# 시계열 분석

시계열 문제

**AR, MA, ARMA, ARIMA**

**순환 신경망(RNN)**

**RNN 구조**

**LSTM**

**게이트 순환 신경망(GRU)**

**RNN, LSTM, GRU 성능 비교**

**양방향 RNN**

# 시계열 문제

# 1 시계열 문제

- **시계열 문제**
  - 특정 대상의 시간에 따라 변하는 데이터를 사용하여 추이를 분석하는 것
  - 주가/환율 변동 및 기온/습도 변화 등이 대표적인 시계열 분석
  - 추세를 파악하거나 향후 전망 등을 예측하기 위한 용도로 시계열 분석을 사용

# 1 시계열 문제

- **시계열 문제**
  - 시계열 형태(the components of time series)는 데이터 변동 유형에 따라 불규칙 변동, 추세 변동, 순환 변동, 계절 변동으로 구분할 수 있음

  - **불규칙 변동**(irregular variation)**:** 시계열 자료에서 시간에 따른 규칙적인 움직임과 달리 어떤 규칙성이 없어 예측 불가능하고 우연적으로 발생하는 변동을 의미
    - 전쟁, 홍수, 화재, 지진, 파업

  - **추세 변동**(trend variation)**:** 시계열 자료가 갖는 장기적인 변화 추세를 의미 이때 추세란 장기간에 걸쳐 지속적으로 증가·감소하거나 또는 일정한 상태(stationary)를 유지하려는 성향을 의미하기 때문에 짧은 기간 동안에는 추세 변동을 찾기 어려운 단점이 있음
    - 국내총생산(GDP), 인구증가율

# 7.1 시계열 문제

- **시계열 문제**
  - **순환 변동**(cyclical variation)**:** 대체로 2~3년 정도의 일정한 기간을 주기로 순환적으로 나타나는 변동을 의미
    - 1년 이내 주기로 곡선을 그리며 추세 변동에 따라 변동하는 것으로, 경기 변동
  - **계절 변동**(seasonal variation)**:** 시계열 자료에서 보통 계절적 영향과 사회적 관습에 따라 1년 주기로 발생하는 것을 의미 보통 계절에 따라 순환하며 변동하는 특성이 있음

# 7.1 시계열 문제

- **시계열 문제**
  - 규칙적 시계열과 불규칙적 시계열로 나눌 수 있음
  - 규칙적 시계열은 트렌드와 분산이 불변하는 데이터이며, 불규칙적 시계열은 트렌드 혹은 분산이 변화하는 시계열 데이터
  - 시계열 데이터를 잘 분석한다는 것은 불규칙성을 갖는 시계열 데이터에 특정한 기법이나 모델을 적용하여 규칙적 패턴을 찾거나 예측하는 것을 의미
  - 불규칙적 시계열 데이터에 규칙성을 부여하는 방법으로는 AR, MA, ARMA, ARIMA 모델을 적용하는 것이 가장 널리 알려져 있음
  - 딥러닝을 이용하여 시계열 데이터의 연속성을 기계 스스로 찾아내도록 하는 방법이 더 좋은 성능을 내고 있음

# AR, MA, ARMA, ARIMA

# 7.2 AR, MA, ARMA, ARIMA

- **AR, MA, ARMA, ARIMA**
  - ❖ 시계열 분석은 독립 변수(independent variable)를 사용하여 종속 변수(dependent variable)를 예측하는 일반적인 머신 러닝에서 시간을 독립 변수로 사용한다는 특징이 있음
  - ❖ 독립 변수로 시간을 사용하는 특성 때문에 분석하는 데 있어 일반적인 방법론들과 차이가 있는데, 그 차이를 AR, MA, ARMA, ARIMA 모형으로 자세히 살펴보자

# 7.2 AR, MA, ARMA, ARIMA

## AR 모델

- AR(AutoRegression)(자기 회귀) 모델은 이전 관측 값이 이후 관측 값에 영향을 준다는 아이디어에 대한 모형으로 자기 회귀 모델이라고도 함
- AR에 대한 수식은 다음과 같음

$$\underset{①}{Z_t} = \underbrace{\Phi_1 Z_{t-1} + \Phi_2 Z_{t-2} + \cdots + \Phi_p Z_{t-p}}_{②} + \underset{③}{a_t}$$

- ①은 시계열 데이터에서 현재 시점을 의미하며, ②는 과거가 현재에 미치는 영향을 나타내는 모수(Φ)에 시계열 데이터의 과거 시점을 곱한 것
- 마지막으로 ③은 시계열 분석에서 오차 항을 의미하며 백색 잡음이라고도 함
- 수식은 p 시점을 기준으로 그 이전의 데이터에 의해 현재 시점의 데이터가 영향을 받는 모형이라고 할 수 있음

# 7.2 AR, MA, ARMA, ARIMA

- **MA 모델**
  - MA(Moving Average)(이동 평균) 모델은 트렌드(평균 혹은 시계열 그래프에서 y 값)가 변화하는 상황에 적합한 회귀 모델
  - MA에 대한 수식은 다음과 같음

$$\underbrace{Z_t}_{①} = \underbrace{\theta_1 a_{t-1} + \theta_2 a_{t-2} + \cdots + \theta_p a_{t-p}}_{②} + \underbrace{a_t}_{③}$$

  - ①은 시계열 데이터에서 현재 시점을 의미하며, ②는 매개변수(θ)에 과거 시점의 오차를 곱한 것
  - 마지막으로 ③은 오차 항을 의미
  - 수식은 AR 모델처럼 이전 데이터의 '상태'에서 현재 데이터의 상태를 추론하는 것이 아닌, 이전 데이터의 오차에서 현재 데이터의 상태를 추론하겠다는 의미

# 7.2 AR, MA, ARMA, ARIMA

- **ARMA 모델**
  - ① ARMA(AutoRegressive Moving Average)(자동 회귀 이동 평균) 모델은 AR과 MA를 섞은 모델로 연구 기관에서 주로 사용
  - 즉, AR, MA 두 가지 관점에서 과거의 데이터를 사용하는 것이 ARMA
  - 이동 평균 모델에서는 윈도우라는 개념을 사용하는데, 시계열을 따라 윈도우 크기만큼 슬라이딩(moving)된다고 하여 이동 평균 모델이라고 함
  - 이동 평균 모델에서 사용하는 수식은 다음과 같음

$$Z_t = a + \Phi_1 Z_{t-1} + \cdots + \Phi_p Z_{t-p} + \theta_1 a_{t-1} + \cdots + \theta_q a_{t-q} + a_t$$

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**
  - ARIMA(AutoRegressive Integrated Moving Average)(자동 회귀 누적 이동 평균) 모델은 자기 회귀와 이동 평균을 둘 다 고려하는 모형인데, ARMA와 달리 과거 데이터의 선형 관계뿐만 아니라 추세(cointegration)까지 고려한 모델
  - ARIMA는 파이썬 코드를 이용하여 직접 살펴보자

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**
  - ❖ statsmodels 라이브러리를 이용하여 ARIMA 모델을 구현하는데, 절차는 다음과 같음

**1.** ARIMA() 함수를 호출하여 사용하는데, ARIMA(p,d,q) 함수에서 쓰는 파라미터는 다음과 같음

**p:** 자기 회귀 차수

**d:** 차분 차수

**q:** 이동 평균 차수

**2.** fit() 메서드를 호출하고 모델에 데이터를 적용하여 훈련시킴

**3.** predict() 메서드를 호출하여 미래의 추세 및 동향에 대해 예측

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**

  **statsmodels 라이브러리**

  ❖ statsmodels는 다음 통계 분석 기능을 제공하는 파이썬 패키지

  - 검정 및 추정(test and estimation)
  - 회귀 분석(regression analysis)
  - 시계열 분석(time-series analysis)

  ❖ 파이썬에서 사용하려면 pip install statsmodels 명령으로 사전 설치 작업이 필요함

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**
  - 예제에서 ARIMA(5,1,0)을 적용해 보겠음
  - 자기 회귀 차수를 5로 설정하고 차분 차수는 1을 사용
  - 시계열을 정지 상태로 만들고 이동 평균 차수는 0을 사용
  - 예제는 두 단계로 진행
  - 첫 번째 단계에서 ARIMA() 함수를 사용하여 간단한 오차 정보만 보여 주는 예제를 먼저 진행
  - 두 번째 단계에서 첫 번째 단계를 확장하여 실제 예측해 보자(코드를 좀 더 단순화하여 이해하기 쉽게 하기 위해 두 단계로 나누어 진행)

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**
  - 먼저 statsmodels 라이브러리를 설치

```
> conda install -c conda-forge statsmodels
```

  - 혹은

```
> pip install statsmodels
```

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**
  - 설치가 완료되었다면, 첫 번째 단계의 예제를 구현하는 데 필요한 라이브러리와 데이터를 호출
  - 데이터셋은 7장 예제 파일의 data 폴더에 있는 sales.csv 파일을 사용하며, 자전거

**코드 7-1** ARIMA() 함수를 호출하여 sales 데이터셋에 대한 예측

```
from pandas import read_csv ------ 파이썬 판다스 라이브러리의 read_csv() 메서드를 사용해서 외부 TEXT 파일,
from pandas import datetime        CSV 파일을 불러와서 DataFrame으로 저장
from pandas import DataFrame
from statsmodels.tsa.arima_model import ARIMA
from matplotlib import pyplot

def parser(x): ------ 시간을 표현하는 함수 정의
    return datetime.strptime('199'+x, '%Y-%m') ------ strptime()은 날짜와 시간 정보를
                                                      문자열로 바꾸어 주는 메서드

series = read_csv('../chap7/data/sales.csv', header=0, parse_dates=[0], index_col=0,
                  squeeze=True, date_parser=parser) ------ 자전거 매출에 대한 CSV 데이터 호출
```

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**

```
model = ARIMA(series, order=(5,1,0)) ------ ARIMA() 함수 호출
model_fit = model.fit(disp=0) ------------------------------ 모형을 적용할 때 많은 디버그 정보가 제공되는데
print(model_fit.summary()) ------ 모델에 대한 정보 표시          disp 인수를 0으로 설정하여 이 기능을 비활성화
residuals = DataFrame(model_fit.resid) ------ DataFrame에 모델에 대한 오차 정보를 residuals에 저장
residuals.plot() ------ residuals 정보를 시각적으로 표현
pyplot.show()
residuals.plot(kind='kde')
pyplot.show()
print(residuals.describe())
```

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**
  - ❖ 코드를 실행하면 ARIMA() 함수를 호출하여 sales 데이터셋에 대한 정보를 보여 줌

```
ARIMA Model Results
==============================================================================
Dep. Variable:                D.Sales   No. Observations:                   35
Model:                 ARIMA(5, 1, 0)   Log Likelihood                -197.350
Method:                       css-mle   S.D. of innovations             66.436
Date:                Sun, 02 Aug 2020   AIC                            408.699
Time:                        10:28:58   BIC                            419.587
Sample:                    02-01-1991   HQIC                           412.458
                         - 12-01-1993
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
```

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**

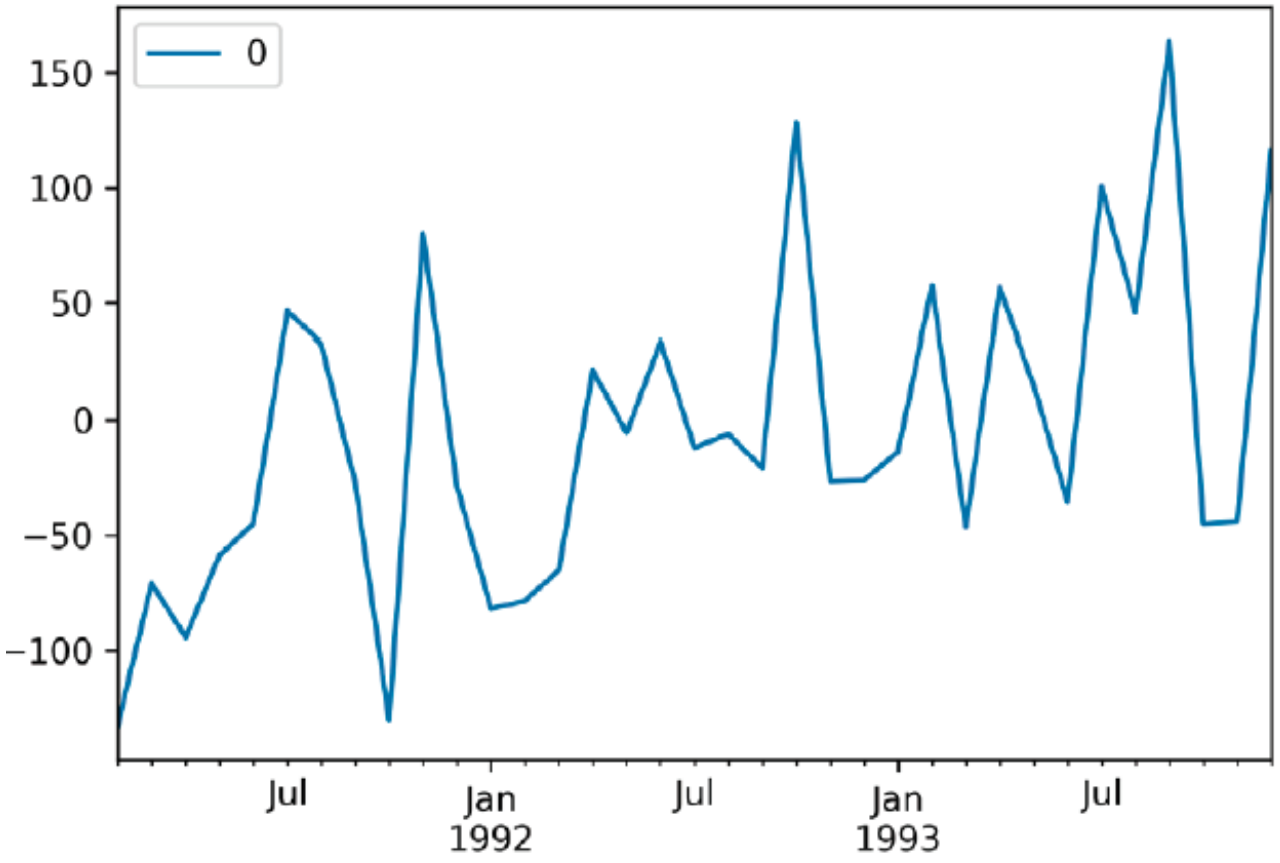
| | | | | | | |
|---|---|---|---|---|---|---|
| const | 12.4256 | 3.774 | 3.292 | 0.001 | 5.028 | 19.823 |
| ar.L1.D.Sales | -1.0850 | 0.188 | -5.764 | 0.000 | -1.454 | -0.716 |
| ar.L2.D.Sales | -0.6688 | 0.283 | -2.365 | 0.018 | -1.223 | -0.114 |
| ar.L3.D.Sales | -0.4426 | 0.297 | -1.489 | 0.136 | -1.025 | 0.140 |
| ar.L4.D.Sales | -0.0495 | 0.288 | -0.172 | 0.864 | -0.614 | 0.515 |
| ar.L5.D.Sales | 0.1652 | 0.197 | 0.840 | 0.401 | -0.220 | 0.551 |

Roots

| | Real | Imaginary | Modulus | Frequency |
|---|---|---|---|---|
| AR.1 | -1.1401 | -0.4612j | 1.2298 | -0.4388 |
| AR.2 | -1.1401 | +0.4612j | 1.2298 | 0.4388 |
| AR.3 | 0.0222 | -1.2562j | 1.2564 | -0.2472 |
| AR.4 | 0.0222 | +1.2562j | 1.2564 | 0.2472 |
| AR.5 | 2.5355 | -0.0000j | 2.5355 | -0.0000 |

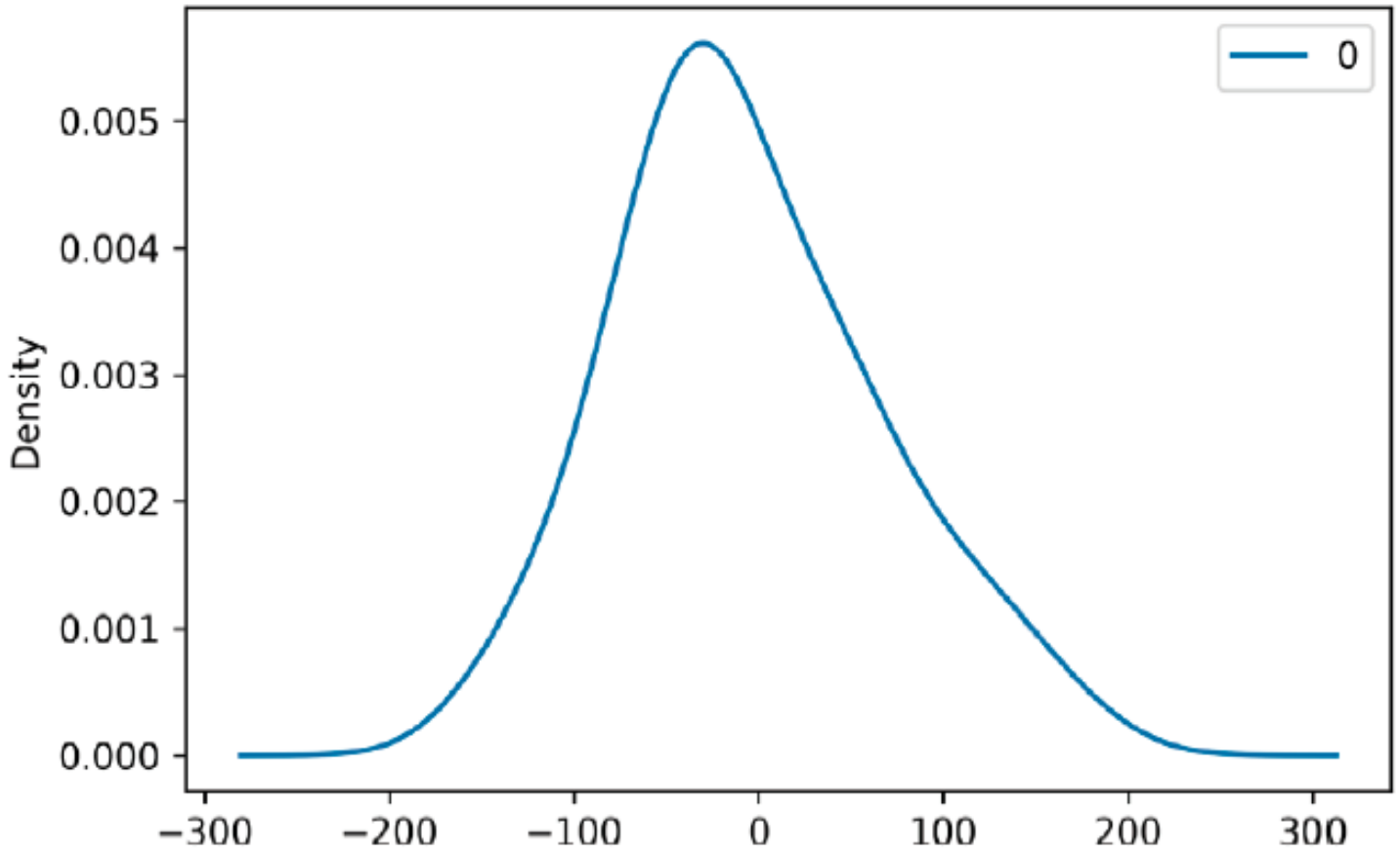
# 7.2 AR, MA, ARMA, ARIMA

▼ 그림 7-1 예제에 대한 오차 정보



# 7.2 AR, MA, ARMA, ARIMA

▼ 그림 7-2 예제에 대한 밀도 정보



# 7.2 AR, MA, ARMA, ARIMA

```
count    35.000000
mean     -5.569266
std      70.272666
min    -132.525611
25%     -45.563800
50%     -20.763477
75%      39.933189
max     163.552115
```

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**
  - 이와 같이 실행 결과는 오류 분포가 표시되는데, 결과를 보면 예측이 치우쳐 있음을 확인할 수 있음(오류 평균(mean) 값이 0이 아님)
  - 두 번째 단계로 ARIMA() 함수를 사용한 예측을 진행해 보자
  - 먼저 필요한 라이브러리를 호출

**코드 7-2** statsmodels 라이브러리를 이용한 sales 데이터셋 예측

```
import numpy as np
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error

def parser(x):
    return datetime.strptime('199'+x, '%Y-%m')

series = read_csv('../chap7/data/sales.csv', header=0, parse_dates=[0], index_col=0,
                  squeeze=True, date_parser=parser)
```

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**

```
X = series.values
X = np.nan_to_num(X)
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)] ------ train과 test로 데이터셋 분리
history = [x for x in train]
predictions = list()
for t in range(len(test)): ------ test 데이터셋의 길이(13번)만큼 반복하여 수행
    model = ARIMA(history, order=(5,1,0)) ------ ARIMA() 함수 호출
    model_fit = model.fit(disp=0)
    output = model_fit.forecast() ------ forecast() 메서드를 사용하여 예측 수행
    yhat = output[0] ------ 모델 출력 결과를 yhat에 저장
    predictions.append(yhat)
    obs = test[t]
    history.append(obs)
    print('predicted=%f, expected=%f' % (yhat, obs)) ------ 모델 실행 결과를 predicted로 출력하고,
                                                        test로 분리해 둔 데이터를
                                                        expected로 사용하여 출력
```

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**

```
error = mean_squared_error(test, predictions) ------ 손실 함수로 평균 제곱 오차 사용
print('Test MSE: %.3f' % error)
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
```

# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**
  - 다음은 statsmodels 라이브러리를 이용한 sales 데이터셋에 대한 예측을 실행한 결과

```
predicted=354.377730, expected=346.300000
predicted=288.627290, expected=329.700000
predicted=382.817953, expected=445.400000
predicted=339.543839, expected=325.900000
predicted=392.897253, expected=449.300000
predicted=354.488010, expected=411.300000
```

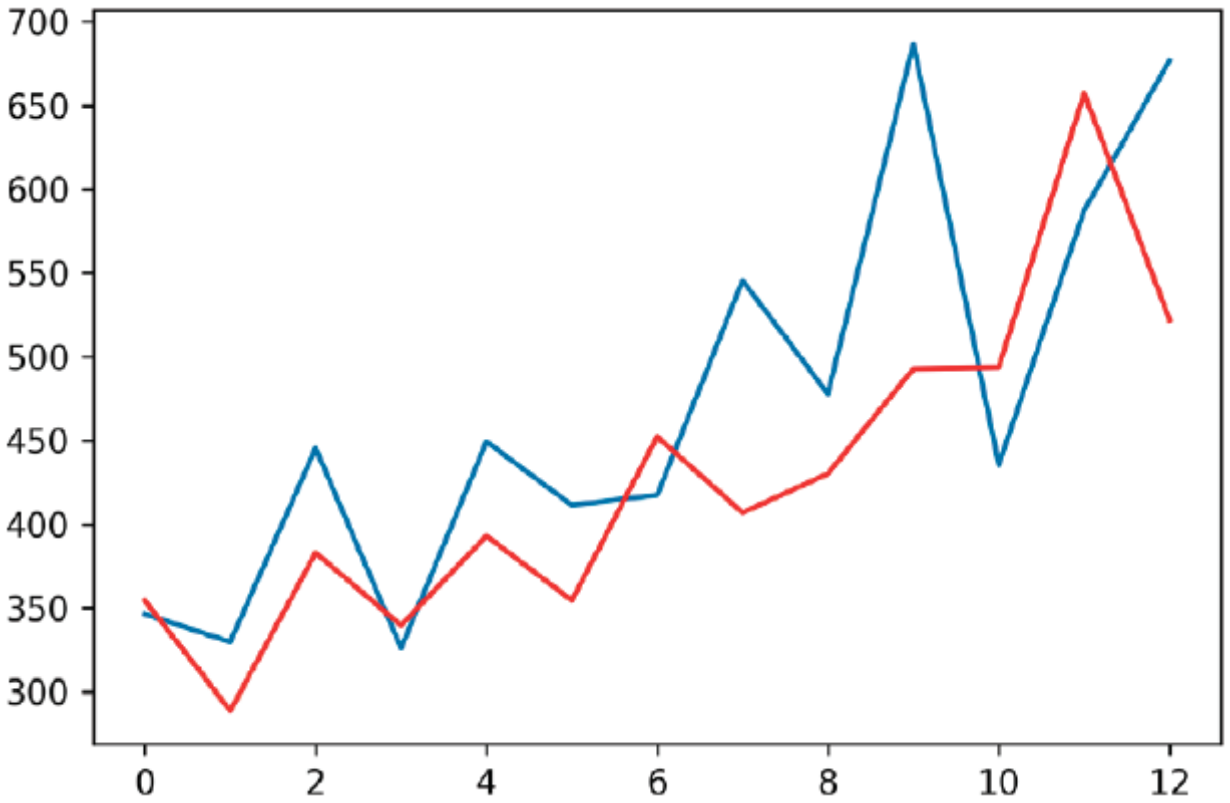
# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**

```
predicted=452.200100, expected=417.400000
predicted=406.806117, expected=545.500000
predicted=430.162052, expected=477.600000
predicted=492.745314, expected=687.000000
predicted=493.604679, expected=435.300000
predicted=657.397158, expected=587.300000
predicted=522.091111, expected=676.900000
Test MSE: 8074.991
```

# 7.2 AR, MA, ARMA, ARIMA

▼ 그림 7-3 예제에 대한 예측 결과



# 7.2 AR, MA, ARMA, ARIMA

- **ARIMA 모델**
  - 실제 데이터(빨간색)와 모형 실행 결과(파란색)를 표시한 그림이 만들어졌음
  - 데이터가 우상향 추세를 나타내고 있으므로, 자전거 판매가 향후에도 계속 증가할 것임을 예측할 수 있음
  - 이와 같이 ARIMA를 사용할 경우 데이터 경향을 파악해서 미래를 예측할 수 있음

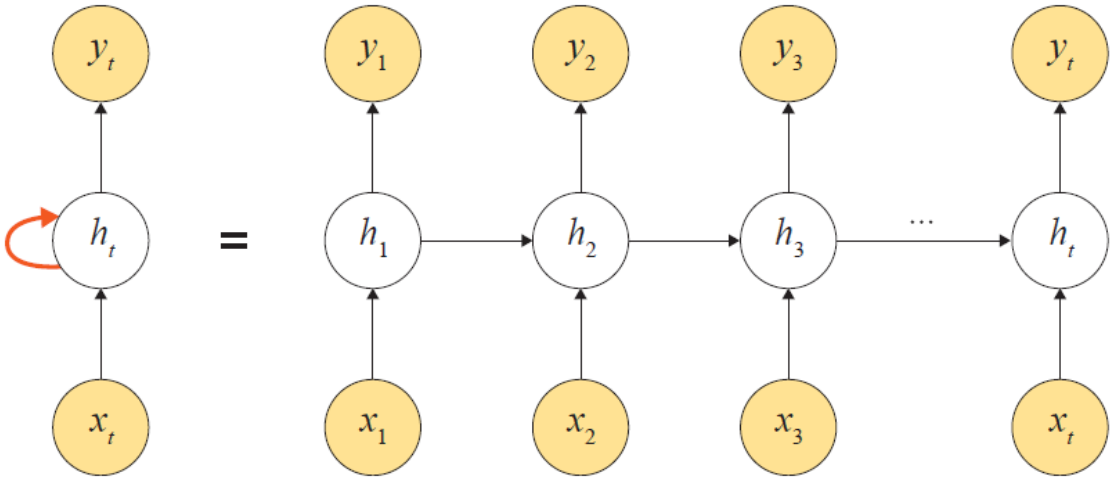
# 순환 신경망(RNN)

# 7.3 순환 신경망(RNN)

- **순환 신경망(RNN)**
  - RNN(Recurrent Neural Network)은 시간적으로 연속성이 있는 데이터를 처리하려고 고안된 인공 신경망
  - RNN의 'Recurrent(반복되는)'는 이전 은닉층이 현재 은닉층의 입력이 되면서 '반복되는 순환 구조를 갖는다'는 의미
  - RNN이 기존 네트워크와 다른 점은 '기억(memory)'을 갖는다는 것
  - 이때 기억은 현재까지 입력 데이터를 요약한 정보라고 생각하면 됨
  - 새로운 입력이 네트워크로 들어올 때마다 기억은 조금씩 수정되며, 결국 최종적으로 남겨진 기억은 모든 입력 전체를 요약한 정보가 됨

# 7.3 순환 신경망(RNN)

▼ 그림 7-4 순환 신경망(RNN)




- 그림과 같이 첫 번째 입력이 들어오면 첫 번째 기억($h_1$)이 만들어지고, 두 번째 입력이 들어오면 기존 기억($h_1$)과 새로운 입력을 참고하여 새 기억($h_2$)을 만듦
- 입력 길이만큼 이 과정을 얼마든지 반복할 수 있음
- 즉, RNN은 외부 입력과 자신의 이전 상태를 입력받아 현재 상태를 갱신

# 7.3 순환 신경망(RNN)

- **순환 신경망(RNN)**
  - ❖ RNN은 입력과 출력에 따라 유형이 다양함

1. **일대일:** 순환이 없기 때문에 RNN이라고 말하기 어려우며, 순방향 네트워크가 대표적 사례

2. **일대다:** 입력이 하나이고, 출력이 다수인 구조
   이미지를 입력해서 이미지에 대한 설명을 문장으로 출력하는 이미지 캡션(image captioning)이 대표적 사례

3. **다대일:** 입력이 다수이고 출력이 하나인 구조로, 문장을 입력해서 긍정/부정을 출력하는 감성 분석기에서 사용

# 7.3 순환 신경망(RNN)

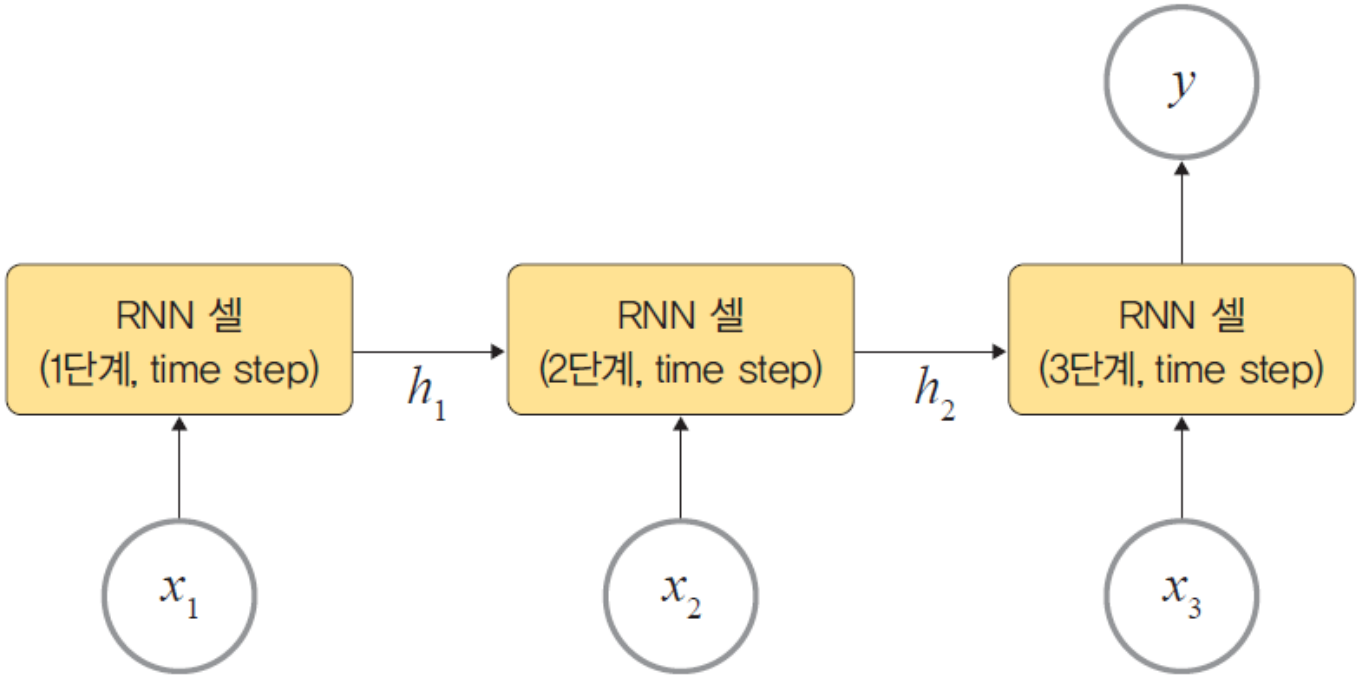
- **순환 신경망(RNN)**
  - 다대일에 대한 모델은 텐서플로 2에서 다음과 같이 구현함
  - 다음은 예시 코드

```
In_layer = tf.keras.input(shape=(3,1), name='input')
RNN_layer = tf.keras.layers.SimpleRNN(100, name='RNN')(In_layer)
Out_layer = tf.keras.layers.Dense(1, name='output')(RNN_layer)
```

# 7.3 순환 신경망(RNN)

- **순환 신경망(RNN)**
  - ❖ 코드를 구조화하면 다음 그림과 같음
  - ❖ 코드는 입력과 출력 사이에 하나의 RNN 셀(cell)만 가지고 있는 것에 주의해야 함

▼ 그림 7-5 다대일 모델



# 7.3 순환 신경망(RNN)

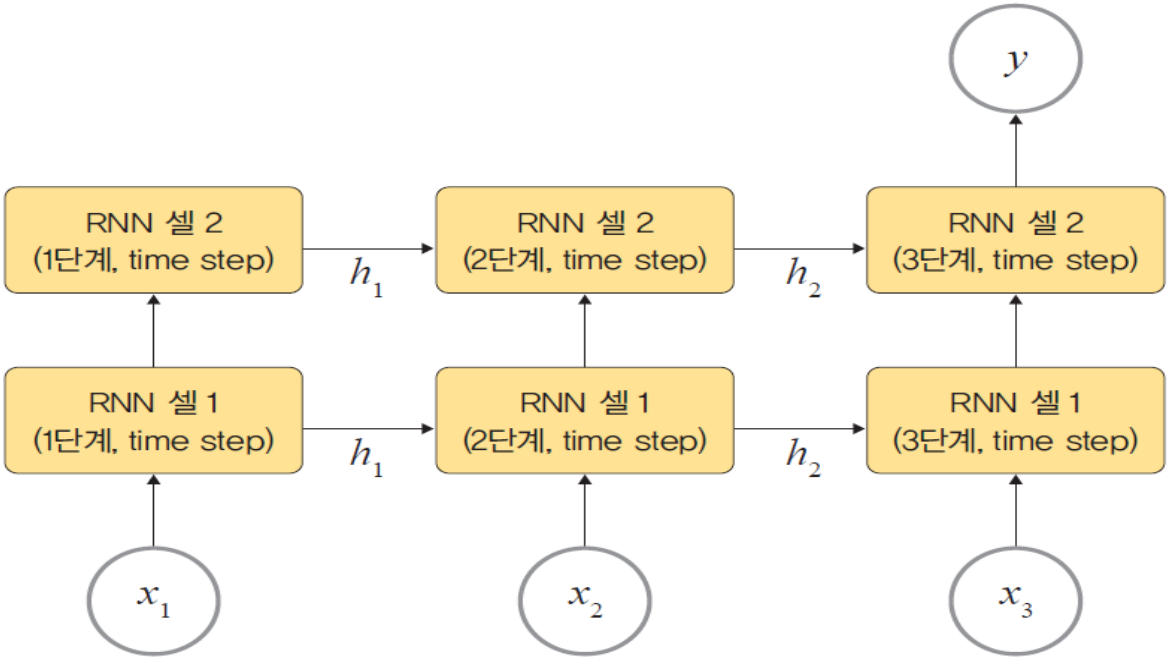
- **순환 신경망(RNN)**
  - ❖ 다대일 구조에 층을 쌓아 올리면 다음과 같이 적층된 구조를 가질 수 있음

```
In_layer = tf.keras.input(shape=(3,1), name='input')
RNN_layer0 = tf.keras.layers.SimpleRNN(100, name='RNN1')(In_layer)
RNN_layer1 = tf.keras.layers.SimpleRNN(100, name='RNN2')(RNN_layer0)
Out_layer = tf.keras.layers.Dense(1, name='output')(RNN_layer1)
```

# 7.3 순환 신경망(RNN)

- **순환 신경망(RNN)**
  - ❖ 코드를 구조화하면 다음 그림과 같음
  - ❖ 코드는 입력과 출력 사이에 두 개의 RNN 셀만 가지고 있는 것에 주의해야 함

▼ 그림 7-6 적층된 다대일 모델



# 7.3 순환 신경망(RNN)

- **순환 신경망(RNN)**

  4. **다대다:** 입력과 출력이 다수인 구조로, 언어를 번역하는 자동 번역기 등이 대표적인 사례

     예를 들어 다대다에 대한 모델은 텐서플로 2에서 다음과 같이 구현함

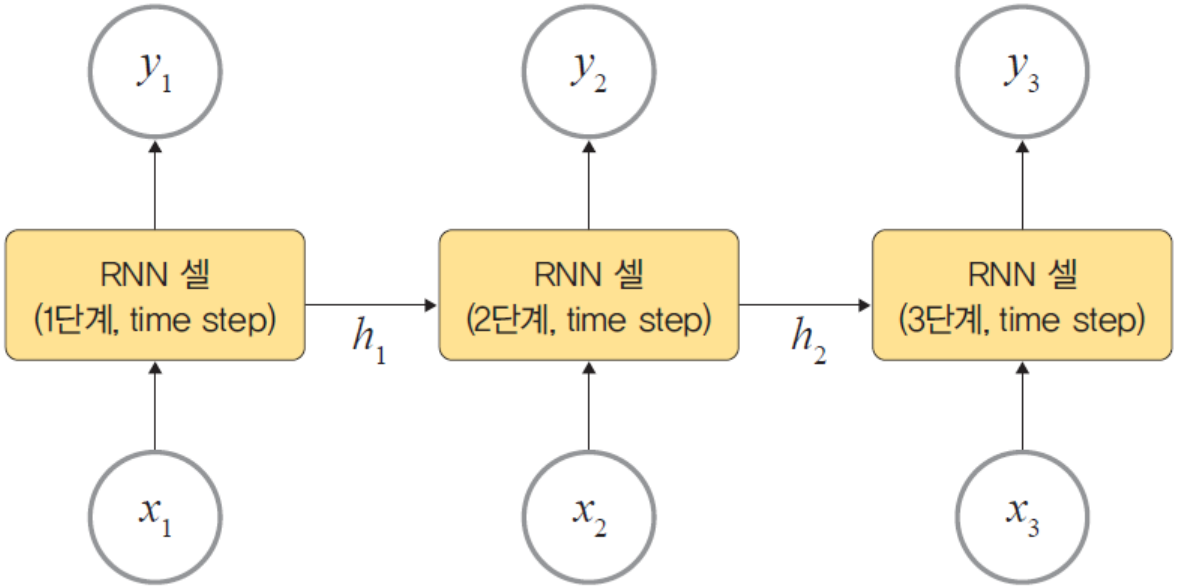
```
In_layer = tf.keras.input(shape=(3,1), name='input')
RNN_layer = tf.keras.layers.SimpleRNN(100, return_sequences=True, name='RNN')(In_layer)
Out_layer = tf.keras.layers.TimeDistributed(keras.layers.Dense(1), name='output')(RNN_
layer)
```

# 7.3 순환 신경망(RNN)

- **순환 신경망(RNN)**
  - ❖ 다음 그림은 다대다에 대한 모델을 표현한 것

▼ 그림 7-7 다대다 모델

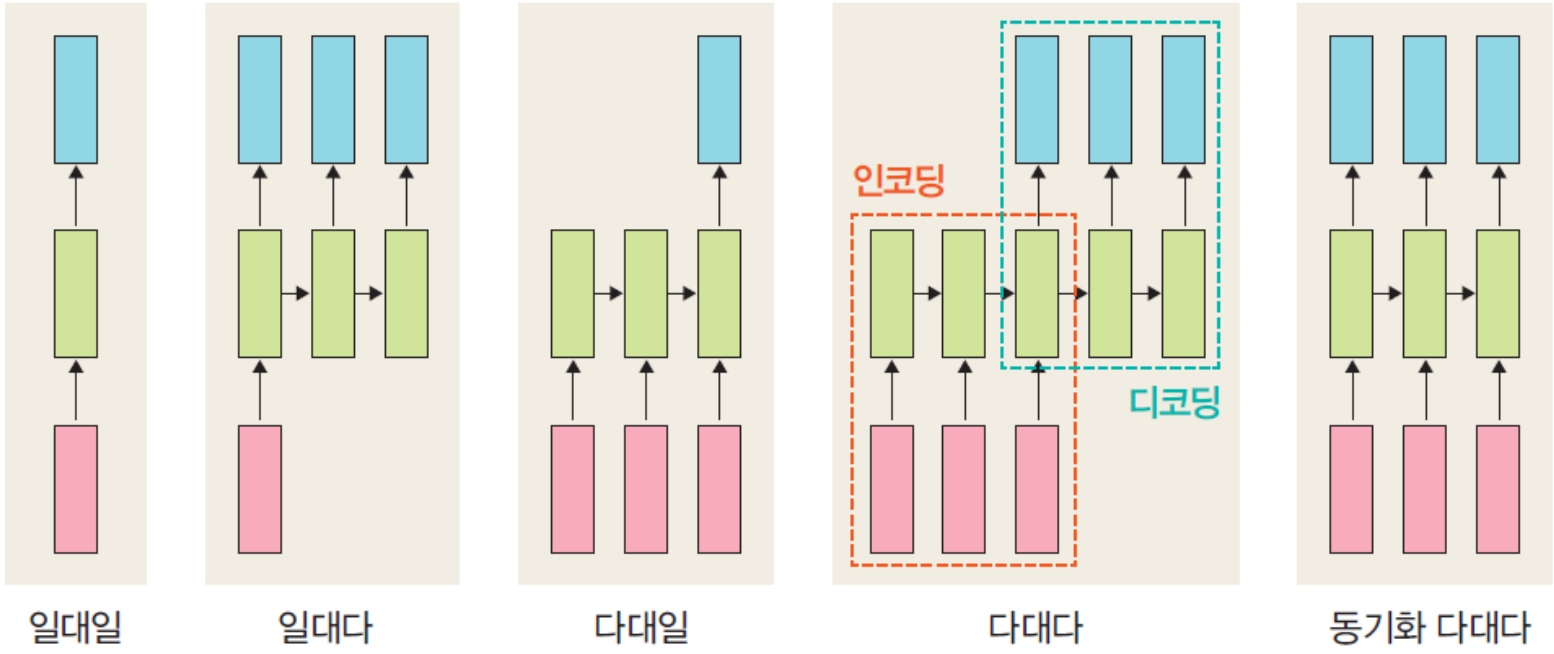
**5. 동기화 다대다:** 4의 유형처럼 입력과 출력이 다수인 구조
문장에서 다음에 나올 단어를 예측하는 언어 모델, 즉 프레임 수준의 비디오 분류가 대표적 사례



# 7.3 순환 신경망(RNN)

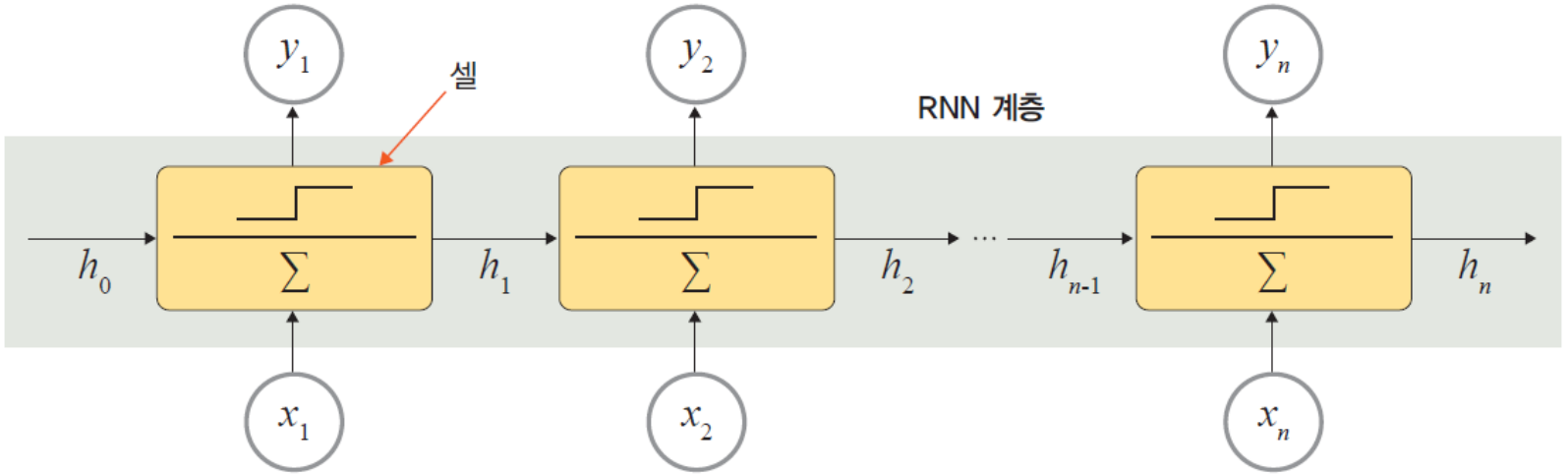
- **순환 신경망(RNN)**
  - ❖ 다음 그림은 앞서 언급된 순환 신경망 구조들을 그림으로 표현한 것

▼ 그림 7-8 RNN 모델 유형



# 7.3 순환 신경망(RNN)

- **RNN 계층과 셀**
  - RNN은 내장된(built-in) 계층뿐만 아니라 셀 레벨의 API도 제공
  - RNN 계층이 입력된 배치 순서열을 모두 처리하는 것과 다르게 RNN 셀은 오직 하나의 단계(time step)만 처리함
  - RNN 셀은 RNN 계층의 for loop 구문의 내부라고 할 수 있음



# 7.3 순환 신경망(RNN)

- **RNN 계층과 셀**
  - RNN 계층은 셀을 래핑하여 동일한 셀을 여러 단계에 적용
  - 그림 7-9에서도 동일한 셀이 $x_1$, $x_2$, …, $x_n$ 등 전체 RNN 네트워크(계층)에서 사용되고 있음
  - 셀은 입력 시퀀스에서 반복되고 return_sequences 같은 옵션을 기반으로 출력 값을 계산
  - 즉, 셀은 실제 계산에 사용되는 RNN 계층의 구성 요소로, 단일 입력과 과거 상태(state)를 가져와서 출력과 새로운 상태를 생성

# 7.3 순환 신경망(RNN)

- **RNN 계층과 셀**
  - 참고로 셀 유형은 다음과 같음
    - **tf.keras.layers.SimpleRNNCell:** SimpleRNN 계층에 대응되는 RNN 셀
    - **tf.keras.layers.GRUCell:** GRU 계층에 대응되는 GRU 셀
    - **tf.keras.layers.LSTMCell:** LSTM 계층에 대응되는 LSTM 셀
  - 이렇게 RNN의 계층과 셀을 분리해서 설명하는 이유는 텐서플로 2에서 이 둘을 분리해서 구현이 가능하기 때문임
  - 앞으로 진행될 RNN 예제는 이 둘을 분리해서 진행

# 7.3 순환 신경망(RNN)

- **RNN 계층과 셀**
  - RNN의 활용 분야로는 대표적으로 '자연어 처리'를 꼽을 수 있음
  - 연속적인 단어들의 나열인 언어(자연어) 처리는 음성 인식, 단어 의미 판단 및 대화 등에 대한 처리가 가능함
  - 이외에도 손글씨, 센서 데이터 등 시계열 데이터 처리에 활용

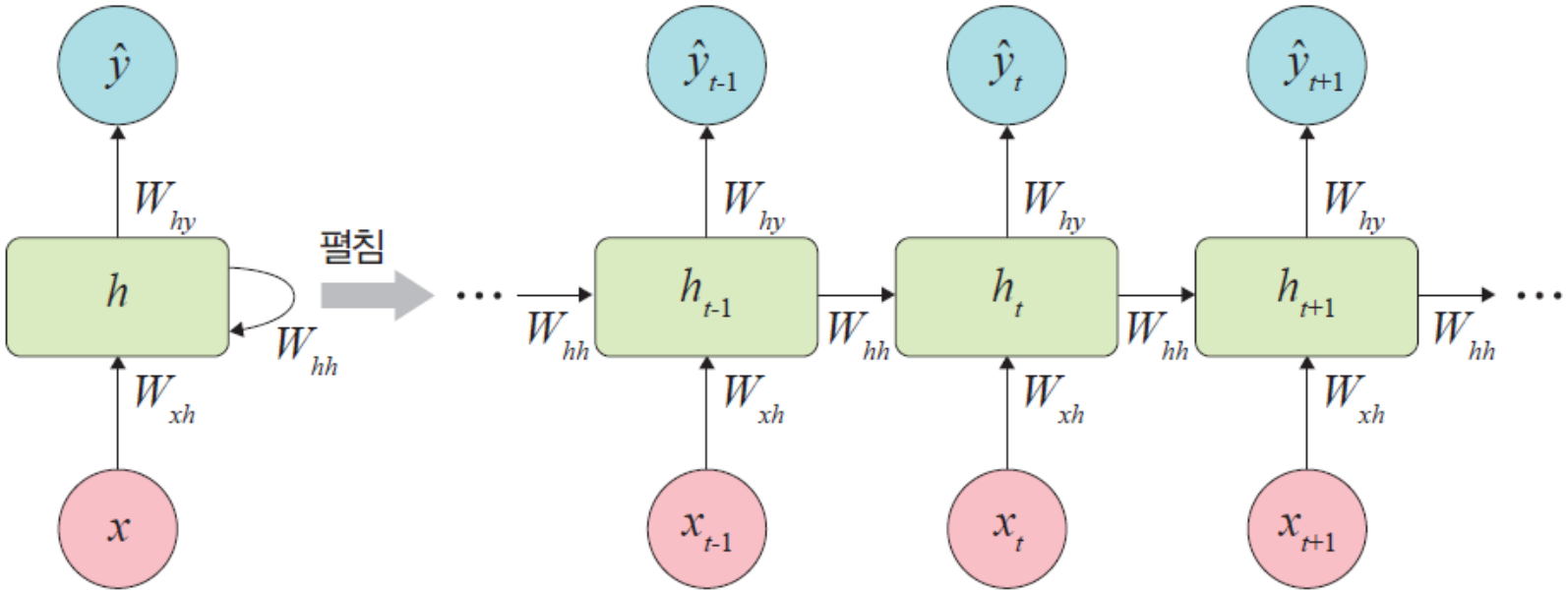
# RNN 구조

# 7.4 RNN 구조

- **RNN 구조**
  - RNN은 은닉층 노드들이 연결되어 이전 단계 정보를 은닉층 노드에 저장할 수 있도록 구성한 신경망
  - 다음 그림에서 볼 수 있듯이 $x_{t-1}$에서 $h_{t-1}$을 얻고 다음 단계에서 $h_{t-1}$과 $x_t$를 사용하여 과거 정보와 현재 정보를 모두 반영함
  - 또한, $h_t$와 $x_{t+1}$의 정보를 이용하여 과거와 현재 정보를 반복해서 반영하는데, 이러한 구조를 요약한 것이 다음 그림의 오른쪽 부분과 같음

# 7.4 RNN 구조

▼ 그림 7-10 RNN 구조



# 7.4 RNN 구조

- **RNN 구조**
  - RNN에서는 입력층, 은닉층, 출력층 외에 가중치를 세 개 가짐
  - RNN의 가중치는 $W_{xh}$, $W_{hh}$, $W_{hy}$로 분류
  - $W_{xh}$는 입력층에서 은닉층으로 전달되는 가중치이고, $W_{hh}$는 t 시점의 은닉층에서 t+1 시점의 은닉층으로 전달되는 가중치
  - 또한, $W_{hy}$는 은닉층에서 출력층으로 전달되는 가중치
  - 가중치 $W_{xh}$, $W_{hh}$, $W_{hy}$는 모든 시점에 동일하다는 것에 주의할 필요가 있음
  - 즉, 가중치를 공유하는데 그림 7-10과 같이 모든 가중치가 동일한 것을 확인할 수 있음

# 7.4 RNN 구조

- **RNN 구조**
  - ❖ 이제 t 단계에서의 RNN 계산에 대해 알아보겠음

**1. 은닉층** 계산을 위해 $x_t$와 $h_{t-1}$이 필요함

즉, (이전 은닉층×은닉층 - 은닉층 가중치 + 입력층 - 은닉층 가중치×(현재) 입력값)으로 계산할 수 있으며, RNN에서 은닉층은 일반적으로 하이퍼볼릭 탄젠트 활성화 함수를 사용

이를 수식으로 나타내면 다음과 같음

$$h_t = \tanh(\hat{y}_t)$$

$$\hat{y}_t = W_{hh} \times h_{t-1} + W_{xh} \times x_t$$

# 7.4 RNN 구조

- **RNN 구조**

  **2. 출력층**은 심층 신경망과 계산 방법이 동일함

  즉, (은닉층 - 출력층 가중치×현재 은닉층)에 소프트맥스 함수를 적용

  이를 수식으로 나타내면 다음과 같음

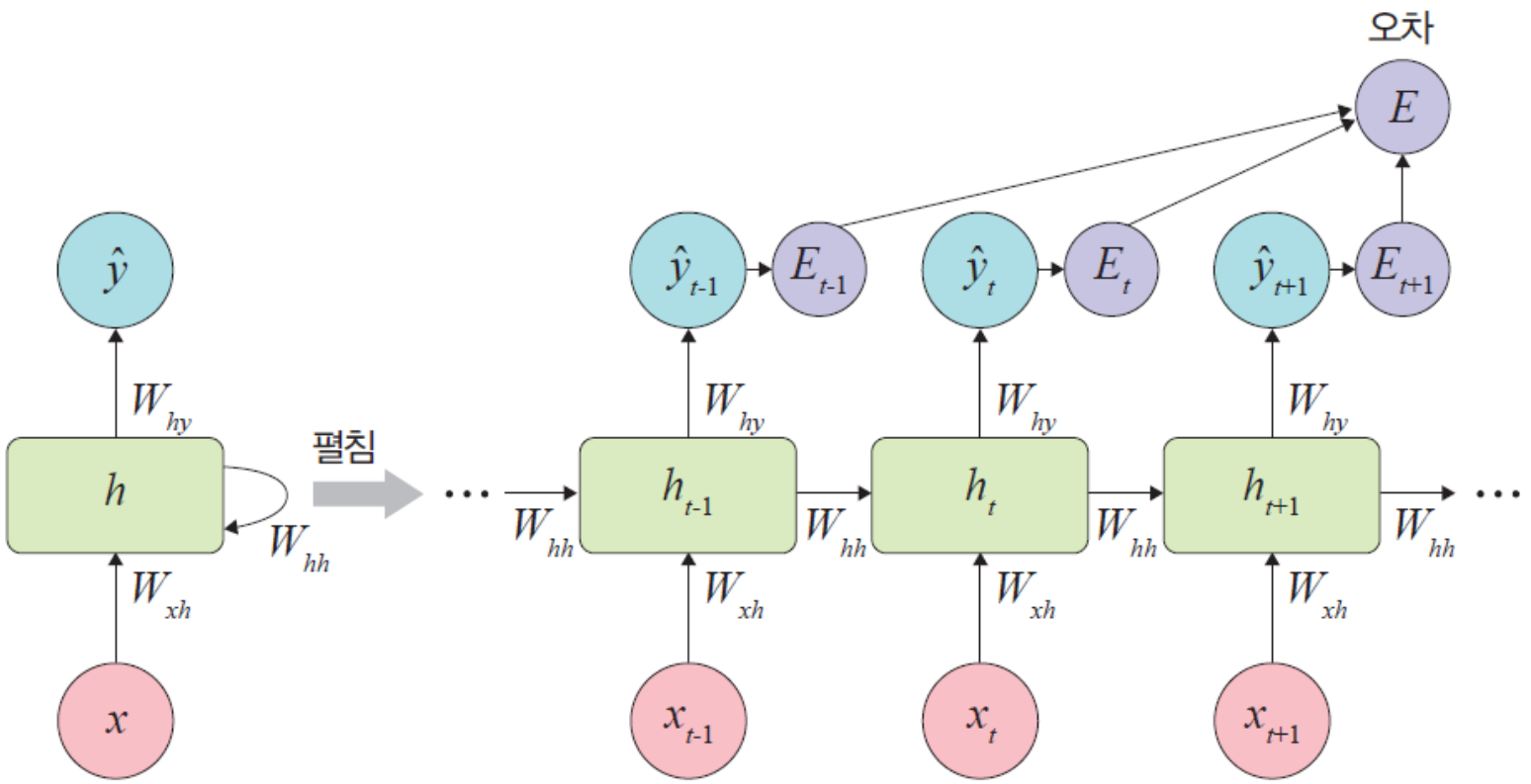
$$\hat{y}_t = \text{softmax}(W_{hy} \times h_t)$$

  **3.** RNN의 **오차**(E)는 심층 신경망에서 전방향(feedforward) 학습과 달리 각 단계(t)마다 오차를 측정

  즉, 각 단계마다 실제 값($y_t$)과 예측 값 $(\text{hat}\,\hat{y}_t)$ 으로 오차(평균 제곱 오차(mean square error) 적용)를 이용하여 측정

# 7.4 RNN 구조

▼ 그림 7-11 RNN의 순방향 학습



# 7.4 RNN 구조

- **RNN 구조**

**4.** RNN에서 **역전파**는 BPTT(BackPropagation Through Time)를 이용하여 모든 단계마다 처음부터 끝까지 역전파

오차는 각 단계(t)마다 오차를 측정하고 이전 단계로 전달되는데, 이것을 BPTT라고 함

즉, **3**에서 구한 오차를 이용하여 $W_{xh}$, $W_{hh}$, $W_{hy}$ 및 바이어스(bias)를 업데이트

이때 BPTT는 오차가 멀리 전파될 때(왼쪽으로 전파) 계산량이 많아지고 전파되는 양이 점차 적어지는 문제점(기울기 소멸 문제(vanishing gradient problem))이 발생

기울기 소멸 문제를 보완하기 위해 오차를 몇 단계까지만 전파시키는 생략된-BPTT(truncated BPTT)를 사용할 수도 있고, 근본적으로는 LSTM 및 GRU를 많이 사용

# 7.4 RNN 구조

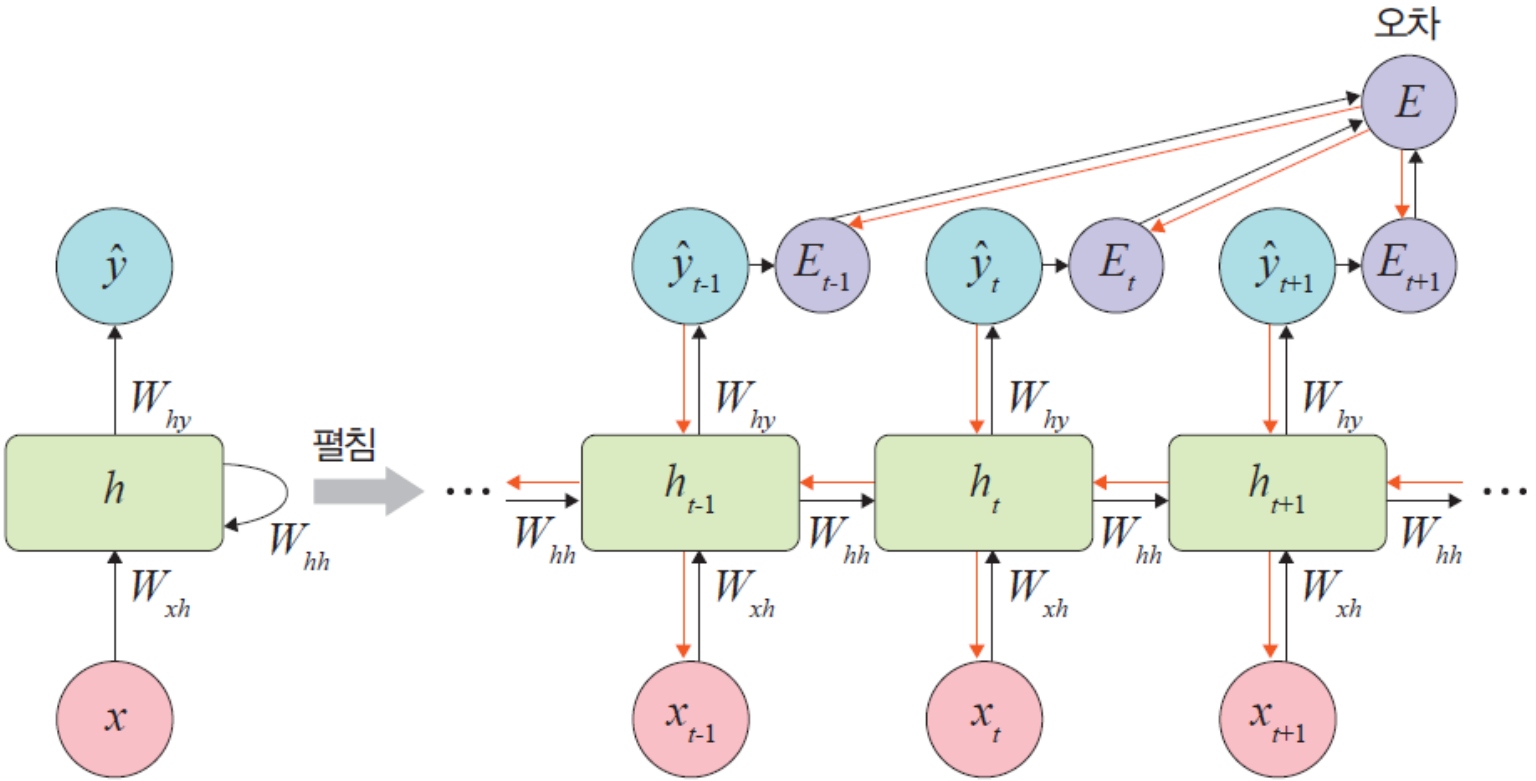
- **RNN 구조**

  **생략된-BPTT**

  - 계산량을 줄이기 위해 현재 단계에서 일정 시점까지만(보통 5단계 이전까지만) 오류를 역전파하는데, 이것을 생략된-BPTT라고 함

# 7.4 RNN 구조

▼ 그림 7-12 RNN의 역방향 학습



# 7.4 RNN 구조

- **RNN 구조**
  - 이제 IMDB 데이터셋을 사용하여 텐서플로 2에서 RNN 계층과 셀을 구현해 보겠음

**IMDB 데이터셋**

- IMDB 데이터셋은 영화 리뷰에 대한 데이터 5만 개로 구성되어 있음
- 이것을 훈련 데이터 2만 5000개와 테스트 데이터 2만 5000개로 나누며, 각각 50%씩 긍정 리뷰와 부정 리뷰가 있음
- 이 데이터는 이미 전처리가 되어 있어 각 리뷰가 숫자로 변환되어 있음
- 스탠포드 대학교에서 2011년에 낸 논문에서 이 데이터를 소개
- 당시 논문에서는 IMDB 데이터셋을 훈련 데이터와 테스트 데이터 50:50 비율로 분할하여 88.89%의 정확도를 얻었다고 소개
- IMDB 영화 리뷰 데이터셋은 imdb.load_data() 메서드로 바로 내려받아 사용할 수 있도록 지원하고 있음
- 데이터셋에 대한 더 자세한 내용은 https://www.imdb.com/interfaces/를 확인

# 7.4 RNN 구조

## RNN 셀 구현

### ❖ 먼저 필요한 라이브러리들을 호출

**코드 7-3** 라이브러리 호출

```
import os ------ 운영 체제의 모듈을 가져옵니다.
os.environ['TF_CPP_MIN_LOG_LEVEL'] = "2" ------ 케라스에서 발생하는 경고(warning) 메시지를 제거합니다.

import tensorflow as tf
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
```

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ❖ 필요한 값들을 초기화

코드 7-4 값 초기화

```
tf.random.set_seed(22)
np.random.seed(22)
assert tf.__version__.startswith('2.') ------ 텐서플로 버전이 2임을 확인

batch_size = 128
total_words = 10000
max_review_len = 80
embedding_len = 100
```

# 7.4 RNN 구조

## ● RNN 셀 구현

### ❖ 모형을 적용하기 위한 데이터셋을 준비

코드 7-5 데이터셋 준비

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.imdb.load_data(num_
                                       words=total_words) ------①

x_train = tf.keras.preprocessing.sequence.pad_sequences(x_train, maxlen=max_review_len) ----②
x_test = tf.keras.preprocessing.sequence.pad_sequences(x_test, maxlen=max_review_len)

train_data = tf.data.Dataset.from_tensor_slices((x_train, y_train)) ------③
train_data = train_data.shuffle(10000).batch(batch_size, drop_remainder=True) ------④
                x_test, y_test 데이터에 대한 넘파일 배열(numpy array)을 바로 Dataset으로 변환
test_data = tf.data.Dataset.from_tensor_slices((x_test, y_test)) ------
test_data = test_data.batch(batch_size, drop_remainder=True) ------ 테스트 데이터셋을 변환
print('x_train_shape:', x_train.shape, tf.reduce_max(y_train), tf.reduce_min(y_train))
print('x_test_shape:', x_test.shape)

sample = next(iter(test_data))
print(sample[0].shape)
```

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ① imdb.load_data() 함수를 사용하여 IMDB 데이터셋을 내려받음
  - imdb.load_data() 파라미터로 num_words를 사용하는데, num_words는 데이터에서 등장 빈도 순위로 몇 번째에 해당하는 단어까지 사용할지를 의미
  - 앞의 코드에서 10000을 사용하고 있는데, 등장 빈도 순위가 1~10000에 해당하는 단어만 사용하겠다는 의미

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ❖ ② 전체 훈련 데이터셋에서 각 샘플의 길이는 서로 다를 수 있음
  - ❖ 또한, 각 문서 혹은 각 문장은 단어 수가 제각각임
  - ❖ 모델의 입력으로 사용하려면 모든 샘플 길이를 동일하게 맞추어야 함
  - ❖ 이를 자연어 처리에서는 패딩(padding) 작업이라고 하며, 보통 숫자 0을 넣어서 길이가 다른 샘플들의 길이를 맞추어 줌
  - ❖ 케라스에서는 pad_sequence()를 사용
  - ❖ pad_sequence()는 정해 준 길이보다 길이가 긴 샘플은 값을 일부 자르고, 정해 준 길이보다 길이가 짧은 샘플은 값을 0으로 채움
    - 첫 번째 인자: 패딩을 진행할 데이터
    - maxlen: 모든 데이터에 대해 정규화할 길이

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ③ 넘파이 배열(NumPy array)을 Dataset으로 변환
  - 이때 주의할 것은 변환하려는 전체 데이터를 메모리로 로딩해야 하므로 큰 용량의 메모리가 필요함
  - 메모리 문제에 대한 해결책은 Dataset의 from_generator를 사용하는 것
  - from_generator를 사용하면 데이터를 한 번에 메모리에 로딩하는 것이 아니고, 필요할 때만 파이썬 generator를 통해 가져옴

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ❖ ④ ③에서 만들어 준 데이터셋을 변형해 줌

```
train_data = train_data.shuffle(10000).batch(batch_size,
                                ⓐ              ⓑ
drop_remainder=True)
ⓒ
```

ⓐ shuffle(): 데이터셋을 임의로 섞어 줌
여기에서 사용되는 것이 buffer_size
데이터를 메모리로 불러와서 섞는 과정이 진행되므로 buffer_size를 지정함
버퍼에서 임의로 샘플을 뽑고, 뽑은 샘플은 다른 샘플로 대체함
데이터셋이 완벽하게 섞이기 위해 전체 데이터셋의 크기에 비해 크거나
같은 버퍼 크기로 지정해야 함

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ⓑ batch(): 데이터셋의 항목들을 하나의 배치로 묶어 줌
    batch_size는 몇 개의 샘플로 가중치를 갱신할지 지정함
  - ⓒ drop_remainder: 마지막 배치 크기를 무시하고 지정한 배치 크기를 사용할 수 있음

# 7.4 RNN 구조

- **RNN 셀 구현**
  - 다음은 훈련과 검증 용도의 데이터셋에 대한 형태를 출력한 결과

```
x_train_shape: (25000, 80) tf.Tensor(1, shape=(), dtype=int64) tf.Tensor(0, shape=(),
dtype=int64)
x_test_shape: (25000, 80)
(128, 80)
```

# 7.4 RNN 구조

## ● RNN 셀 구현

❖ 이제 RNN 셀을 이용한 네트워크(혹은 신경망)를 생성

**코드 7-6** RNN 셀을 이용한 네트워크 생성

```
class RNN_Build(tf.keras.Model): ------①
    def __init__(self, units): ------②
        super(RNN_Build, self).__init__() ------③

        self.state0 = [tf.zeros([batch_size, units])] ------④
        self.state1 = [tf.zeros([batch_size, units])]
        self.embedding = tf.keras.layers.Embedding(total_words, embedding_len,
                                                   input_length=max_review_len) ------⑤

        self.RNNCell0 = tf.keras.layers.SimpleRNNCell(units, dropout=0.2) ------⑥
        self.RNNCell1 = tf.keras.layers.SimpleRNNCell(units, dropout=0.2)
        self.outlayer = tf.keras.layers.Dense(1)
```

# 7.4 RNN 구조

- **RNN 셀 구현**

```
def call(self, inputs, training=None): ------ ②'
    x = inputs
    x = self.embedding(x) ------ 입력 데이터에 원-핫 인코딩 적용
    state0 = self.state0
    state1 = self.state1
    for word in tf.unstack(x, axis=1): ------ ⑦
        out0, state0 = self.RNNCell0(word, state0, training)
        out1, state1 = self.RNNCell1(out0, state1, training)
    x = self.outlayer(out1) ------ 출력층 out1을 적용한 후 그 값을 x 변수에 저장
    prob = tf.sigmoid(x) ------ 마지막으로 x에 시그모이드 활성화 함수를 적용하여 prob에 저장
    return prob ------ prob 값을 반환
```

out0, state0 각각에 self.RNNCell0에서 받아 온 값을 저장

out1, state1 각각에 self.RNNCell1에서 받아 온 값을 저장

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ① 객체 지향 프로그램을 파이썬에서 구현한 것

    즉, 구조를 설계한 후 재사용성을 고려하거나 코드의 반복을 최소화하는 데 사용
  - ②, ②′ __init__은 클래스 인스턴스를 생성할 때 초기화하는 부분

    이때 init은 객체가 생성될 때 호출되며, call은 인스턴스가 생성될 때 호출
  - ③ 기반 클래스의 __init__ 메서드를 호출해 줌

    super() 뒤에 .(점)을 붙여서 메서드를 호출하는 방식
  - ④ self는 자신의 인스턴스를 의미하는 것으로, tf.zeros를 사용하여 0 값으로 채워진 텐서를 생성해서 state0에 저장

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ⑤ 케라스는 텍스트 데이터에 대해 워드 임베딩을 수행하는 임베딩층(embedding layer)을 제공

    임베딩층을 사용하려면 각 입력이 모두 정수로 인코딩되어 있어야 함

    즉, 각각의 입력은 정수로 변환된 상태에서 임베딩층을 구성

```
self.embedding = tf.keras.layers.Embedding(total_words, embedding_len,
                                           input_length=max_review_len)
```

ⓐ ⓑ ⓒ

ⓐ 첫 번째 인자: 텍스트 데이터의 전체 단어 집합 크기

예를 들어 데이터셋의 단어들이 0부터 20000까지 인코딩되었다면 단어 집합 크기는 20001이 되어야 함(이때 인덱스에 주의)

# 7.4 RNN 구조

- **RNN 셀 구현**

  ⓑ 두 번째 인자: 임베딩이 되고 난 후 단어의 차원

  이 값을 256으로 준다면 모든 단어의 차원이 256이 됨

  ⓒ input_length: 입력 데이터의 길이

  예를 들어 각 데이터 길이가 단어 500개로 구성되어 있다면 이 값은 500이 됨

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ❖ ⑥ SimpleRNN의 셀 클래스를 의미

    - units: 출력 공간의 차원

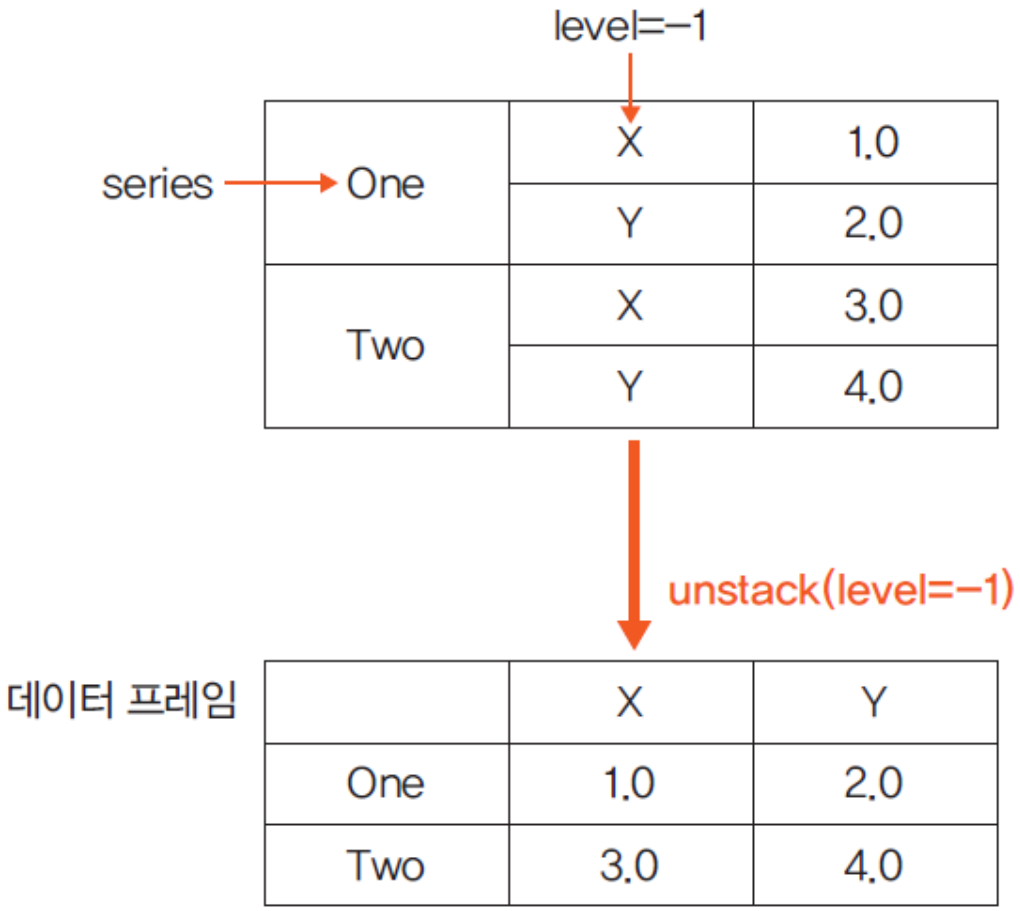
    - dropout: 0과 1 사이의 부동소수점. 입력 중에서 삭제할(고려하지 않을) 유닛의 비율

  - ❖ ⑦ unstack(): 중복된 값(예 날짜, 숫자)이 있을 때 사용하면 유용함

    즉, 다음 그림과 같이 그룹으로 묶은 데이터를 행렬 형태로 전환하여 연산할 때 사용

# 7.4 RNN 구조

▼ 그림 7-13 unstack 예시



# 7.4 RNN 구조

- **RNN 셀 구현**
  - ❖ 생성된 네트워크를 활용하여 모형을 훈련시킴

코드 7-7 모델 훈련

```
import time
units = 64
epochs = 4
t0 = time.time() ------ 모형의 실행 시간을 위해 시간을 t0에 저장

model = RNN_Build(units)
model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss=tf.losses.BinaryCrossentropy(),
              metrics=['accuracy'],
              experimental_run_tf_function=False) ------①

model.fit(train_data, epochs=epochs, validation_data=test_data, validation_freq=2) -----②
```

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ① model.compile()에서는 다양한 하이퍼파라미터를 정의

```
model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
                                    ⓐ
              loss=tf.losses.BinaryCrossentropy(), metrics=['accuracy'],
                                    ⓑ                      ⓒ
              experimental_run_tf_function=False)
                                    ⓓ
```

ⓐ optimizer: 옵티마이저를 설정
여기에서는 0.001의 학습률을 적용한 Adam을 사용

ⓑ loss: 훈련 과정에서 사용할 손실 함수(loss function)를 설정
여기에서는 이진 분류(binary classification)로 BinaryCrossentropy를 사용

# 7.4 RNN 구조

- **RNN 셀 구현**

  ⓒ metrics: 훈련을 모니터링하기 위한 지표를 선택
  여기에서는 정확도(accuracy)를 사용

  ⓓ experimental_run_tf_function: 모델을 인스턴스화하는 기능을 제공하며,
  실제로experimental_run_tf_function이 호출되어야 컴파일이 실행

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ❖ ② model.fit()은 모델을 학습하는 데 사용

```
model.fit(train_data, epochs=epochs, validation_data=test_data, validation_freq=2)
          ⓐ           ⓑ              ⓒ                  ⓓ
```

ⓐ 첫 번째 인자: 입력 데이터

ⓑ epochs: 학습 데이터 반복 횟수

ⓒ validation_data: 검증 데이터

ⓓ validation_freq: 에포크마다 무조건 검증 데이터셋에 대한 계산을 수행하지 않고 적절한 간격을 두고 계산하는 것

예를 들어 validation_freq=[1, 2, 10]을 설정한다면 첫 번째, 두 번째, 열 번째 에포크 후 검증

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ❖ 다음은 모델 훈련을 출력한 결과

```
Epoch 1/4
195/195 [==============================] - 5s 27ms/step - loss: 0.6249 - accuracy:
0.6134
Epoch 2/4
195/195 [==============================] - 8s 42ms/step - loss: 0.3543 - accuracy:
0.8494 - val_loss: 0.4064 - val_accuracy: 0.8224
Epoch 3/4
195/195 [==============================] - 5s 27ms/step - loss: 0.1926 - accuracy:
0.9270
Epoch 4/4
195/195 [==============================] - 7s 37ms/step - loss: 0.0828 - accuracy:
0.9709 - val_loss: 0.6795 - val_accuracy: 0.7970
<tensorflow.python.keras.callbacks.History at 0x21fad010240>
```

# 7.4 RNN 구조

## RNN 셀 구현

❖ 이제 모델에 대한 평가를 해 보자

코드 7-8 모델 평가

```
print("훈련 데이터셋 평가...")
(loss, accuracy) = model.evaluate(train_data, verbose=0) ------①
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))
print("테스트 데이터셋 평가...")
(loss, accuracy) = model.evaluate(test_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))
t1 = time.time()
print('시간:', t1-t0)
```

# 7.4 RNN 구조

- **RNN 셀 구현**
  - ❖ ① model.evaluate는 모델을 평가하기 위한 함수로 파라미터 의미는 다음과 같음
    - train_data: 훈련 데이터
    - verbose: 얼마나 자세하게 정보를 표시할지 지정

  - ❖ 다음은 모델 평가를 실행한 결과

```
훈련 데이터셋 평가...
loss=0.0260, accuracy: 99.3550%
테스트 데이터셋 평가...
loss=0.6795, accuracy: 79.7035%
시간: 40.00159311294556
```

# 7.4 RNN 구조

- **RNN 셀 구현**
  - 훈련 데이터셋의 정확도가 99%이고, 테스트 데이터셋은 80%로 결과가 나쁘지 않음
  - RNN 계층 모델도 훈련한 후 결과를 비교해 보자
  - RNN 계층을 이용한 네트워크를 만들어 데이터들을 훈련시켜 보겠음
  - 데이터는 RNN 셀에서 사용했던 동일한 데이터이며, 네트워크만 다르게 구성하여 정확도 및 수행 시간만 비교해 보자

# 7.4 RNN 구조

- **RNN 계층 구현**
  - ❖ 필요한 라이브러리 및 데이터 호출은 RNN 셀에서의 수행과 동일하므로 생략
  - ❖ 바로 RNN 계층 네트워크(신경망)부터 생성

**코드 7-9** 네트워크(신경망) 구축

```
class RNN_Build(tf.keras.Model):
    def __init__(self, units):
        super(RNN_Build, self).__init__()
        self.embedding = tf.keras.layers.Embedding(total_words, embedding_len,
                                                   input_length=max_review_len)

        self.rnn = tf.keras.Sequential([
            tf.keras.layers.SimpleRNN(units, dropout=0.5, return_sequences=True), ------ ①
            tf.keras.layers.SimpleRNN(units, dropout=0.5)
        ])
        self.outlayer = tf.keras.layers.Dense(1)
```

# 7.4 RNN 구조

- **RNN 계층 구현**

```
def call(self, inputs, training=None):
    x = inputs
    x = self.embedding(x)
    x = self.rnn(x)
    x = self.outlayer(x)
    prob = tf.sigmoid(x)

    return prob
```

# 7.4 RNN 구조

- **RNN 계층 구현**
  - ❖ ① SimpleRNN 함수를 사용하여 은닉 노드가 다수 개인 RNN 셀을 여러 개 구축할 수 있음(SimpleRNNCell은 셀이 하나였으나, SimpleRNN은 한 번에 셀을 여러 개 구축할 수 있음)

```
tf.keras.layers.SimpleRNN(units, dropout=0.5, return_sequences=True)
                          ⓐ       ⓑ            ⓒ
```

ⓐ units: 네트워크의 층 수(출력 공간의 차원)

ⓑ dropout: 전체 노드 중 20% 값을 0으로 설정하여 사용하지 않겠다는 의미

ⓒ return_sequences: 마지막 출력 또는 전체 순서를 반환하는 것

이때 주의해야 할 점은 return_sequences=True는 출력 순서 중 마지막 값만 출력하는 것이 아니라 전체 순서열을 3차원 텐서 형태로 출력하라는 것

# 7.4 RNN 구조

- **RNN 계층 구현**
  - SimpleRNNCell과 SimpleRNN의 코드 구현은 거의 비슷함
  - 단지 네트워크의 def call 함수에서 SimpleRNNCell은 다음과 같이 for 문을 사용하여 SimpleRNNCell을 반복 수행한다는점이 다름
  - 즉, SimpleRNNCell은 셀 단위로 수행되므로 다수의 셀을 수행하려면 다음 예시 코드의 for 문처럼 반복적인 수행이 필요함

```
#SimpleRNNCell
for word in tf.unstack(x, axis=1):
    out0, state0 = self.RNNCell0(word, state0, training)
    out1, state1 = self.RNNCell1(out0, state1, training)

#SimpleRNN
x = self.rnn(x)
```

# 7.4 RNN 구조

- **RNN 계층 구현**
  - 생성된 네트워크를 활용하여 모델을 훈련시킴(RNNCell과 동일한 코드이지만 결과를 확인하고자 또 한 번 실행)

코드 7-10 모델 훈련

```
import time
units = 64
epochs = 4
t0 = time.time()

model = RNN_Build(units)

model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss=tf.losses.BinaryCrossentropy(),
              metrics=['accuracy'],
              experimental_run_tf_function=False)

model.fit(train_data, epochs=epochs, validation_data=test_data, validation_freq=2)
```

# 7.4 RNN 구조

- **RNN 계층 구현**
  - 다음은 모델 훈련을 실행시킨 결과

```
Epoch 1/4
195/195 [==============================] - 12s 61ms/step - loss: 0.5376 - accuracy:
0.7124
Epoch 2/4
195/195 [==============================] - 15s 79ms/step - loss: 0.3508 - accuracy:
0.8511 - val_loss: 0.4648 - val_accuracy: 0.8200
Epoch 3/4
195/195 [==============================] - 12s 63ms/step - loss: 0.2842 - accuracy:
0.8864
Epoch 4/4
195/195 [==============================] - 16s 81ms/step - loss: 0.2370 - accuracy:
0.9079 - val_loss: 0.4664 - val_accuracy: 0.8226
<tensorflow.python.keras.callbacks.History at 0x21fb440f780>
```

# 7.4 RNN 구조

- **RNN 계층 구현**
  - ❖ 마지막으로 모델에 대한 평가를 확인

코드 7-11 모델 평가

```
print("훈련 데이터셋 평가...")
(loss, accuracy) = model.evaluate(train_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))
print("테스트 데이터셋 평가...")
(loss, accuracy) = model.evaluate(test_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))

t1 = time.time()
print('시간:', t1-t0)
```

# 7.4 RNN 구조

- **RNN 계층 구현**
  - 다음은 모델 평가를 실행한 결과

```
훈련 데이터셋 평가...
loss=0.1137, accuracy: 96.3542%
테스트 데이터셋 평가...
loss=0.4664, accuracy: 82.2636%
시간: 69.2024393081665
```

  - SimpleRNNCell보다 훈련 데이터에 대한 정확도가 낮아졌고, 테스트 데이터셋에 대한 정확도는 조금 높아졌으나 수행 시간은 길어졌음

# LSTM

# 7.5 LSTM

- **LSTM**
  - RNN은 결정적 단점이 있음
  - 앞서 언급했듯이 가중치가 업데이트되는 과정에서 1보다 작은 값이 계속 곱해지기 때문에 기울기가 사라지는 기울기 소멸 문제가 발생
  - 이를 해결하기 위해 LSTM이나 GRU 같은 확장된 RNN 방식들을 사용하고 있음

# 7.5 LSTM

- **LSTM 구조**
  - ❖ LSTM 구조는 순전파와 역전파 과정으로 살펴보겠음

**LSTM 순전파**

- ❖ LSTM은 기울기 소멸 문제를 해결하기 위해 망각 게이트, 입력 게이트, 출력 게이트라는 새로운 요소를 은닉층의 각 뉴런에 추가

# 7.5 LSTM

- **LSTM 구조**

  **망각 게이트**

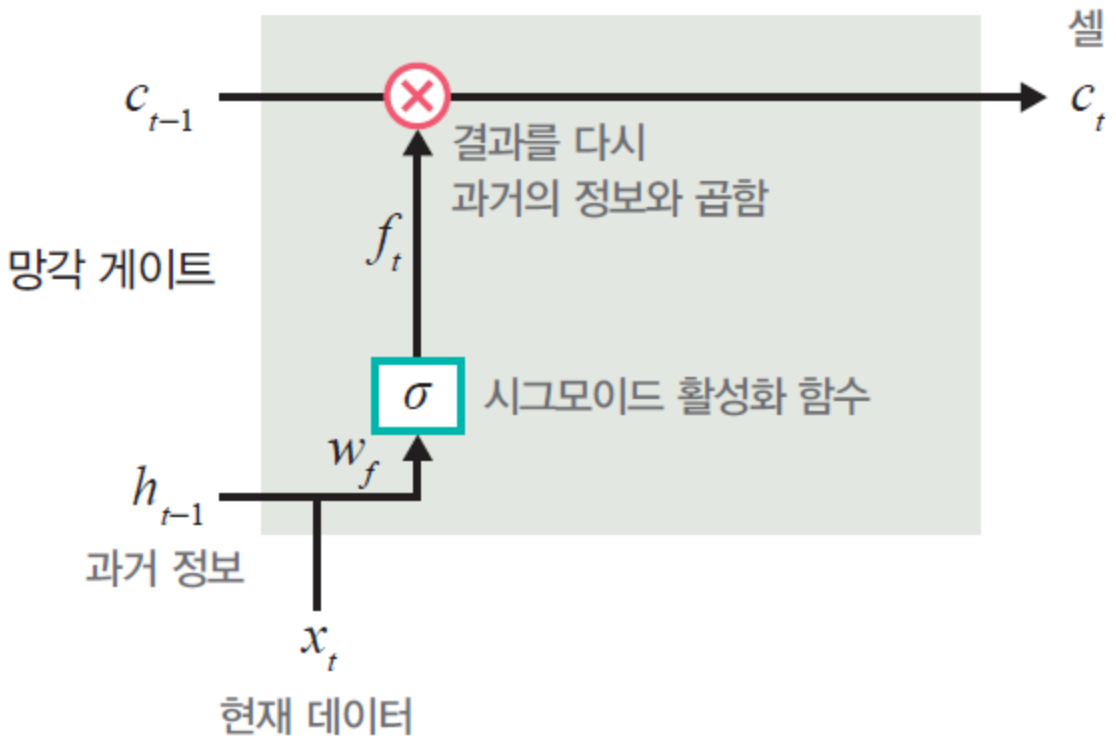
  - 망각 게이트(forget gate)는 과거 정보를 어느 정도 기억할지 결정
  - 과거 정보와 현재 데이터를 입력받아 시그모이드를 취한 후 그 값을 과거 정보에 곱해 줌
  - 시그모이드의 출력이 0이면 과거 정보는 버리고, 1이면 과거 정보는 온전히 보존
  - 0과 1 사이의 출력 값을 가지는 $h_{t-1}$과 $x_t$를 입력 값으로 받음
  - 이때 $x_t$는 새로운 입력 값이고 $h_{t-1}$은 이전 은닉층에서 입력되는 값
  - 즉, $h_{t-1}$과 $x_t$를 이용하여 이전 상태 정보를 현재 메모리에 반영할지 결정하는 역할을 함
    - 계산한 값이 1이면 바로 직전의 정보를 메모리에 유지
    - 계산한 값이 0이면 초기화

# 7.5 LSTM

- **LSTM 구조**
  - ❖ 망각 게이트에 대한 수식은 다음과 같음

$$f_t = \sigma(w_f[h_{t-1}, x_t])$$

$$c_t = f_t \cdot c_{t-1}$$



# 7.5 LSTM

- **LSTM 구조**

  **입력 게이트**

  - 입력 게이트(input gate)는 현재 정보를 기억하기 위해 만들어졌음
  - 과거 정보와 현재 데이터를 입력받아 시그모이드와 하이퍼볼릭 탄젠트 함수를 기반으로 현재 정보에 대한 보존량을 결정
  - 즉, 현재 메모리에 새로운 정보를 반영할지 결정하는 역할을 함

    - 계산한 값이 1이면 입력 $x_t$가 들어올 수 있도록 허용(open)
    - 계산한 값이 0이면 차단

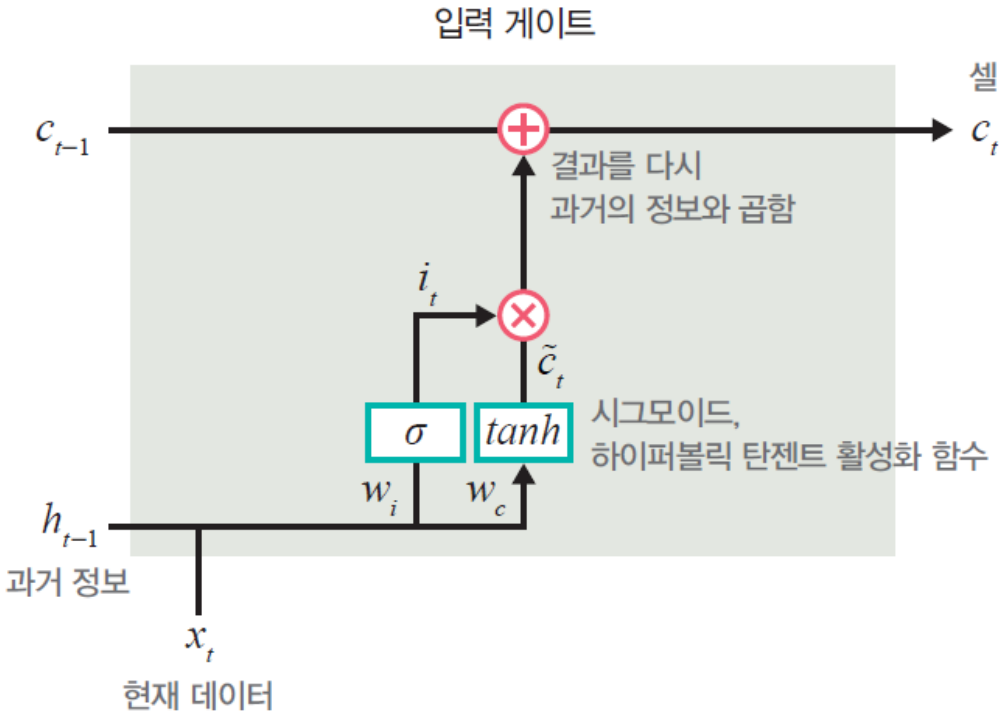
# 7.5 LSTM

- **LSTM 구조**
  - ❖ 이것을 수식으로 정리하면 다음과 같음

$$i_t = \sigma(w_i[h_{t-1}, x_t])$$

$$\tilde{c}_t = \tanh(w_c[h_{t-1}, x_t])$$

$$c_t = c_{t-1} + i_t \cdot \tilde{c}_t$$



# 7.5 LSTM

- **LSTM 구조**

  **셀**

  - 각 단계에 대한 은닉 노드(hidden node)를 메모리 셀이라고 함
  - '총합(sum)'을 사용하여 셀 값을 반영하며, 이것으로 기울기 소멸 문제가 해결
  - 셀을 업데이트하는 방법은 다음과 같음

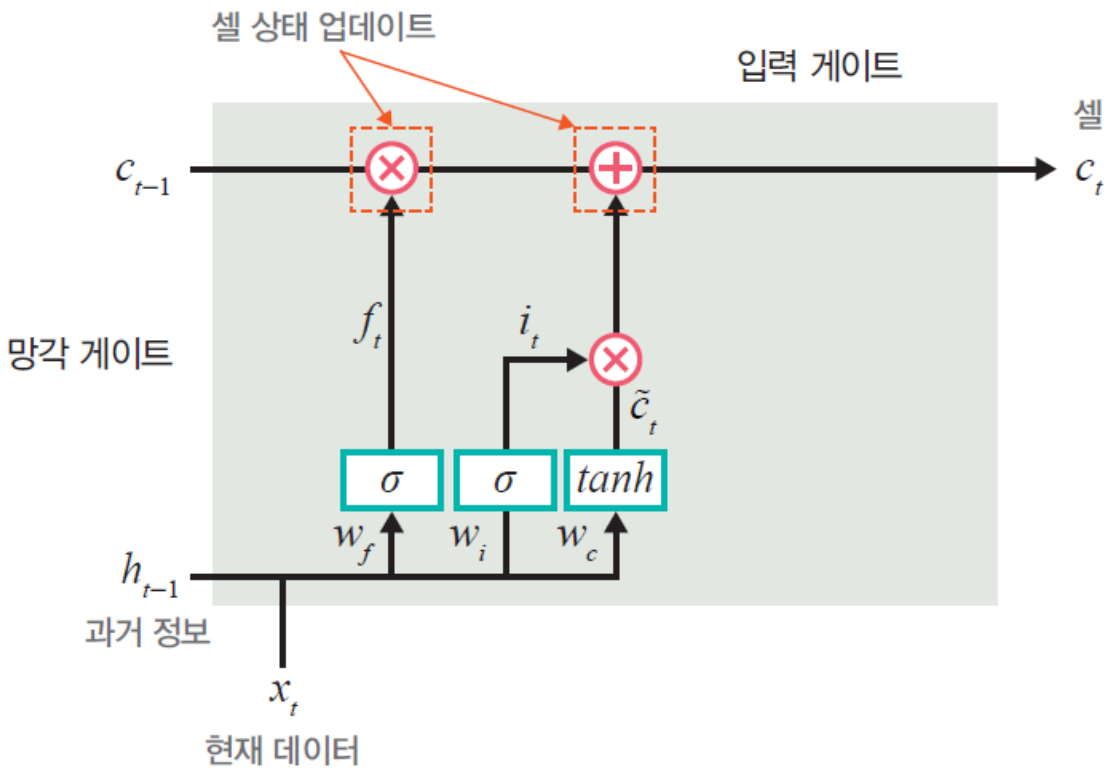
# 7.5 LSTM

- **LSTM 구조**
  - 망각 게이트와 입력 게이트의 이전 단계 셀 정보를 계산하여 현재 단계의 셀 상태(cell state)를 업데이트
  - 다음은 셀에 대한 수식

$$i_t = \sigma(w_i[h_{t-1}, x_t])$$

$$\tilde{c}_t = \tanh(w_c[h_{t-1}, x_t])$$

$$c_t = c_{t-1} + i_t \cdot \tilde{c}_t$$



# 7.5 LSTM

- **LSTM 구조**

  **출력 게이트**

  - 출력 게이트(output gate)는 과거 정보와 현재 데이터를 사용하여 뉴런의 출력을 결정
  - 이전 은닉 상태(hidden state)와 t번째 입력을 고려해서 다음 은닉 상태를 계산
  - LSTM에서는 이 은닉 상태가 그 시점에서의 출력이 됨
  - 출력 게이트는 갱신된 메모리의 출력 값을 제어하는 역할을 함

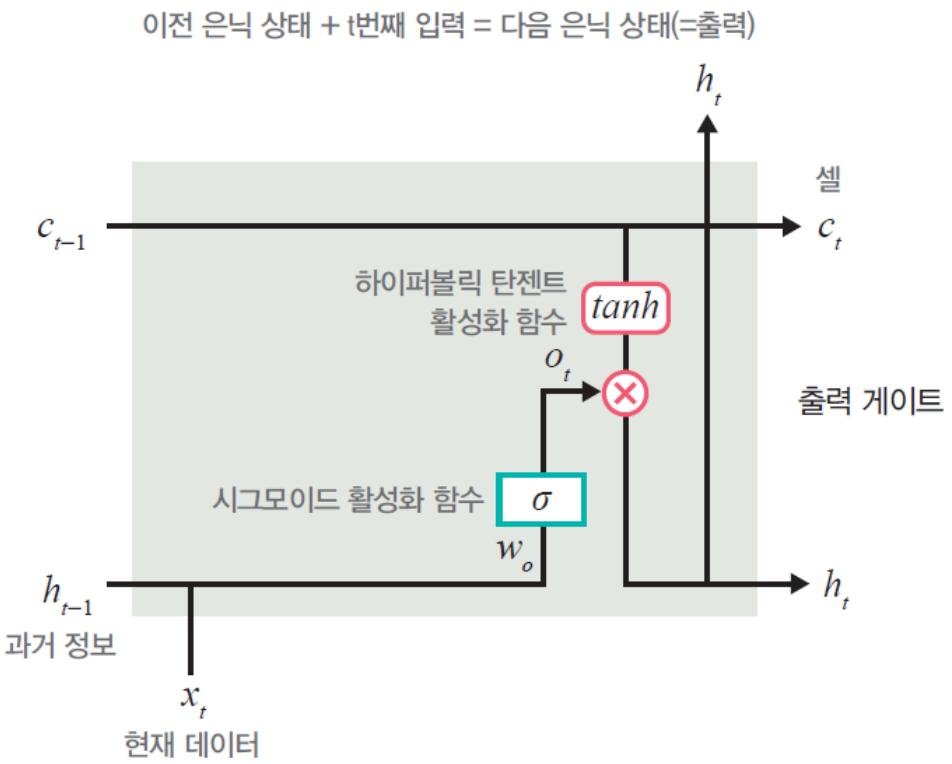
    - 계산한 값이 1이면 의미 있는 결과로 최종 출력
      계산한 값이 0이면 해당 연산 출력을 하지 않음

# 7.5 LSTM

- **LSTM 구조**
  - ❖ 이것을 수식으로 정리하면 다음과 같음

$$o_t = \sigma(w_o [h_{t-1}, x_t])$$

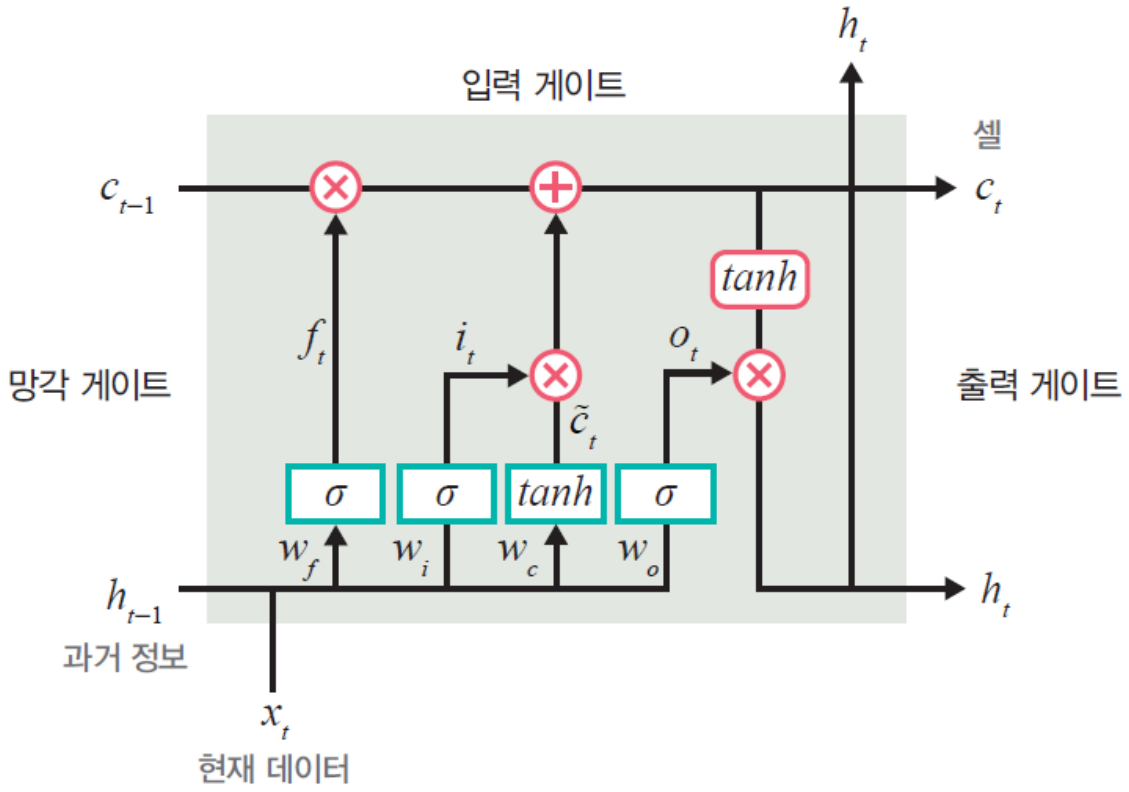
$$h_t = o_t \cdot \tanh(c_{t-1})$$



# 7.5 LSTM

- **LSTM 구조**
  - ❖ 다음 그림은 망각 게이트, 입력 게이트, 출력 게이트를 모두 표현한 것

▼ 그림 7-18 LSTM 전체 게이트



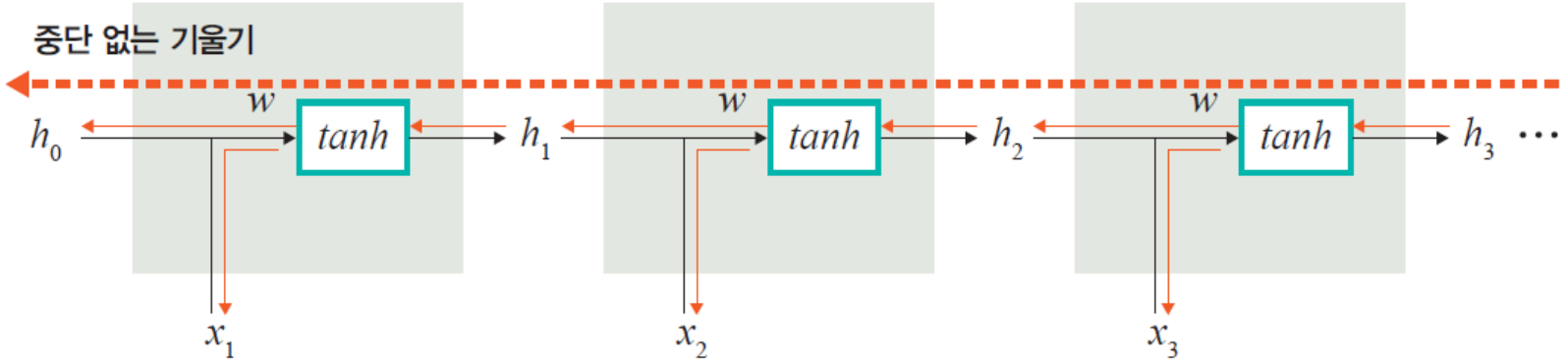
# 7.5 LSTM

- **LSTM 구조**

  **LSTM 역전파**

  - LSTM은 셀을 통해서 역전파를 수행하기 때문에 '중단 없는 기울기(uninterrupted gradient flow)'라고도 함
  - 즉, 다음 그림과 같이 최종 오차는 모든 노드에 전파되는데, 이때 셀을 통해서 중단 없이 전파

▼ 그림 7-19 LSTM 셀 단위 역전파



# 7.5 LSTM

- **LSTM 구조**
  - ❖ 다음은 역전파를 수행하기 위한 공식

$$t_t = \tanh(w_{hh} h_{t-1} + w_{xh} x_t)$$

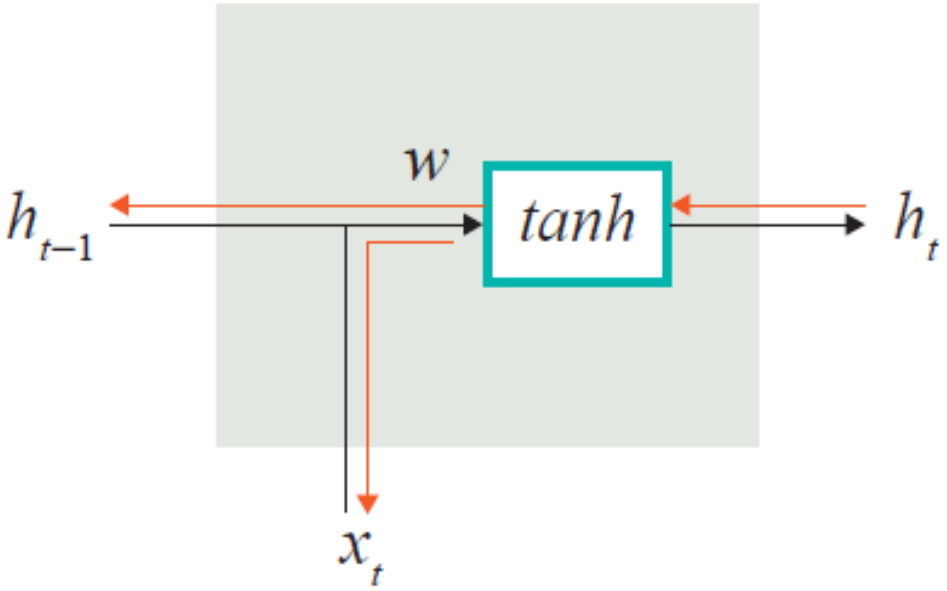
$$= \tanh((w_{hh} \quad w_{xh}) \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix})$$

$$= \tanh(w \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix})$$

# 7.5 LSTM

- **LSTM 구조**
  - 이때 주의해야 할 것은 셀 단위로 오차가 전파된다고 해서 입력 방향으로 오차가 전파되지 않는 것은 아님
  - 다음 그림과 같이 셀 내부적으로는 오차가 입력($x_t$)으로 전파된다는 것도 잊지 말아야 함

▼ 그림 7-20 입력층으로의 역전파



# 7.5 LSTM

## LSTM 셀 구현

- 필요한 라이브러리 및 데이터 호출은 RNN 셀에서의 수행과 동일하므로 생략하고, LSTM 셀을 이용한 네트워크 코드를 살펴보자

코드 7-12 네트워크 생성

```
class LSTM_Build(tf.keras.Model):
    def __init__(self, units):
        super(LSTM_Build, self).__init__()

        self.state0 = [tf.zeros([batch_size, units]), tf.zeros([batch_size, units])]
        self.state1 = [tf.zeros([batch_size, units]), tf.zeros([batch_size, units])]

        self.embedding = tf.keras.layers.Embedding(total_words, embedding_len,
                                                   input_length=max_review_len)
        self.RNNCell0 = tf.keras.layers.LSTMCell(units, dropout=0.5) ------①
        self.RNNCell1 = tf.keras.layers.LSTMCell(units, dropout=0.5)
        self.outlayer = tf.keras.layers.Dense(1)
```

# 7.5 LSTM

- **LSTM 셀 구현**

```
def call(self, inputs, training=None):
    x = inputs
    x = self.embedding(x)
    state0 = self.state0 ------ 초기 상태 0으로 설정
    state1 = self.state1
    for word in tf.unstack(x, axis=1):
        out0, state0 = self.RNNCell0(word, state0, training) ------ train 매개변수 추가
        out1, state1 = self.RNNCell1(out0, state1, training)

    x = self.outlayer(out1)
    prob = tf.sigmoid(x)

    return prob
```

# 7.5 LSTM

- **LSTM 셀 구현**
  - ❖ ① LSTM의 셀 클래스를 의미
    - 첫 번째 인자: 메모리 셀의 개수
    - dropout: 전체 가중치 중 50% 값을 0으로 설정하여 사용하지 않겠다는 의미

# 7.5 LSTM

- **LSTM 셀 구현**
  - 생성된 네트워크를 활용하여 모델을 훈련시킴(RNNCell과 동일한 코드이지만 결과를 확인하려고 또 한 번 실행)

코드 7-13 모델 훈련

```
import time
units = 64
epochs = 4
t0 = time.time()

model = LSTM_Build(units)

model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss=tf.losses.BinaryCrossentropy(),
              metrics=['accuracy'],
              experimental_run_tf_function=False)

model.fit(train_data, epochs=epochs, validation_data=test_data, validation_freq=2)
```

# 7.5 LSTM

- **LSTM 셀 구현**
  - 다음은 모델을 훈련시킨 결과

```
Epoch 1/4
195/195 [==============================] - 18s 91ms/step - loss: 0.4792 - accuracy:
0.7595
Epoch 2/4
195/195 [==============================] - 28s 145ms/step - loss: 0.3136 - accuracy:
0.8711 - val_loss: 0.3617 - val_accuracy: 0.8397
Epoch 3/4
195/195 [==============================] - 20s 103ms/step - loss: 0.2613 - accuracy:
0.8954
Epoch 4/4
195/195 [==============================] - 27s 137ms/step - loss: 0.2258 - accuracy:
0.9129 - val_loss: 0.4071 - val_accuracy: 0.8243
<tensorflow.python.keras.callbacks.History at 0x21fb632da90>
```

# 7.5 LSTM

- **LSTM 셀 구현**
  - 다음과 같이 모델에 대한 평가를 확인하는 코드를 추가

**코드 7-14 모델 평가**

```
print("훈련 데이터셋 평가...")
(loss, accuracy) = model.evaluate(train_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))
print("테스트 데이터셋 평가...")
(loss, accuracy) = model.evaluate(test_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))
t1 = time.time()
print('시간:', t1-t0)
```

# 7.5 LSTM

- **LSTM 셀 구현**
  - ❖ 다음은 모델 평가에 대한 실행 결과

```
훈련 데이터셋 평가...
loss=0.1755, accuracy: 93.8301%
테스트 데이터셋 평가...
loss=0.4071, accuracy: 82.4319%
시간: 235.93058919906616
```

  - ❖ RNN을 사용하는 것과 비교할 때 훈련 데이터셋과 검증 데이터셋에 대한 정확도가 높아졌음

# 7.5 LSTM

- **LSTM 계층 구현**
  - ❖ 필요한 라이브러리 및 데이터 호출은 RNN 셀에서의 수행과 동일하므로 생략하며, LSTM 계층을 이용한 네트워크 코드를 살펴보자

코드 7-15 네트워크 생성

```
class LSTM_Build(tf.keras.Model):

    def __init__(self, units):
        super(LSTM_Build, self).__init__()

        self.embedding = tf.keras.layers.Embedding(total_words, embedding_len,
                                                   input_length=max_review_len)
        self.rnn = tf.keras.Sequential([
            tf.keras.layers.LSTM(units, dropout=0.5, return_sequences=True,
                                 unroll=True), ------①
            tf.keras.layers.LSTM(units, dropout=0.5, unroll=True)
        ])
        self.outlayer = tf.keras.layers.Dense(1)
```

# 7.5 LSTM

- **LSTM 계층 구현**

```
def call(self, inputs, training=None):
    x = inputs
    x = self.embedding(x)
    x = self.rnn(x)
    x = self.outlayer(x)
    prob = tf.sigmoid(x)

    return prob
```

# 7.5 LSTM

- **LSTM 계층 구현**
  - ❖ ① LSTM 함수를 사용하여 LSTM 셀을 다수 개 구축할 수 있음(layers.LSTMCell은 셀이 하나였으나, layers.LSTM은 한 번에 셀을 여러 개 구축할 수 있음)

```
tf.keras.layers.LSTM(units, dropout=0.5, return_sequences=True, unroll=True)
                     ⓐ      ⓑ           ⓒ                    ⓓ
```

ⓐ units: 네트워크의 층 수(출력 공간의 차원)

ⓑ dropout: 전체 가중치 중 50% 값을 0으로 설정하여 사용하지 않겠다는 의미

ⓒ return_sequences: 마지막 출력 또는 전체 순서를 반환하는 것

이때 return_sequences=False는 마지막 셀에서 밀집층이 한 번만 적용되었다는 것을

의미

ⓓ unroll: 시간 순서에 따라 입력층과 은닉층에 대한 네트워크를 펼치겠다는 의미

메모리 사용률은 높을 수 있지만 계속 속도는 빨라질 수 있음

# 7.5 LSTM

- **LSTM 계층 구현**
  - layers.LSTMCell과 layers.LSTM의 코드 구현은 거의 비슷함
  - 단지 네트워크의 def call 함수에서 LSTMCell은 다음과 같이 for 문을 사용하여 LSTMCell을 반복 수행한다는 점이 다름
  - 즉, LSTMCell은 셀 단위로 수행되므로 다수 셀을 수행하려면 for 문처럼 반복적 수행이 필요함
  - 다음은 LSTMCell과 LSTM을 구현하기 위한 예시 코드

```
#LSTMCell
for word in tf.unstack(x, axis=1):
    out0, state0 = self.RNNCell0(word, state0, training)
    out1, state1 = self.RNNCell1(out0, state1, training)


#LSTM
x = self.rnn(x)
```

# 7.5 LSTM

- **LSTM 계층 구현**
  - 생성된 네트워크를 활용하여 모델을 훈련시켜 보자(RNNCell과 동일한 코드이지만 결과를 확인하려고 또 한 번 실행)

코드 7-16 모델 훈련

```
import time
units = 64
epochs = 4
t0 = time.time()

model = LSTM_Build(units)

model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss=tf.losses.BinaryCrossentropy(),
              metrics=['accuracy'],
              experimental_run_tf_function=False)

model.fit(train_data, epochs=epochs, validation_data=test_data, validation_freq=2)
```

# 7.5 LSTM

- **LSTM 계층 구현**
  - 다음은 모델을 훈련시킨 결과

```
Epoch 1/4
195/195 [==============================] - 24s 124ms/step - loss: 0.4885 - accuracy:
0.7488
Epoch 2/4
195/195 [==============================] - 37s 192ms/step - loss: 0.3153 - accuracy:
0.8706 - val_loss: 0.3548 - val_accuracy: 0.8423
Epoch 3/4
195/195 [==============================] - 23s 116ms/step - loss: 0.2572 - accuracy:
0.8968
Epoch 4/4
195/195 [==============================] - 35s 179ms/step - loss: 0.2148 - accuracy:
0.9183 - val_loss: 0.3987 - val_accuracy: 0.8340
<tensorflow.python.keras.callbacks.History at 0x21fbac26c88>
```

# 7.5 LSTM

- **LSTM 계층 구현**
  - ❖ 이제 모델을 평가해 보자

**코드 7-17 모델 평가**

```
print("훈련 데이터셋 평가...")
(loss, accuracy) = model.evaluate(train_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))
print("테스트 데이터셋 평가...")
(loss, accuracy) = model.evaluate(test_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))

t1 = time.time()
print('시간:', t1-t0)
```

# 7.5 LSTM

- **LSTM 계층 구현**
  - 다음은 모델을 평가한 출력 결과

```
훈련 데이터셋 평가...
loss=0.1413, accuracy: 95.6090%
테스트 데이터셋 평가...
loss=0.3987, accuracy: 83.4014%
시간: 158.35215830802917
```

  - LSTMCell을 사용할 때처럼 훈련 데이터셋과 테스트 데이터셋의 정확도가 비슷함

# 게이트 순환 신경망(GRU)

# 7.6 게이트 순환 신경망(GRU)

- **게이트 순환 신경망(GRU)**
  - GRU(Gated Recurrent Unit)는 게이트 메커니즘이 적용된 RNN 프레임워크의 한 종류이면서 LSTM보다 구조가 간단함

# 7.6 게이트 순환 신경망(GRU)

- **GRU 구조**
  - GRU는 LSTM에서 사용하는 망각 게이트와 입력 게이트를 하나로 합친 것이며, 별도의 업데이트 게이트로 구성되어 있음
  - 하나의 게이트 컨트롤러(gate controller)가 망각 게이트와 입력 게이트를 모두 제어함
  - 게이트 컨트롤러가 1을 출력하면 망각 게이트는 열리고 입력 게이트는 닫히며, 반대로 0을 출력하면 망각 게이트는 닫히고 입력 게이트는 열림
  - 즉, 이전 기억이 저장될 때마다 단계별 입력은 삭제
  - GRU는 출력 게이트가 없어 전체 상태 벡터가 매 단계마다 출력되며, 이전 상태의 어느 부분이 출력될지 제어하는 새로운 게이트 컨트롤러가 별도로 존재함

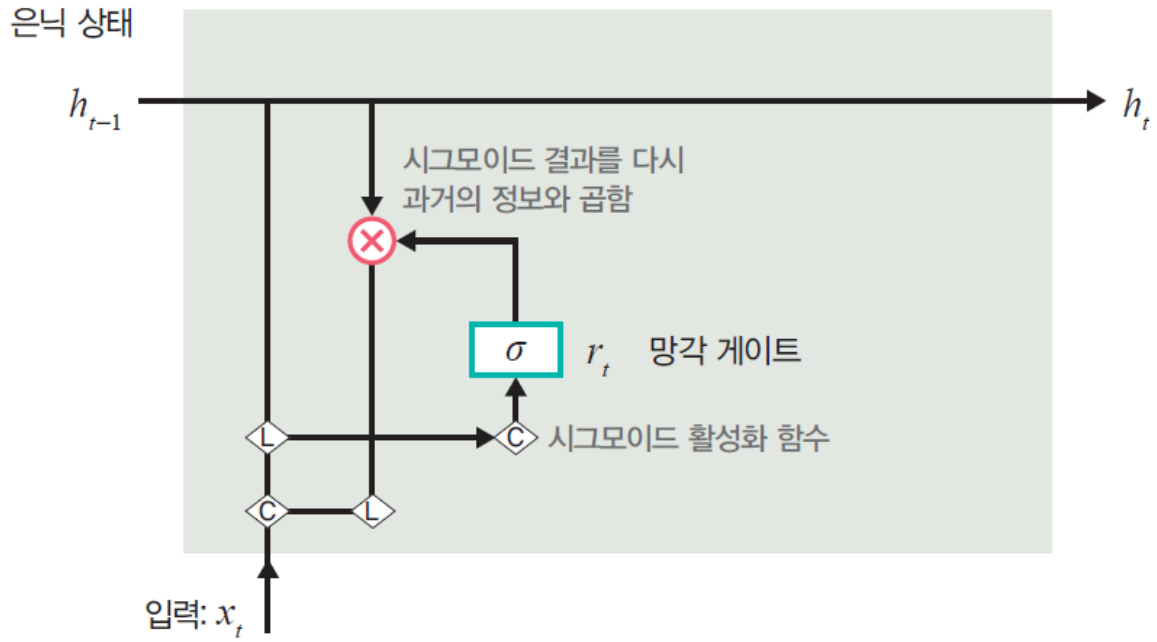
# 7.6 게이트 순환 신경망(GRU)

- **GRU 구조**

  **망각 게이트**

  - 망각 게이트(reset gate)는 과거 정보를 적당히 초기화(reset)시키려는 목적으로 시그모이드 함수를 출력으로 이용하여 (0,1) 값을 이전 은닉층에 곱함
  - 이전 시점의 은닉층 값에 현시점의 정보에 대한 가중치를 곱한 것으로 수식은 다음과 같음

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$



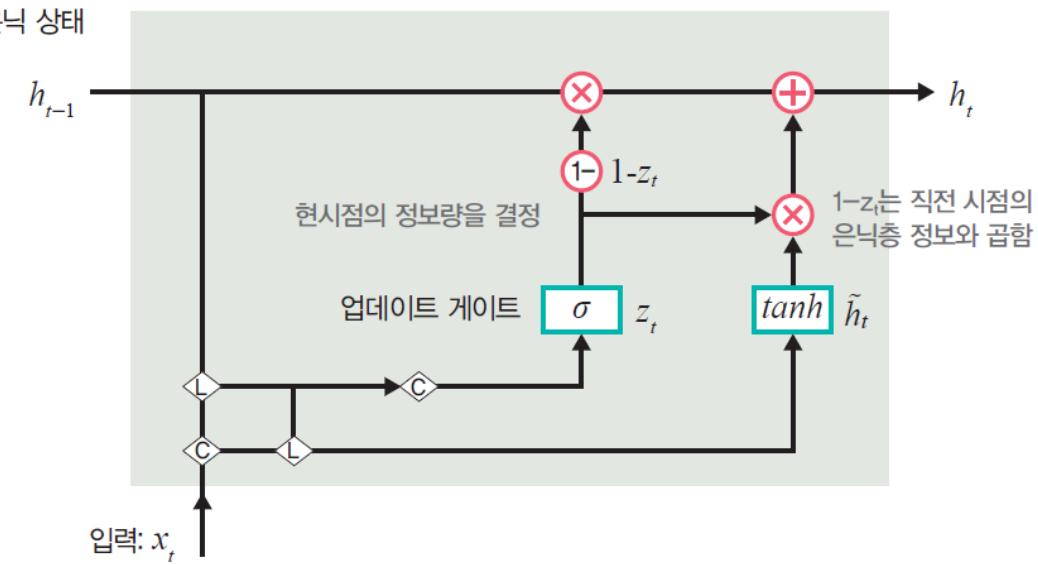
# 7.6 게이트 순환 신경망(GRU)

- **GRU 구조**

  **업데이트 게이트**

  - 업데이트 게이트(update gate)는 과거와 현재 정보의 최신화 비율을 결정하는 역할을 함
  - 시그모이드로 출력된 결과($z_t$)는 현시점의 정보량을 결정하고 1에서 뺀 값(1- $z_t$)은 직전 시점의 은닉층 정보와 곱함
  - 이를 수식으로 나타내면 다음과 같음

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$



# 7.6 게이트 순환 신경망(GRU)

- **GRU 구조**

  **후보군**

  - 후보군(candidate)은 현시점의 정보에 대한 후보군을 계산
  - 과거 은닉층의 정보를 그대로 이용하지 않고 망각 게이트의 결과를 이용하여 후보군을 계산

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$

(*는 점 단위 연산(pointwise operation)입니다. 예를 들어 벡터를 더할 때 각각의 차원(dimension)에 맞게 곱하거나 더하는 것이 가능해집니다.)

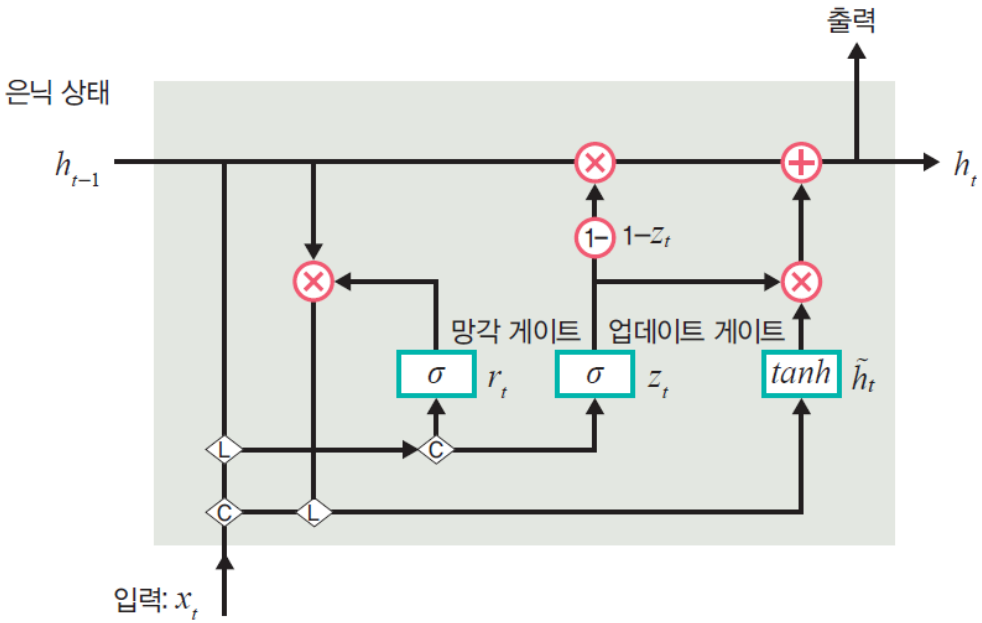
# 7.6 게이트 순환 신경망(GRU)

- **GRU 구조**

  **은닉층 계산**

  - 마지막으로 업데이트 게이트 결과와 후보군 결과를 결합하여 현시점의 은닉층을 계산
  - 시그모이드 함수의 결과는 현시점에서 결과에 대한 정보량을 결정하고, 1-시그모이드 함수의 결과는 과거의 정보량을 결정
  - 이것을 수식으로 나타내면 다음과 같음

$$h_t = (1 - z_t) * h_{t-1} + z_t \times \tilde{h}_t$$



# 7.6 게이트 순환 신경망(GRU)

- **GRU 셀 구현**
  - ❖ 필요한 라이브러리 및 데이터 호출은 RNN 셀에서의 수행과 동일하므로 생략하며, GRU 셀을 이용한 네트워크 코드를 살펴보자

코드 7-18 네트워크 생성

```
class GRU_Build(tf.keras.Model):

    def __init__(self, units):
        super(GRU_Build, self).__init__()

        self.state0 = [tf.zeros([batch_size, units])]
        self.state1 = [tf.zeros([batch_size, units])]

        self.embedding = tf.keras.layers.Embedding(total_words, embedding_len,
                                                   input_length=max_review_len)
        self.RNNCell0 = tf.keras.layers.GRUCell(units, dropout=0.5) ------①
        self.RNNCell1 = tf.keras.layers.GRUCell(units, dropout=0.5)
        self.outlayer = tf.keras.layers.Dense(1)
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 셀 구현**

```
def call(self, inputs, training=None):
    x = inputs
    x = self.embedding(x)
    state0 = self.state0 ------초기 상태는 모두 0으로 설정
    state1 = self.state1
    for word in tf.unstack(x, axis=1):
        out0, state0 = self.RNNCell0(word, state0, training)
        out1, state1 = self.RNNCell1(out0, state1, training)

    x = self.outlayer(out1)
    prob = tf.sigmoid(x)

    return prob
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 셀 구현**
  - ❖ ① GRU의 셀 클래스를 의미
    - 첫 번째 인자: 메모리 셀의 개수
    - dropout: 전체 가중치 중 50% 값을 0으로 설정하여 사용하지 않겠다는 의미

# 7.6 게이트 순환 신경망(GRU)

- **GRU 셀 구현**
  - 생성된 네트워크를 활용하여 모델을 훈련시킴(RNNCell과 동일한 코드이지만 결과를 확인하려고 또 한 번 실행)

코드 7-19 모델 훈련

```
import time
units = 64
epochs = 4
t0 = time.time()

model = GRU_Build(units)

model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss=tf.losses.BinaryCrossentropy(),
              metrics=['accuracy'],
              experimental_run_tf_function=False)

model.fit(train_data, epochs=epochs, validation_data=test_data, validation_freq=2)
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 셀 구현**
  - 다음은 모델을 훈련시킨 결과

```
Epoch 1/4
195/195 [==============================] - 17s 86ms/step - loss: 0.5182 - accuracy:
0.7264
Epoch 2/4
195/195 [==============================] - 25s 129ms/step - loss: 0.3262 - accuracy:
0.8627 - val_loss: 0.3595 - val_accuracy: 0.8411
Epoch 3/4
195/195 [==============================] - 17s 90ms/step - loss: 0.2680 - accuracy:
0.8926
Epoch 4/4
195/195 [==============================] - 24s 121ms/step - loss: 0.2264 - accuracy:
0.9123 - val_loss: 0.3955 - val_accuracy: 0.8282
<tensorflow.python.keras.callbacks.History at 0x21fc6c48710>
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 셀 구현**
  - ❖ 이제 모델을 평가해 보자

코드 7-20 모델 평가

```
print("훈련 데이터셋 평가...")
(loss, accuracy) = model.evaluate(train_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))
print("테스트 데이터셋 평가...")
(loss, accuracy) = model.evaluate(test_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))

t1 = time.time()
print('시간:', t1-t0)
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 셀 구현**
  - 다음은 모델을 평가한 출력 결과

```
훈련 데이터셋 평가...
loss=0.1771, accuracy: 94.1747%
테스트 데이터셋 평가...
loss=0.3955, accuracy: 82.8245%
시간: 162.60217928886414
```

  - LSTM과 비교했을 때 훈련 데이터셋에 대한 정확도가 약간 낮음
  - 전반적인 정확도는 나쁘지 않음

# 7.6 게이트 순환 신경망(GRU)

- **GRU 계층 구현**
  - ❖ 필요한 라이브러리 및 데이터 호출은 RNN 셀에서의 수행과 동일하므로 생략하며, GRU 계층을 이용한 네트워크 코드를 살펴보자

코드 7-21 네트워크 생성

```
class GRU_Build(tf.keras.Model):

    def __init__(self, units):
        super(GRU_Build, self).__init__()

        self.embedding = tf.keras.layers.Embedding(total_words, embedding_len,
                                                   input_length=max_review_len)
        self.rnn = tf.keras.Sequential([
            tf.keras.layers.GRU(units, dropout=0.5, return_sequences=True, unroll=True),
            tf.keras.layers.GRU(units, dropout=0.5, unroll=True)
        ])
        self.outlayer = tf.keras.layers.Dense(1)
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 계층 구현**

```
def call(self, inputs, training=None):
    x = inputs
    x = self.embedding(x)
    x = self.rnn(x)
    x = self.outlayer(x)
    prob = tf.sigmoid(x)

    return prob
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 계층 구현**
  - GRUCell과 GRU의 코드 역시 거의 비슷함
  - GRUCell은 셀 단위로 수행되므로 다수 셀을 수행하려면 for 문처럼 반복적 수행이 필요함
  - 다음은 GRUCell과 GRU를 구현하기 위한 예시 코드

```
#GRUCell
for word in tf.unstack(x, axis=1):
    out0, state0 = self.RNNCell0(word, state0, training)
    out1, state1 = self.RNNCell1(out0, state1, training)


#GRU
x = self.rnn(x)
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 계층 구현**
  - ❖ 이제 생성된 네트워크를 활용하여 모델을 훈련시킴(RNNCell과 동일한 코드이지만 결과를 확인하려고 또 한 번 실행)

코드 7-22 모델 훈련

```
import time
units = 64
epochs = 4
t0 = time.time()

model = GRU_Build(units)

model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss=tf.losses.BinaryCrossentropy(),
              metrics=['accuracy'],
              experimental_run_tf_function=False)

model.fit(train_data, epochs=epochs, validation_data=test_data, validation_freq=2)
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 계층 구현**
  - 다음은 모델을 훈련시킨 결과

```
Epoch 1/4
195/195 [==============================] - 23s 120ms/step - loss: 0.5033 - accuracy:
0.7383
Epoch 2/4
195/195 [==============================] - 39s 201ms/step - loss: 0.3166 - accuracy:
0.8670 - val_loss: 0.3613 - val_accuracy: 0.8435
Epoch 3/4
195/195 [==============================] - 24s 124ms/step - loss: 0.2569 - accuracy:
0.8970
Epoch 4/4
195/195 [==============================] - 39s 200ms/step - loss: 0.2165 - accuracy:
0.9168 - val_loss: 0.4122 - val_accuracy: 0.8338
<tensorflow.python.keras.callbacks.History at 0x21f80fd5fd0>
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 계층 구현**
  - ❖ 이제 모델을 평가해 보자

코드 7-23 모델 평가

```
print("훈련 데이터셋 평가...")
(loss, accuracy) = model.evaluate(train_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))
print("테스트 데이터셋 평가...")
(loss, accuracy) = model.evaluate(test_data, verbose=0)
print("loss={:.4f}, accuracy: {:.4f}%".format(loss,accuracy * 100))

t1 = time.time()
print('시간:', t1-t0)
```

# 7.6 게이트 순환 신경망(GRU)

- **GRU 계층 구현**
  - 다음은 모델을 평가한 출력 결과

```
훈련 데이터셋 평가...
loss=0.1378, accuracy: 95.5889%
테스트 데이터셋 평가...
loss=0.4122, accuracy: 83.3814%
시간: 190.3636932373047
```

  - LSTM과 성능이 유사함

# RNN, LSTM, GRU 성능 비교

# 7.7 RNN, LSTM, GRU 성능 비교

- **RNN, LSTM, GRU 성능 비교**
  - ❖ 앞서 구현했던 모델들의 정확도 및 수행 시간을 정리하면 다음 표와 같음

▼ 표 7-1 모델 여섯 개에 대한 평가

| 구분 | | RNN 셀 | RNN 계층 | LSTM 셀 | LSTM 계층 | GRU 셀 | GRU 계층 |
|---|---|---|---|---|---|---|---|
| 훈련 데이터 | 정확도 | 99% | 96% | 94% | 96% | 95% | 94% |
| | 오차 | 0.03 | 0.1 | 0.2 | 0.1 | 0.1 | 0.2 |
| 테스트 데이터 | 정확도 | 80% | 82% | 82% | 83% | 83% | 83% |
| | 오차 | 0.7 | 0.5 | 0.4 | 0.4 | 0.4 | 0.4 |
| 수행 시간 | | 40 | 69 | 235 | 662 | 158 | 162 |

# 7.7 RNN, LSTM, GRU 성능 비교

- **RNN, LSTM, GRU 성능 비교**
  - RNN 셀의 훈련 데이터셋에 대한 정확도가 가장 높으나, 검증 데이터셋에 대한 정확도는 다른 모델에 비해 조금 낮음
  - 훈련 데이터셋과 테스트 데이터셋의 정확도 및 수행 시간을 모두 고려한다면 RNN 계층의 성능이 가장 좋다고 할 수도 있음(물론 수행 시간이 중요하지 않다고 판단할 때는 GRU를 선택할 수도 있음)
  - 모델 여섯 개에 대한 정확도 차이가 크지 않기 때문에 모든 모델을 실행하여 하이퍼파라미터 값을 제일 빨리 찾는 모델을 사용하길 권장

# 양방향 RNN

# 7.8 양방향 RNN

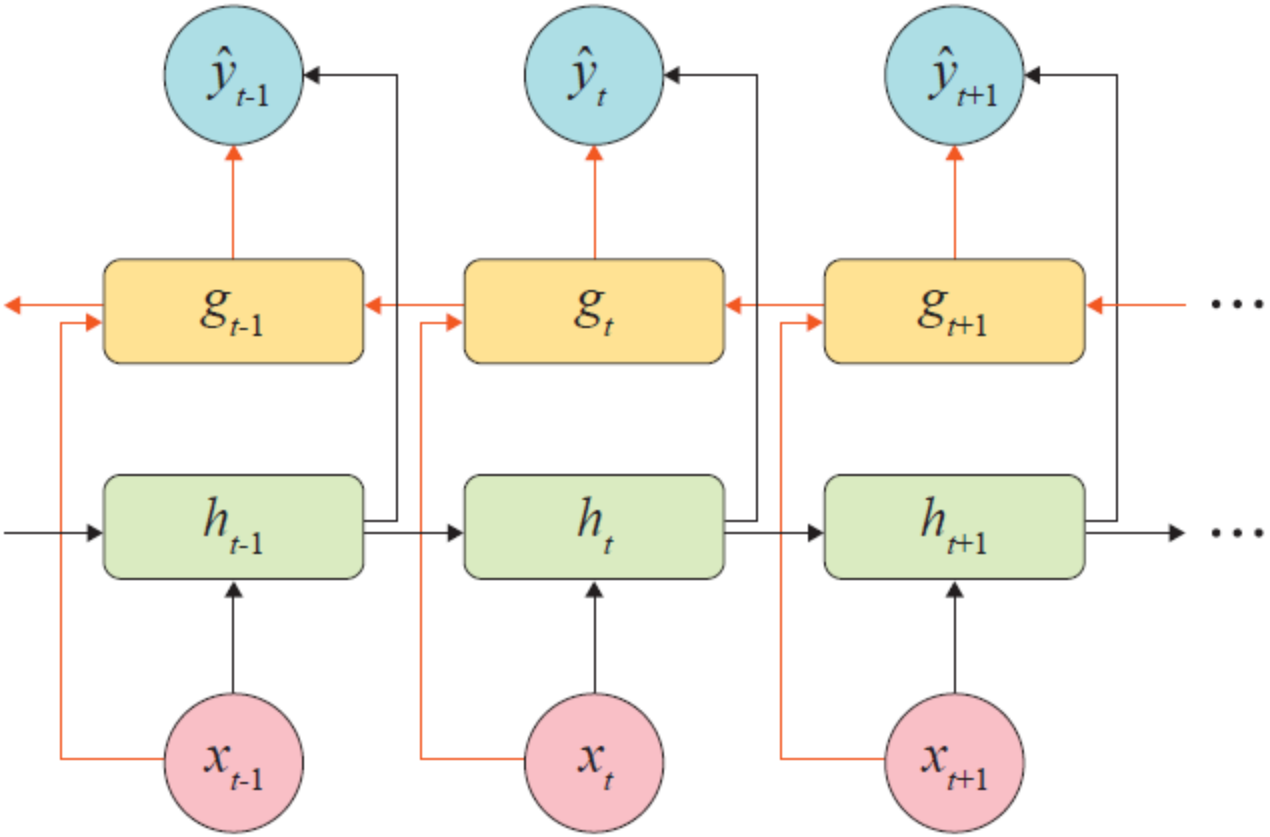
- **양방향 RNN**
  - RNN은 이전 시점의 데이터들을 참고해서 정답을 예측하지만 실제 문제에서는 과거 시점이 아닌 미래 시점의 데이터에 힌트가 있는 경우도 많음
  - 이전 시점의 데이터뿐만 아니라, 이후 시점의 데이터도 함께 활용하여 출력 값을 예측하고자 하는 것이 양방향 RNN(bidirectional RNN)
  - 먼저 양방향 RNN의 구조를 살펴본 후 코드를 구현해 보겠음

# 7.8 양방향 RNN

- **양방향 RNN 구조**
  - 양방향 RNN은 하나의 출력 값을 예측하는 데 메모리 셀 두 개를 사용
  - 첫 번째 메모리 셀은 이전 시점의 은닉 상태(forward states)를 전달받아 현재의 은닉 상태를 계산
  - 다음 그림에서는 초록색 메모리 셀에 해당
  - 두 번째 메모리 셀은 다음 시점의 은닉 상태(backward states)를 전달받아 현재의 은닉 상태를 계산
  - 다음 그림의 주황색 메모리 셀에 해당
  - 이 값 두 개를 모두 출력층에서 출력 값을 예측하는 데 사용

# 7.8 양방향 RNN

▼ 그림 7-24 양방향 RNN



# 7.8 양방향 RNN

- **양방향 RNN 구현**
  - 계속 IMDB 데이터셋을 사용한 예제를 살펴보자
  - IMDB 데이터셋을 사용하여 텐서플로 2로 코드를 작성

**코드 7-24** 모델을 생성하고 훈련

```
import numpy as np
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Embedding, LSTM, Bidirectional
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import imdb

n_unique_words = 10000
maxlen = 200
batch_size = 128
```

# 7.8 양방향 RNN

- **양방향 RNN 구현**

```
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=n_unique_words)
x_train = sequence.pad_sequences(x_train, maxlen=maxlen) ------ 데이터 길이가 같지 않을 때 일정한
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)          길이로 맞추어 줍니다.
y_train = np.array(y_train) ------ y_train을 배열로 생성
y_test = np.array(y_test)

model = Sequential()
model.add(Embedding(n_unique_words, 128, input_length=maxlen))
model.add(Bidirectional(LSTM(64))) ------ LSTM에 양방향 RNN을 적용
model.add(Dropout(0.5)) ------ 50%만 모델에 반영
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

# 7.8 양방향 RNN

- **양방향 RNN 구현**

```
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=4,
          validation_data=[x_test, y_test])
```

# 7.8 양방향 RNN

- **양방향 RNN 구현**
  - 다음은 모델을 훈련시킨 결과

```
Epoch 1/4
196/196 [==============================] - 156s 796ms/step - loss: 0.4357 - accuracy:
0.7910 - val_loss: 0.0000e+00 - val_accuracy: 0.0000e+00
Epoch 2/4
196/196 [==============================] - 162s 826ms/step - loss: 0.2391 - accuracy:
0.9106 - val_loss: 0.0000e+00 - val_accuracy: 0.0000e+00
Epoch 3/4
196/196 [==============================] - 165s 844ms/step - loss: 0.1733 - accuracy:
0.9392 - val_loss: 0.0000e+00 - val_accuracy: 0.0000e+00
Epoch 4/4
196/196 [==============================] - 173s 883ms/step - loss: 0.1379 - accuracy:
0.9528 - val_loss: 0.0000e+00 - val_accuracy: 0.0000e+00
<tensorflow.python.keras.callbacks.History at 0x21ffd686b00>
```

# 7.8 양방향 RNN

- **양방향 RNN 구현**
  - 모델 구조를 살펴보면, 다음 결과와 같이 양방향 LSTM을 사용하고 있는 것을 확인할 수 있음

**코드 7-25** LSTM 모델 구조 확인

```
model.summary()
```

# 7.8 양방향 RNN

- **양방향 RNN 구현**
  - 다음은 출력된 LSTM 모델의 구조

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 200, 128)          1280000
_________________________________________________________________
bidirectional (Bidirectional (None, 128)               98816
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 129
=================================================================
Total params: 1,378,945
Trainable params: 1,378,945
Non-trainable params: 0
_________________________________________________________________
```

# 7.8 양방향 RNN

- **양방향 RNN 구현**
  - ❖ 모델에 대한 평가를 위해 정확도(accuracy)와 오차(loss)를 확인해 보자

코드 7-26 모델 평가

```
loss, acc = model.evaluate(x_train, y_train, batch_size=384, verbose=1)
print('Training accuracy', model.metrics_names, acc)
print('Training accuracy', model.metrics_names, loss)
loss, acc = model.evaluate(x_test, y_test, batch_size=384, verbose=1)
print('Testing accuracy', model.metrics_names, acc)
print('Testing accuracy', model.metrics_names, loss)
```

# 7.8 양방향 RNN

- **양방향 RNN 구현**
  - ❖ 다음은 모델을 평가한 출력 결과

```
66/66 [==============================] - 21s 314ms/step - loss: 0.0870 - accuracy:
0.9722
Training accuracy ['loss', 'accuracy'] 0.9721599817276001
Training accuracy ['loss', 'accuracy'] 0.08702150732278824
66/66 [==============================] - 21s 325ms/step - loss: 0.3949 - accuracy:
0.8616
Testing accuracy ['loss', 'accuracy'] 0.8615999817848206
Testing accuracy ['loss', 'accuracy'] 0.3949168622493744
```

# 7.8 양방향 RNN

- **양방향 RNN 구현**
  - 훈련 데이터셋은 97%의 정확도이며, 테스트 데이터셋은 86%의 정확도로 나쁘지 않은 결과를 보임
  - 지금까지 RNN 구현 방법을 알아보았음
  - 앞서 살펴본 것처럼 RNN 구현은 어렵지 않음
  - RNN에서 사용되는 데이터는 시계열 데이터로 모델을 적용하기 전에 전처리 과정이 상당히 중요함
  - 대체로 시계열 데이터들은 일반적인 숫자의 나열보다는 한글 및 영문으로 사람의 언어(자연어)로 구현된 데이터가 대부분이기 때문임
  - RNN 구현에서 가장 중요한 것은 데이터에 대한 전처리