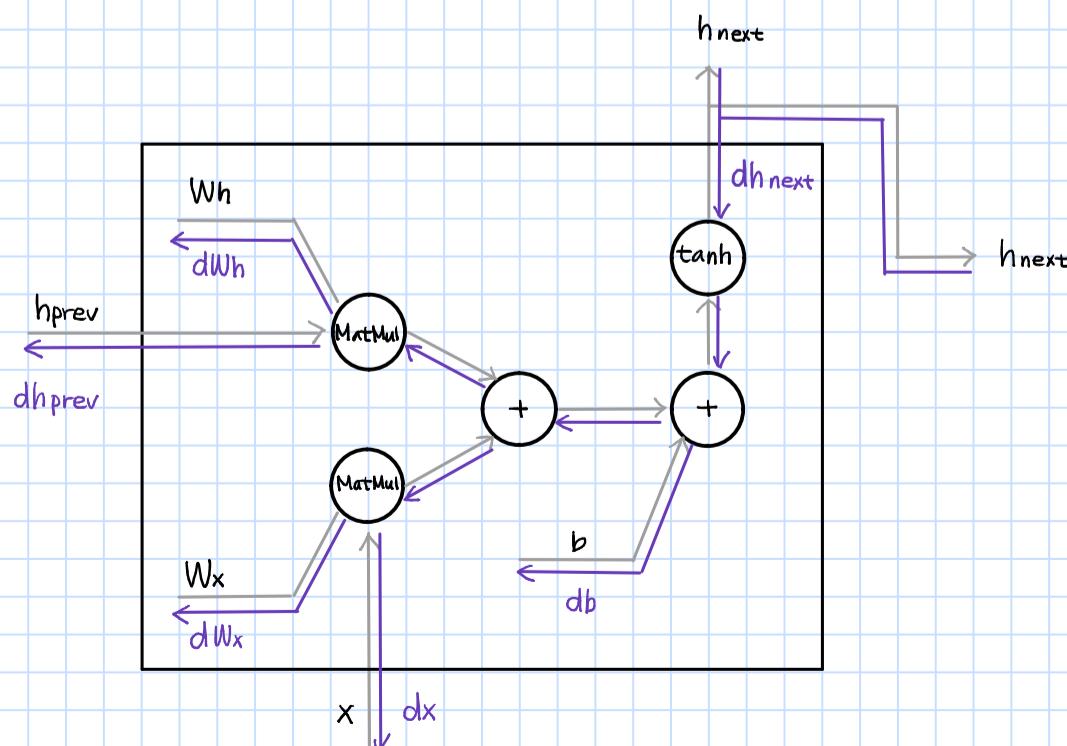
5.3 RNN 구현

5.3.1 RNN 계층 구현

$$- h_t = \tanh(h_{t-1}W_h + x_tW_x + b)$$



- N : 미니배치 크기
- D : 이전 배치의 차원 수
- H : 은닉상태 벡터의 차원 수



5.3.2 TimeRNN 계층 구조

- RNN 계층의 은닉상태를 TimeRNN 계층에서 관리
- RNN 계층 사이에서 은닉상태를 '인계하는 책임'을 생각하지 않아도 됨

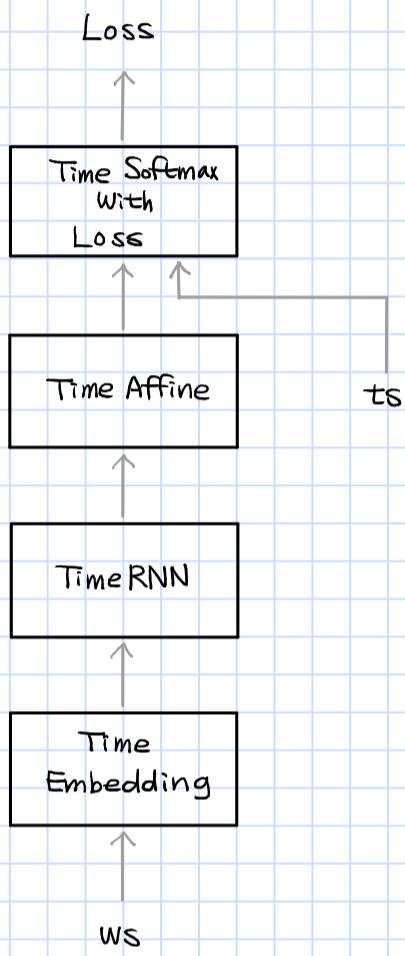
5.4 시계열데이터 처리 계층 구조

5.4.1 RNNLM의 전체구조

5.4.2 Time 계층 구조

5.5 RNNLM 학습과 평가

5.5.1 RNNLM 구조



class RNN:

```
def __init__(self, Wx, Wh, b):
    self.params = [Wx, Wh, b]
    self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
    self.cache = None

def forward(self, x, h_prev):
    Wx, Wh, b = self.params
    t = np.matmul(h_prev, Wh) + np.matmul(x, Wx) + b
    h_next = np.tanh(t)

    self.cache = (x, h_prev, h_next)
    return h_next

def backward(self, dh_next):
    Wx, Wh, b = self.params
    x, h_prev, h_next = self.cache

    dt = dh_next * (1 - h_next ** 2)
    db = np.sum(dt, axis=0)
    dWh = np.matmul(h_prev.T, dt)
    dh_prev = np.matmul(dt, Wh.T)
    dWx = np.matmul(x.T, dt)

    dx = np.matmul(dt, Wx.T)

    self.grads[0][...] = dWx
    self.grads[1][...] = dWh
    self.grads[2][...] = db

    return dx, dh_prev
```

class TimeRNN:

```
def __init__(self, Wx, Wh, b, stateful = False):  
    self.params = [Wx, Wh, b]  
    self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
```

self.layers = None → 다수의 RNN 계층을 리스트로 저장하는 방식

self.h, self.dh = None, None

self.stateful = stateful

```
def set_state(self, h): TimeRNN 계층의 은닉상태를 설정하는 메서드
```

self.h = h

```
def reset_state(self): 은닉상태를 초기화하는 메서드
```

self.h = None

```
def forward(self, xs):
```

Wx, Wh, b = self.params

N, T, D = xs.shape
D: 배치 크기 T: 시계열 차원수

D, H = Wx.shape

self.layers = []

hs = np.empty((N, T, H), dtype='f')

if not self.stateful or self.h is None:

self.h = np.zeros((N, H), dtype='f')

for t in range(T):

layer = RNN(*self.params)

self.h = layer.forward(xs[:, t, :], self.h)

hs[:, t, :] = self.h

self.layers.append(layer)

return hs

) 각 시가 t의 은닉상태 h를 계산하고
이를 hs에 해당 인덱스(시가)의 값으로 설정



```
def backward (self, dhs):
    Wx, Wh, b = self.params
    N, T, H = dhs.shape
    D, H = Wx.shape

    dxs = np.empty ((N, T, D), dtype='f')
    dh = 0
    grads = [0, 0, 0]

    for t in reversed (range(T)):
        layer = self.layers [t]
        dx, dh = layer.backward (dhs [:, t, :] + dh)      # 합산된 기울기
        dxs [:, t, :] = dx

        for i, grad in enumerate (layer.grads):
            grads [i] += grad      → 각 층 가중치 기울기를 합산

        for i, grad in enumerate (grads):
            self.grads [i][...] = grad      → 최종 결과를 덮어씀
            self.dh = dh

    return dxs
```

```

import sys
sys.path.append('..')
import numpy as np
from common.time_layers import *

class SimpleRnnlm:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        # 가중치 초기화
        embed_W = (rn(V,D) / 100).astype('f')
        rnn_Wx = (rn(D,H) / np.sqrt(D)).astype('f')          → Xavier 초기값 사용 : 이전 계층의 노드가 n개라면
        rnn_Wh = (rn(H,H) / np.sqrt(H)).astype('f')          표준편차가  $\frac{1}{\sqrt{n}}$  일 때로 값들을 초기화
        rnn_b = np.zeros(H).astype('f')

        affine_W = (rn(H,V) / np.sqrt(H)).astype('f')
        affine_b = np.zeros(V).astype('f')

        # 계층 생성
        self.layers = [
            TimeEmbedding(embed_W),
            TimeRNN(rnn_Wx, rnn_Wh, rnn_b, stateful=True),
            TimeAffine(affine_W, affine_b)
        ]
        self.loss_layer = TimeSoftmaxWithLoss()
        self.rnn_layer = self.layers[1]

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

```



```
def forward (self, xs, ts):  
    for layer in self.layers:  
        xs = layer.forward(xs)  
  
    loss = self.loss_layer.forward(xs, ts)  
  
    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  
  
def reset_state (self):  
    self.rnn_layer.reset_state()
```