

# RNN과 LSTM

---

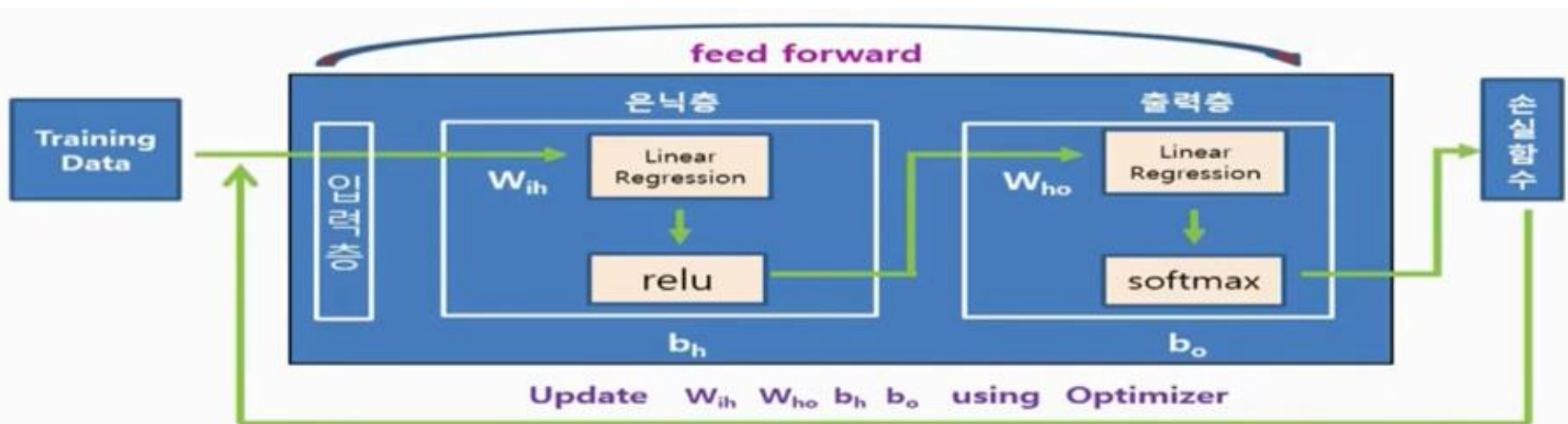
박경미

# 목차

---

- ❖ RNN 개념과 원리
- ❖ Simple RNN 구조와 동작원리
- ❖ LSTM 구조와 동작원리
- ❖ LSTM 활용한 삼성전자 주가 예측

# 1. RNN 구조와 원리- 아키텍처 비교(NN vs. RNN)

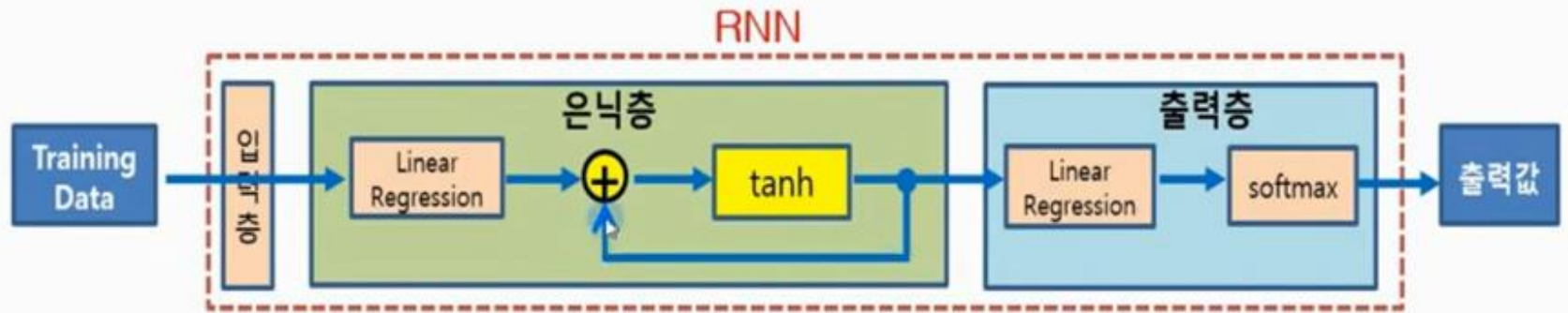


↓ RNN 아키텍처



# 1. RNN 구조와 원리

## ❖ RNN- 순환신경망(Recurrent Neural Network)



- ① 내부적으로 순환(recurrent) 되는 구조를 이용하여
- ② 순서(sequence)가 있는 데이터를 처리하는 데 강점을 가진 신경망



순서(sequence)가 있는 데이터 ?

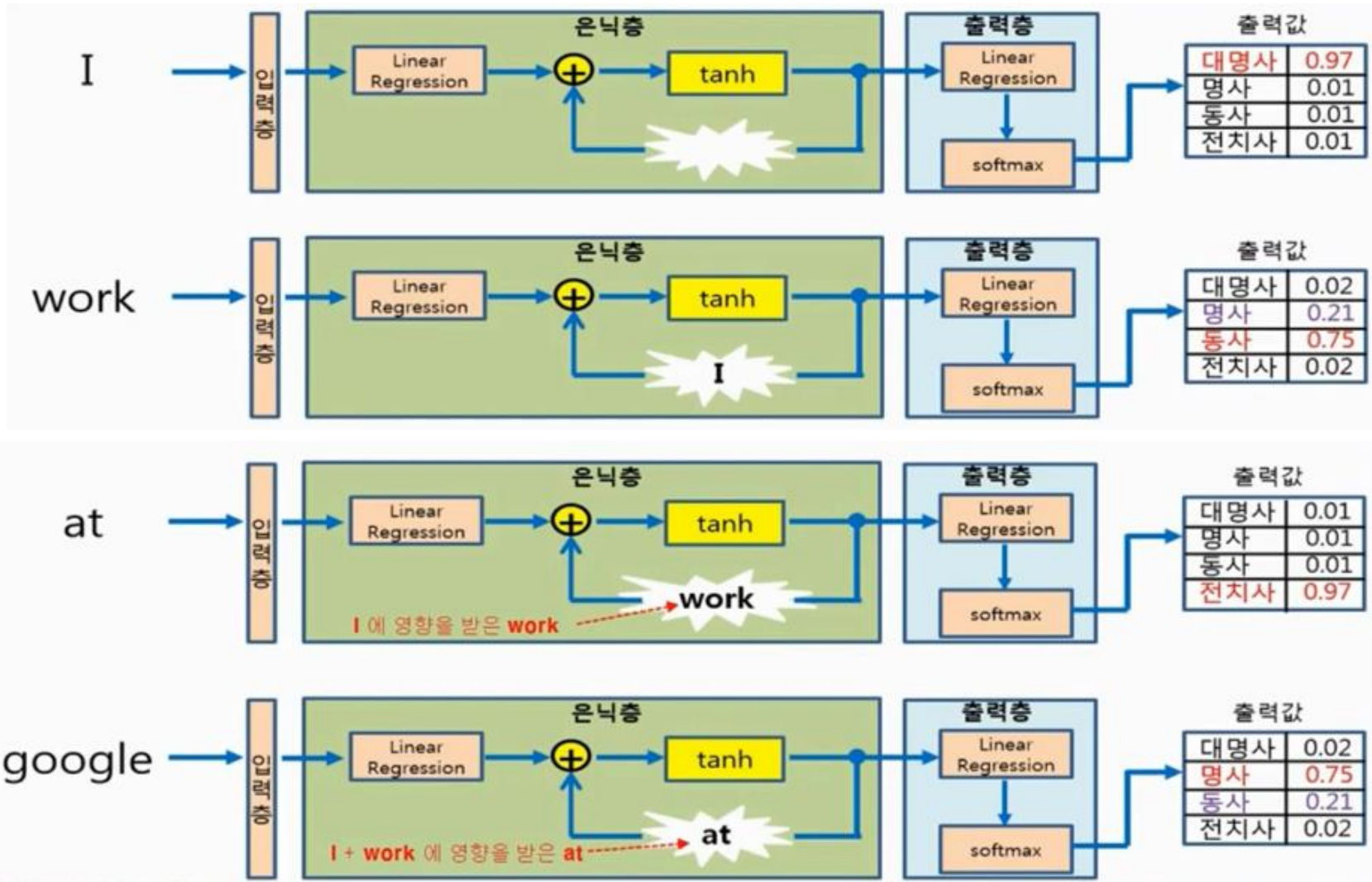
# 1. RNN 구조와 원리

I <u>work</u> at <u>google</u>	→ 나는 구글에 근무한다	
I <u>google</u> at <u>work</u>	→ 나는 회사에서 구글링한다	

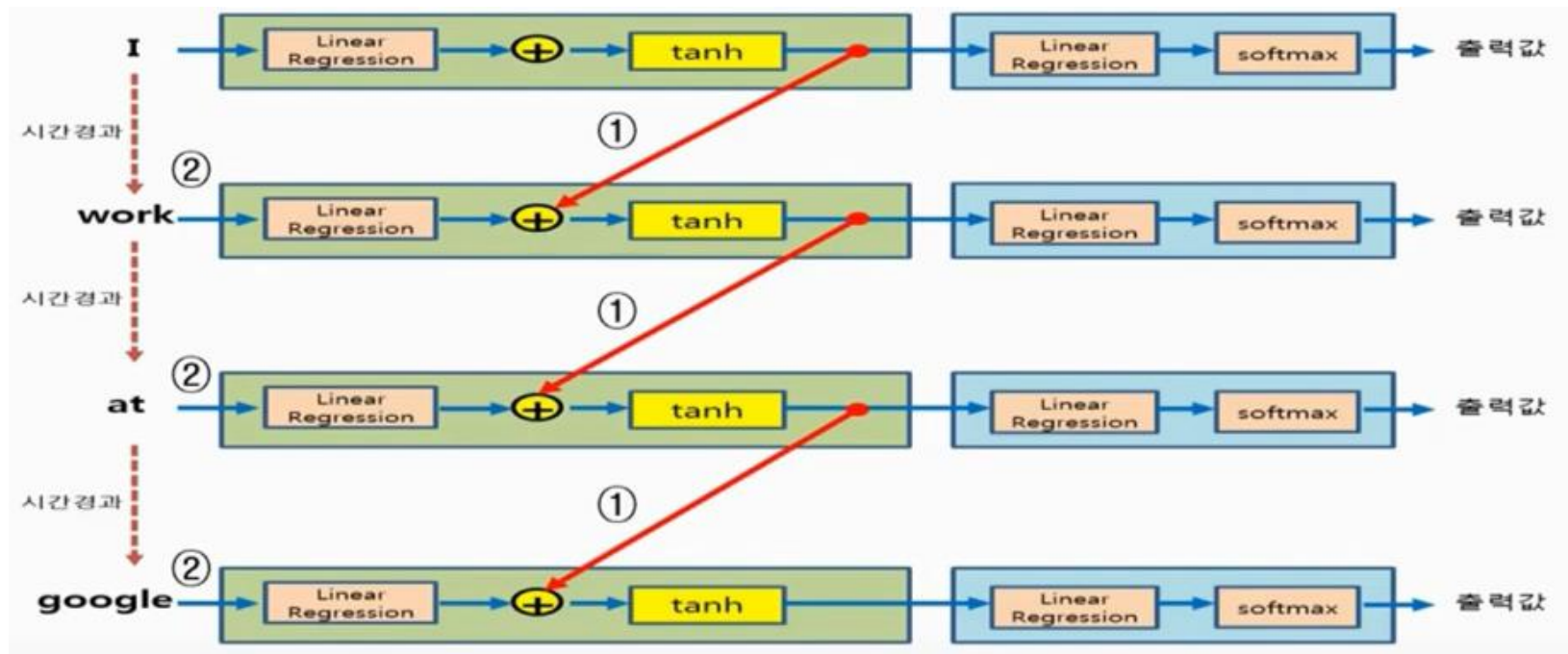
## ❖ 순서(sequence)가 있는 데이터

- 문장이나 음성같은 연속적인 데이터를 말함. 이런 데이터는 문장에서 놓여진 위치(순서)에 따라 의미가 달라지는 것을 알 수 있음
- 즉, 현재 데이터 의미를 알기 위해서는 이전에 놓여 있는 과거 데이터도 알고 있어야 함(I work/ I google[대명사+동사], at google/ at work[전치사+명사])
- RNN은 이러한 과거 데이터를 알기 위해서 ① 은닉층내에서 순환(Recurrent) 구조를 이용하여 과거의 데이터를 기억해 두고 있다가 ② 새롭게 입력된 데이터와 은닉층에서 기억하고 있는 데이터를 연결 시켜서 그 의미를 알아내는 기능을 가지고 있음

# 1. RNN 구조와 원리 - 동작원리: 정성적 분석(I work at google)



# 1. RNN 구조와 원리 - 시간 개념 포함



## ❖ 시간 개념을 포함한 RNN 구조

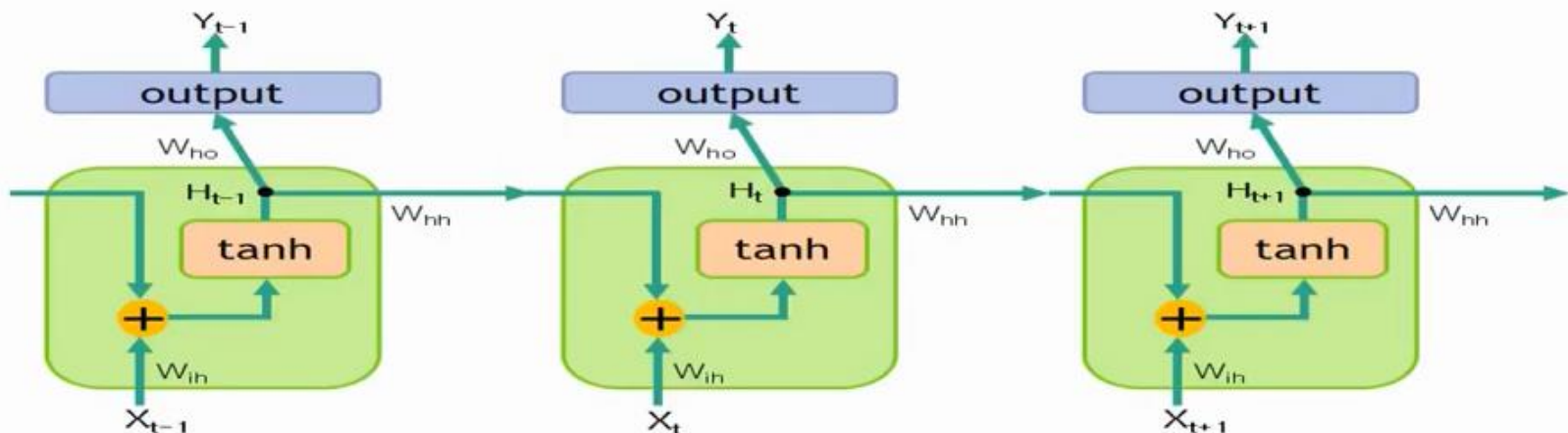
- 순환 구조를 ① 은닉층에서 기억하는 과거의 데이터(붉은색 화살표)와 ② 일정 시간이 지난 후에 입력되는 데이터를 연결시켜 주는 구조로 바꾸어서 생각해볼 수 있음
- 즉 문장이나 음성 같은 순서가 있는 데이터라는 것은 시간의 경과에 따라서 데이터가 순차적으로 들어온다는 것과 같은 의미라는 것을 알수 있음



## 2. Simple RNN 구조와 동작원리

### ❖ SimpleRNN 레이어

- 가장 간단한 형태의 RNN 레이어이며 기본 구조 다음과 같다.



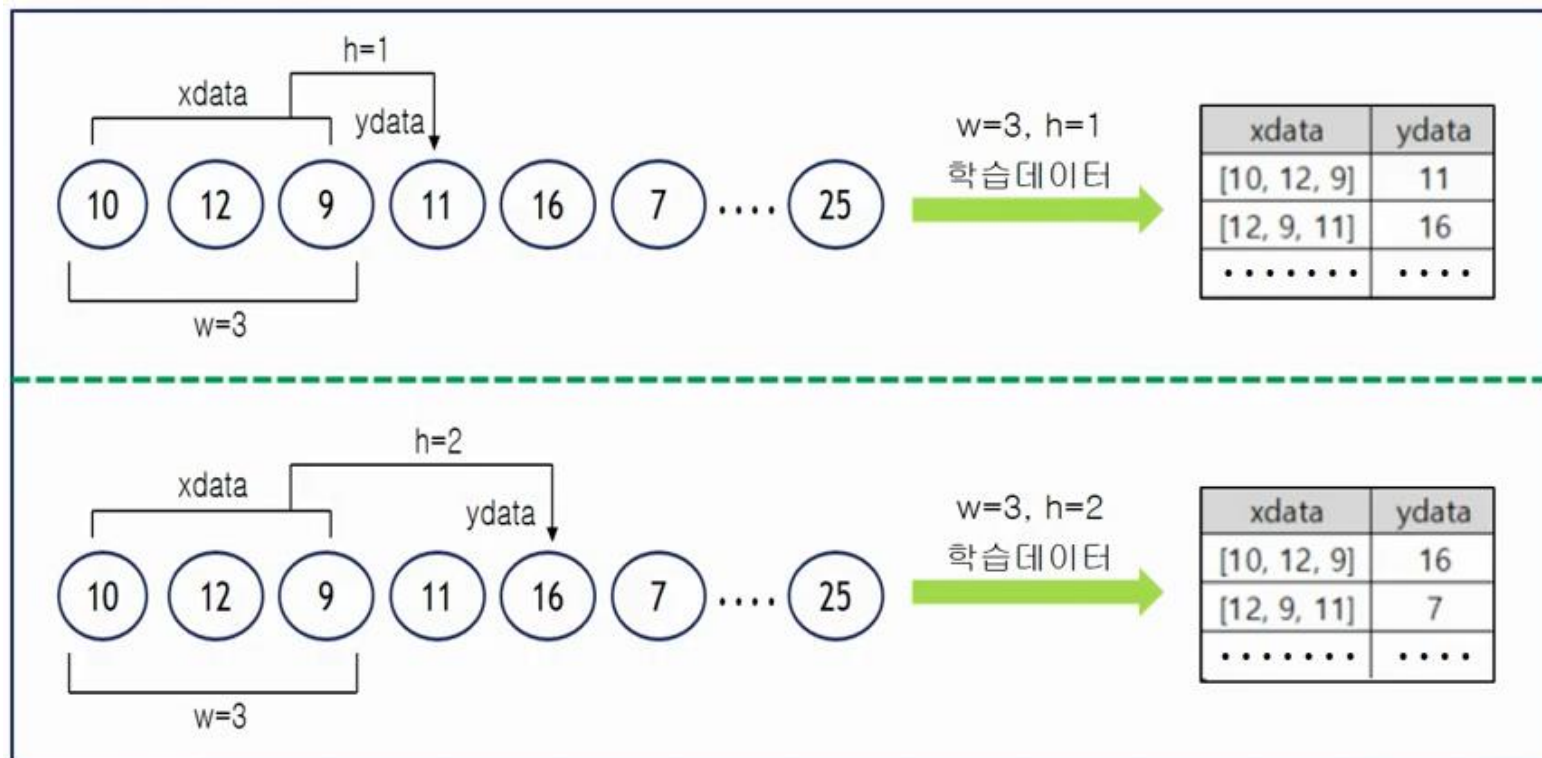
- ✓  $X_{t-1}$ ,  $X_t$ ,  $X_{t+1}$  은 입력 데이터를 나타내고  $H_{t-1}$ ,  $H_t$ ,  $H_{t+1}$  등은 은닉층 개념의 SimpleRNN 레이어 출력 값을, 그리고  $Y_{t-1}$ ,  $Y_t$ ,  $Y_{t+1}$  등은 출력층의 출력 값을 나타냄
- ✓ 학습 대상의 가중치는 ① 입력층과 은닉층 사이의 가중치  $W_{ih}$  ② 시간  $t$  에서의 은닉층과 시간  $t+1$  에서의 은닉층 간의 가중치  $W_{hh}$  ③ 은닉층과 출력층 사이의 가중치  $W_{ho}$
- ✓ 시간  $t$  에서 은닉층 SimpleRNN 레이어 출력  $H_t = \tanh(X_t W_{ih} + H_{t-1} W_{hh})$



## 2. Simple RNN 구조와 동작원리

### ❖ 시계열 데이터 기반 RNN 구조

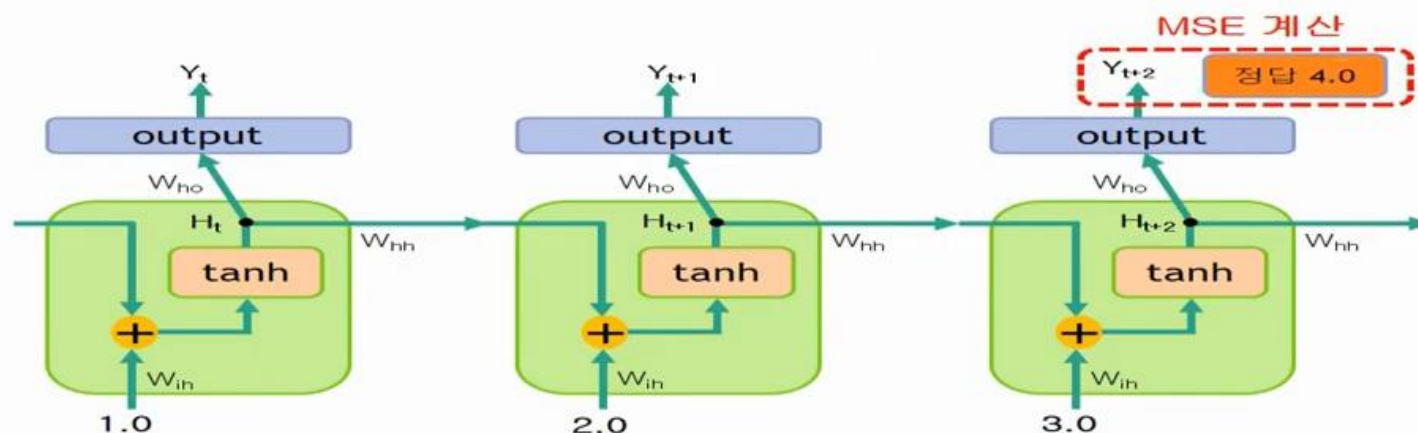
- 시계열 데이터를 이용해서 미래 값을 예측하는 RNN구조라면, 다음과 같은  $w, h$  등을 설정하여 일정한 길이로 패턴을 잘라서 학습 데이터를 만들어야 함
  - 이전 데이터 몇 개를 묶을 것인지를 나타내는 윈도우 크기(window size)  $W$  설정
  - 얼마나 먼 미래 값을 예측할 것인지를 지정하는 수평선 개수(horizon factor)  $h$  설정



## 2. Simple RNN 구조와 동작원리

### ❖ simpleRNN API

- 앞쪽 3개의 숫자를 바탕으로 그 다음에 오는 숫자를 예측하는 경우
- 예를 들어 [1.0, 2.0, 3.0] 입력에 대해서 [4.0] 예측하고, [2.0, 3.0, 4.0] 입력에 대해서 [5.0] 예측하기 위해서 1개의 SimpleRNN 레이어를 가지는 모델을 구축 예



```
tf.keras.layers.SimpleRNN(units=10, activation='tanh', input_shape=(3, 1))
```

일반 신경망의 은닉층 노드 수와 같은 개념. 즉 units=10 의미는 SimpleRNN 레이어 내의 노드 개수는 10개이며 노드 1개당 활성화 함수 1개를 가지고 있음

'relu' 같은 다른 활성화 함수를 사용할 수도 있음

3개의 time-step 데이터를 이용해서 정답을 만든다는 의미이며 window size=3 과 같음

RNN 레이어로 한번에 1개의 데이터가 들어간다는 의미

## 2. Simple RNN 구조와 동작원리

### ❖ RNN 개발 프로세스

시계열 데이터 정의



학습 데이터 생성

RNN 모델을 구현할 때 가장 핵심이 되는 부분은 데이터의 구조이고 RNN 레이어 입력 데이터는 (batch size, time steps, input dims) 같은 구조로 주어져야 함

※ batch size: time steps (=window size) 으로 분리되어 있는 데이터의 총 개수

※ time steps: 몇 개의 데이터를 이용해서 정답을 만들어 내는지를 나타내며 window size 크기와 동일함

※ input dims: RNN 레이어로 한번에 들어가는 데이터의 개수

[예] 3개의 time step 데이터 [1, 2, 3] 이용하여 정답 4 를 만들고, 정답 5 또한 3개의 time step [2,3,4] 데이터로 만들어 진다면, 입력데이터 (2, 3, 1) 형태의 3차원 텐서로 나타내야 함. 즉 RNN 레이어로 한번에 들어가는 데이터 개수는 1개이며 (input dims=1), 이러한 데이터 3개를 이용해서 정답을 만들어 내는데 (time steps=3), 이러한 3개의 time step 데이터가 총 2개 라는 의미임 (batch size=2) . 즉 입력 데이터는 [ [[1],[2],[3]] , [[2],[3],[4]] ] 같은 3차원 텐서가 되도록 변형 후에 입력으로 넣어주어야 함



RNN 모델 구축 및 학습

# simpleRNN Example

## [1]시계열 데이터 $y=0.5\sin(2x)-\cos(x/2)$ 정의

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras import Sequential
```

```
x = np.arange(0, 100, 0.1)
y = 0.5*np.sin(2*x) - np.cos(x/2.0)
```

1,000개  
시계열  
데이터

```
seq_data = y.reshape(-1,1)
```

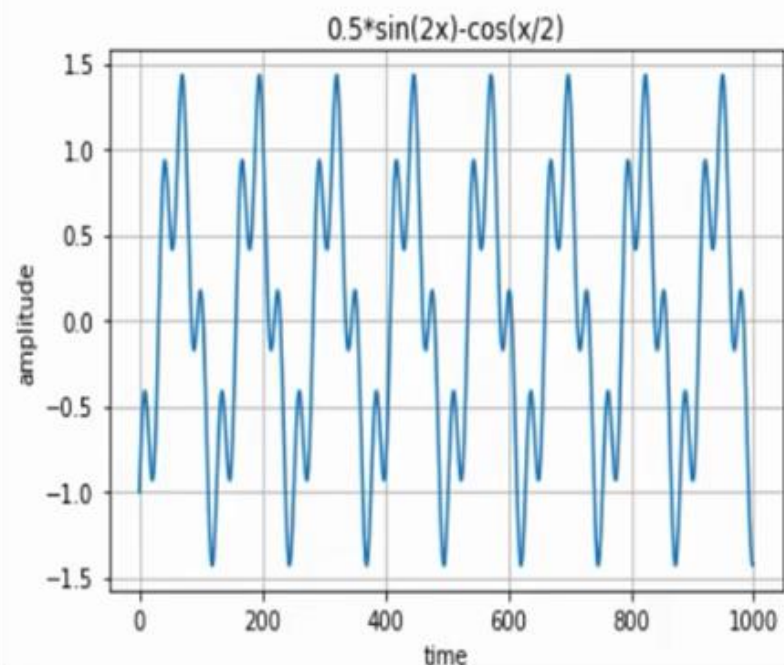
```
print(seq_data.shape)
print(seq_data[:5])
```

RNN 입력에 필수적인  
(batch size, time steps,  
input\_dim) 3차원 텐서 형  
태의 입력 데이터로 쉽게  
만들기 위해서 reshape(-  
1,1) 사용하여 (1000, 1)  
행렬로 바꾸어줌

```
(1000, 1)
[[-1.          ]
 [-0.89941559]
 [-0.80029499]
 [-0.70644984]
 [-0.62138853]]
```

```
plt.grid()
plt.title('0.5*sin(2x)-cos(x/2)')
plt.xlabel('time')
plt.ylabel('amplitude')
plt.plot(seq_data)

plt.show()
```



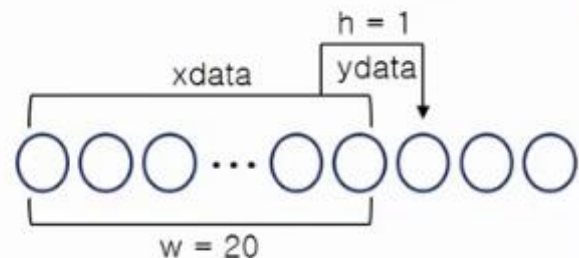


# simpleRNN Example

## [2] 입력 데이터 X, 정답 데이터 Y 생성

```
w = 20      # window size  
h = 1      # horizon factor  
  
X, Y = seq2dataset(seq_data, w, h)  
print(X.shape, Y.shape) ①  
  
(980, 20, 1) (980, 1)
```

```
def seq2dataset(seq, window, horizon):  
    X = [] ← 입력 데이터를 저장하는 list  
    Y = [] ← 정답 데이터를 저장하는 list  
  
    for i in range(len(seq)-(window+horizon)+1):  
        x = seq[i:(i+window)] ②  
        y = (seq[i+window+horizon-1])  
  
        X.append(x)  
        Y.append(y)  
  
    return np.array(X), np.array(Y)
```



① 시계열 데이터 `seq_data`로 부터, window size `w`, horizon factor `h`에 맞게 RNN 입력 데이터 `X`, 정답 데이터 `Y` 생성함, 이때 리턴되는 입력 데이터 `X.shape = (batch size, time steps, input dims)`

② `seq[i:(i+window)]` 슬라이싱 이용하여 `[.., .., ..]` 형상으로 `x` 데이터를 생성함

③ `x.shape = [.., .., ..]`은 2차원 행렬인데, `np.array(X)` 통해서 `(batch size, time steps, input dims)` 형상을 가지는 3차원 텐서로 변환되어 리턴됨

# simpleRNN Example

## [3] 트레이닝 데이터/테스트 데이터 분리

```
split_ratio = 0.8

split = int(split_ratio*len(X))

x_train = X[0:split]
y_train = Y[0:split]

x_test = X[split:]
y_test = Y[split:]

print(x_train.shape, y_train.shape,
      x_test.shape, y_test.shape)

(784, 20, 1) (784, 1) (196, 20, 1) (196, 1)
```



## [4] SimpleRNN 모델 구축

```
model = Sequential()

model.add(SimpleRNN(units=128,
                    activation='tanh',
                    input_shape=x_train[0].shape))

model.add(Dense(1))

model.summary()
```

SimpleRNN 계층에 tanh를 활성화 함수로 가지는 노드 수 128개

input\_shape=(20, 1)

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
simple_rnn_1 (SimpleRNN)	(None, 128)	16640
dense_1 (Dense)	(None, 1)	129

Total params: 16,769

Trainable params: 16,769

Non-trainable params: 0

# SimpleRNN Example

## [5] 모델학습(EarlyStop 적용)

```
from tensorflow.keras.callbacks import EarlyStopping

early_stop = EarlyStopping(monitor='val_loss', patience=5)

model.fit(x_train, y_train,
          validation_data=(x_test, y_test),
          epochs=100, batch_size=16,
          callbacks=[early_stop])
```

```
Epoch 1/100
315/315 [=====] - 3s 5ms/step - loss: 8.4391e-04 - mae:
0.0115 - val_loss: 8.1629e-04 - val_mae: 0.0191
Epoch 2/100
315/315 [=====] - 1s 4ms/step - loss: 9.1896e-05 - mae:
0.0062 - val_loss: 6.6948e-04 - val_mae: 0.0182
Epoch 3/100
315/315 [=====] - 1s 4ms/step - loss: 7.8710e-05 - mae:
0.0058 - val_loss: 6.0811e-04 - val_mae: 0.0168
Epoch 4/100
315/315 [=====] - 1s 4ms/step - loss: 7.3097e-05 - mae:
0.0057 - val_loss: 4.9352e-04 - val_mae: 0.0161
Epoch 5/100
315/315 [=====] - 1s 4ms/step - loss: 7.0582e-05 - mae:
Epoch 25/100
315/315 [=====] - 1s 4ms/step - loss: 2.6528e-05 - mae:
0.0034 - val_loss: 1.9279e-04 - val_mae: 0.0099
<tensorflow.python.keras.callbacks.History at 0x7f5ed008bc90>
```



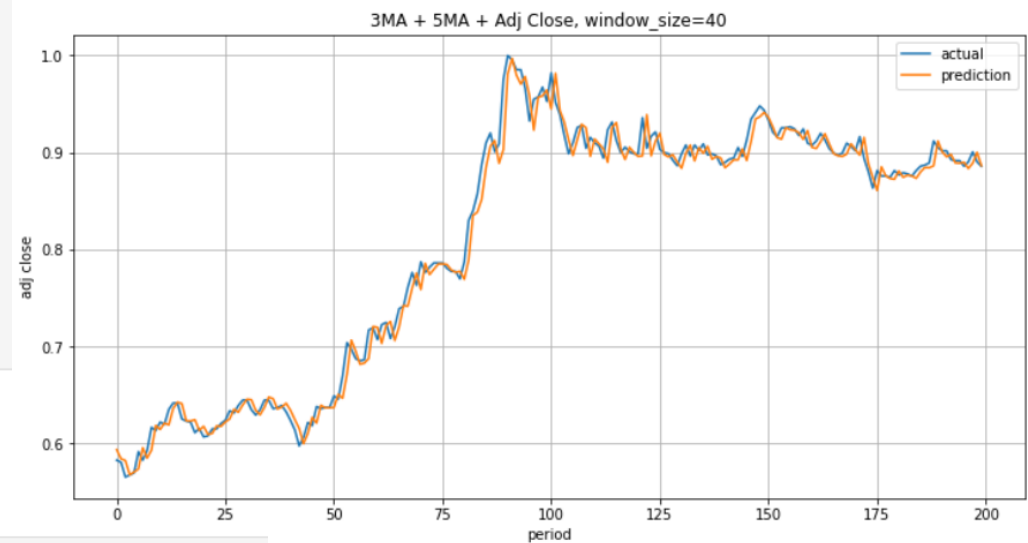
# SimpleRNN Example

예측을 통한 정답과의 비교 (오차계산 MAPE 사용, 평균절대값백분율오차)

```
pred = model.predict(x_test)

plt.figure(figsize=(12, 6))
plt.title('3MA + 5MA + Adj Close, window_size=40')
plt.ylabel('adj close')
plt.xlabel('period')
plt.plot(y_test, label='actual')
plt.plot(pred, label='prediction')
plt.grid()
plt.legend(loc='best')

plt.show()
```



# 평균절대값백분율오차계산 (MAPE)

```
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

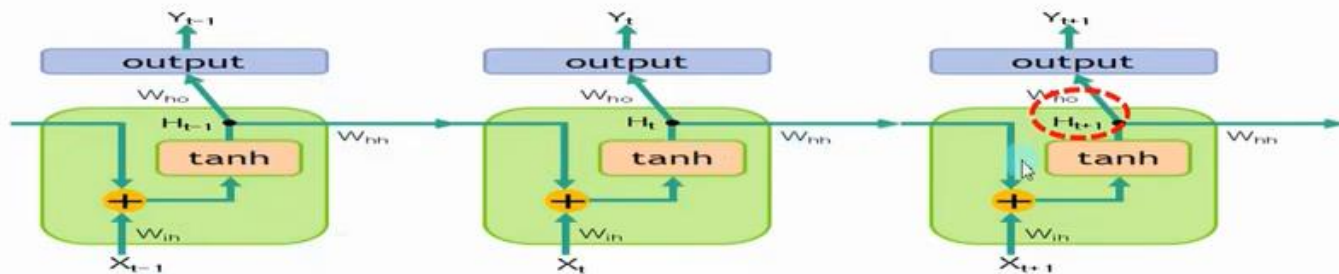
0.01224175273632353

### 3. LSTM 구조와 동작원리

#### ❖ SimpleRNN 단점

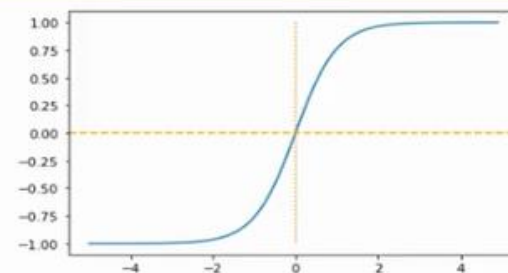
##### ▪ 장기 의존성 문제(the problems of long-term dependency)

- 입력데이터가 많아 질수록 SimpleRNN 레이어가, 즉 은닉 층에서 보관하는 과거의 정보가 마지막 레이어까지 충분히 전달되지 못하는 현상을 의미함



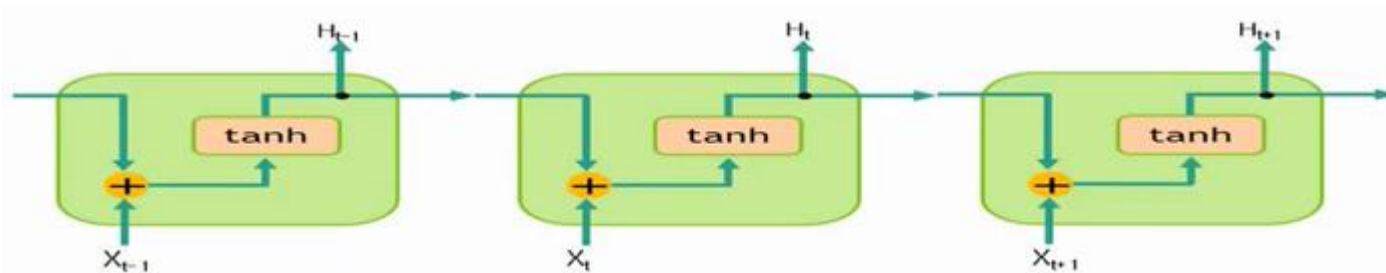
$$\begin{aligned} H_{t-1} &= \tanh(H_{t-2}W_{hh} + X_{t-1}W_{ih}) & H_t &= \tanh(H_{t-1}W_{hh} + X_tW_{ih}) & H_{t+1} &= \tanh(H_tW_{hh} + X_{t+1}W_{ih}) \end{aligned}$$

시간  $t+1$  에서의 은닉층 출력 값  $H_{t+1}$  은  $\tanh$  함수의 출력 값. 그런데  $\tanh$  출력 값 범위는  $-1 < \tanh < 1$  , 즉 1 보다 작은 값은 곱할수록 더 작아지기 때문에 시간이 흐를수록 가장 앞에 있는  $H_{t-1}$  같은 은닉층 정보가 마지막 레이어까지 충분히 전달되는 것이 불가함

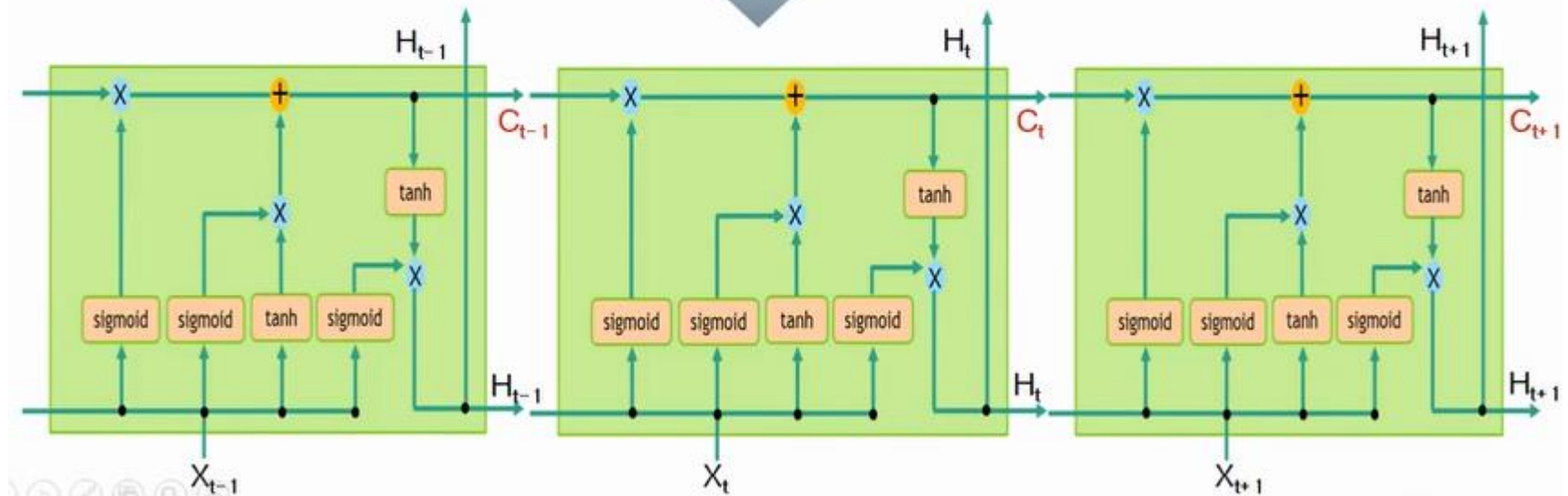


# SimpleRNN vs LSTM

- ❖ LSTM 레이어 시간  $t$ 에서의 출력값  $H_t$  이외에, LSTM레이어 사이에서 공유되는 셀 상태(cell state)  $C_t$ 라는 변수가 추가적으로 공유되는 특징이 있음



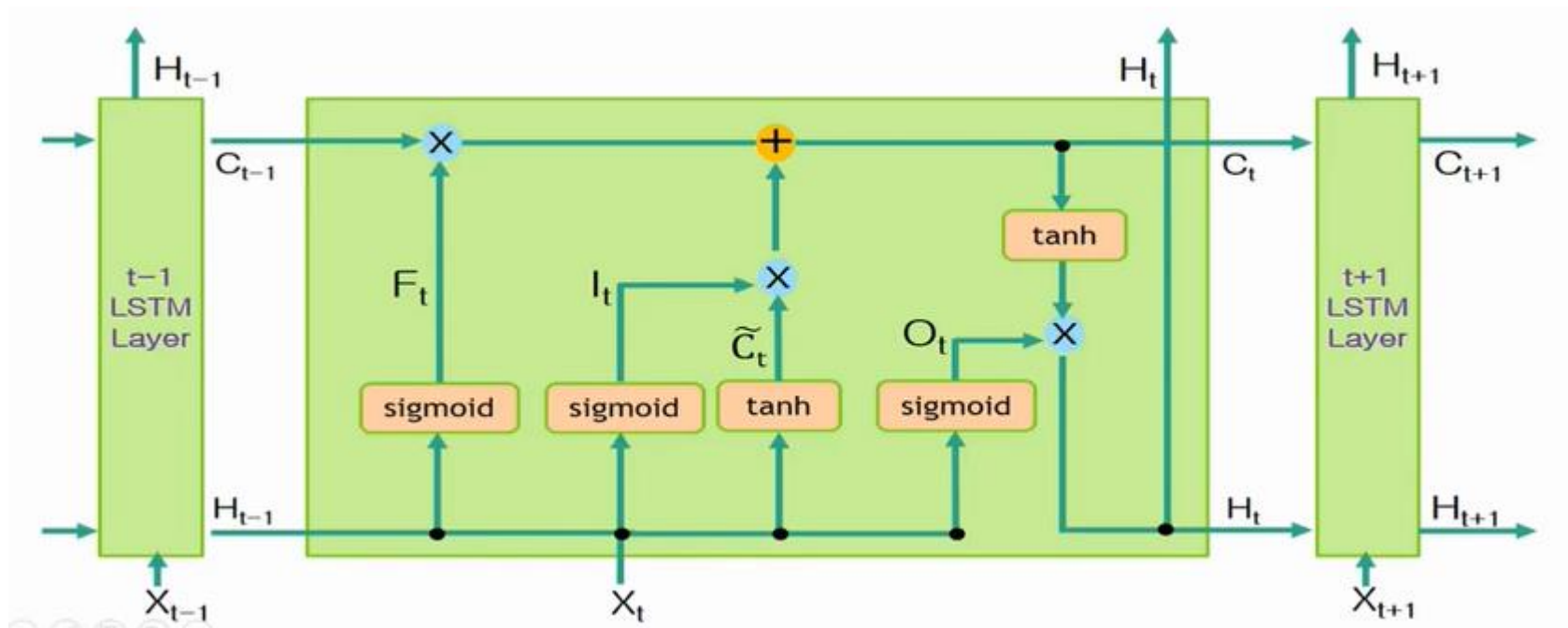
LSTM의 셀 상태가 다음 레이어로 전달되면서 기존의 상태를 보존하므로 장기 의존성 문제를 해결할 수 있음



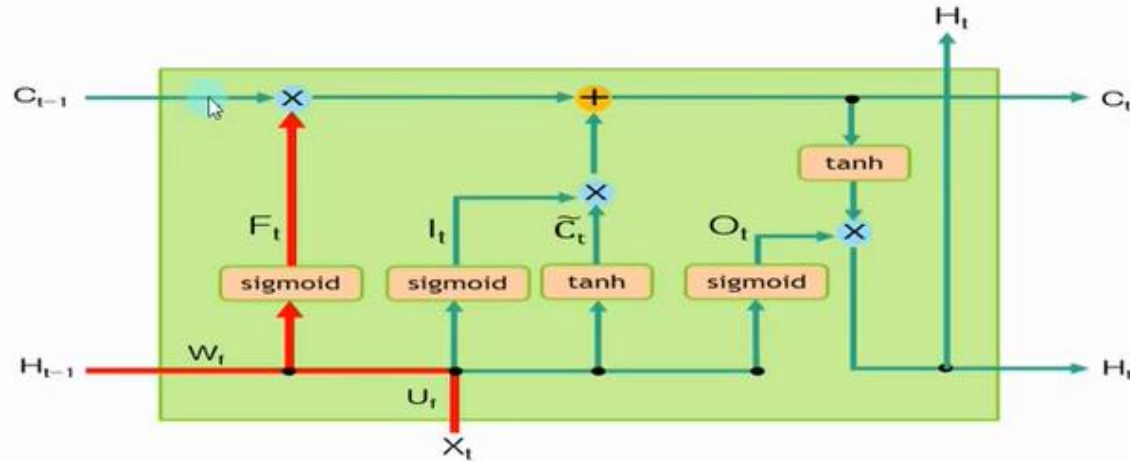
# LSTM 구조 - 개요

❖ LSTM 핵심은 이전 단계 정보를 memory cell에 저장하여 다음 단계로 전달하는 것임

- LSTM은 현재 시점의 정보를 바탕으로 과거 내용을 얼마나 잊을지 또는 기억할지 등을 계산하고, 그 결과에 현재 정보를 추가해서 다음 시점으로 정보를 전달
- 이러한 기능을 구현하기 위해 LSTM을 forget gate, input gate, output gate 등으로 구성되며, 이러한 gate는 memory cell에 정보를 저장하고 다음 단계로 전달하는 역할 수행



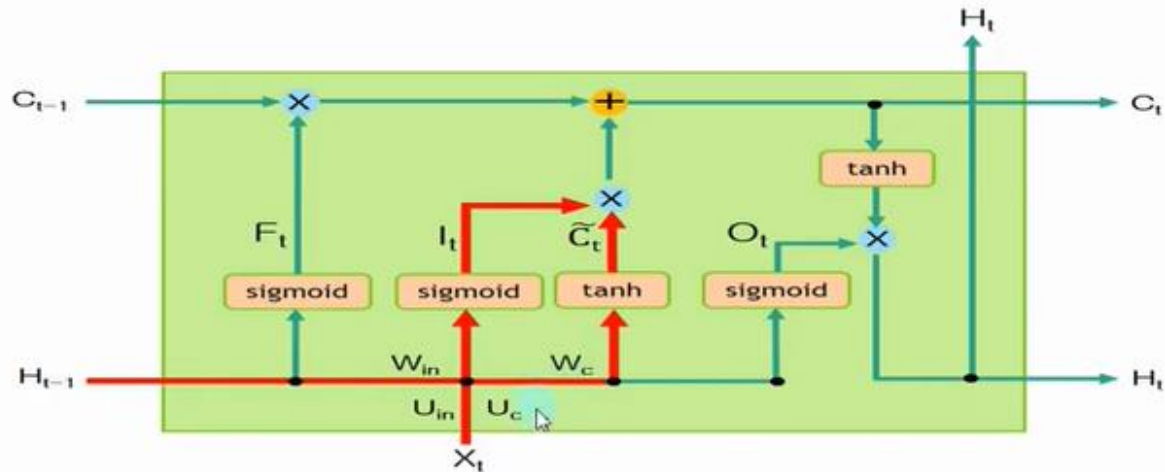
# LSTM 구조 – forget gate



$$F_t = \text{sigmoid}(U_f X_t + W_f H_{t-1} + b_f)$$

- forget gate는 과거의 정보를 얼마나 잊을지(또는 기억할지) 결정하는 게이트이며, ① 현 시점의 데이터  $X_t$  와 과거의 은닉층 값  $H_{t-1}$ 에 각각의 가중치  $W_f$ ,  $U_f$  곱한 후에 ② 그 값들을 더한 후 sigmoid 함수를 적용하는 과정임
- sigmoid 함수 값은 0~1 사이 값을 가지므로, 계산 값이 1에 가깝다면 과거 정보를 많이 활용한다는 의미이고, 만약 sigmoid 값이 0에 가깝다면 과거 정보를 많이 잃게 되는 원리임

# LSTM 구조 -input gate



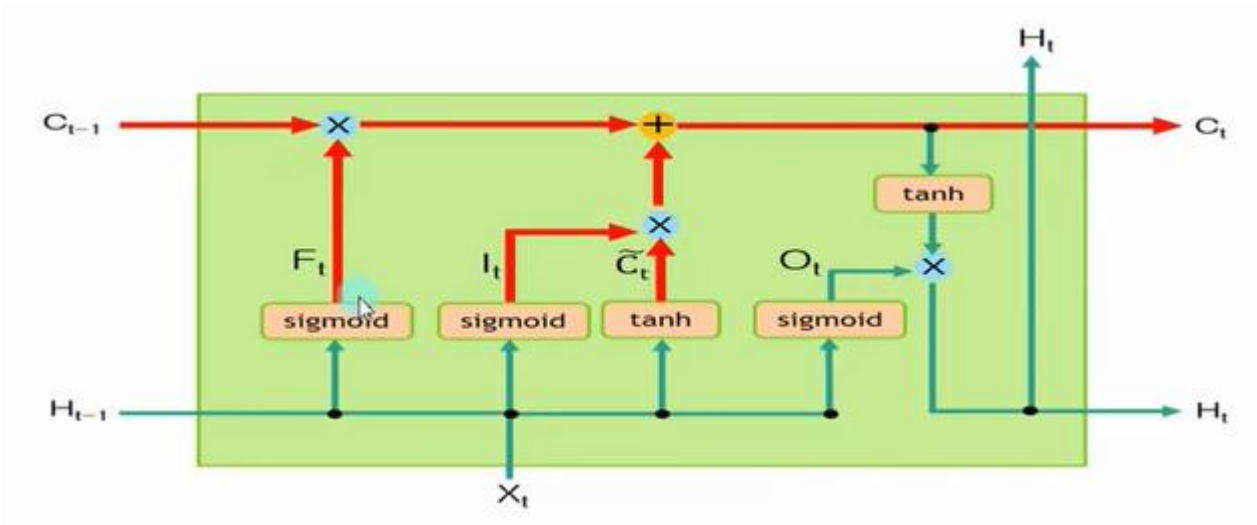
$$\tilde{C}_t = \tanh(U_c X_t + W_c H_{t-1} + b_c)$$

$$I_t = \text{sigmoid}(U_{in} X_t + W_{in} H_{t-1} + b_i)$$

- input gate는 ① 현재 시점의 데이터  $X_t$ 와 과거의 은닉층 값  $H_{t-1}$ 에 각각의 가중치  $W_{in}$ ,  $U_{in}$  곱하고 더한 결과에 sigmoid 함수를 적용하여, 어떤 정보를 업데이트 할 지 결정하고 ( $I_t$ )  
② 현재 시점의 데이터  $X_t$ 와 과거 은닉층 값  $H_{t-1}$ 에 각각의 가중치  $W_c$ ,  $U_c$  곱하여 더한 후  $\tanh$  함수를 적용하여 현재 시점의 새로운 정보를 생성함 ( $\tilde{C}_t$ )
- 즉 현 시점에서 실제로 갖고 있는 정보가 얼마나 중요한지를 반영하여 cell에 기록함



# LSTM 구조 - cell state

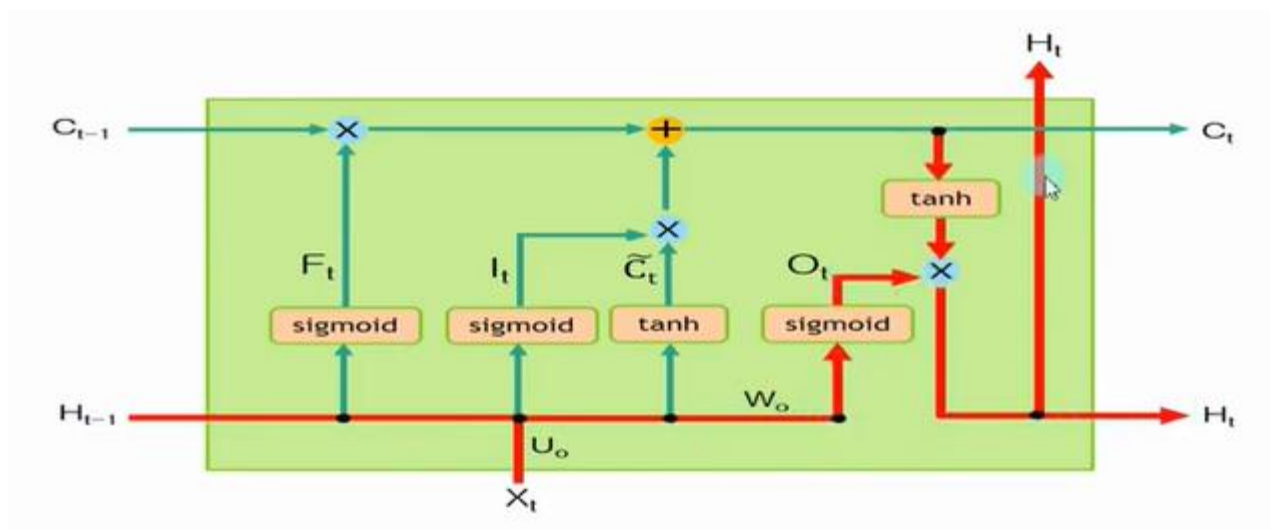


$$C_t = F_t C_{t-1} + I_t \tilde{C}_t$$

- cell state는 forget gate 출력 값  $F_t$ , input gate 출력  $I_t$ ,  $\tilde{C}_t$  값을 이용하여 memory cell에 저장하는 단계임
- 즉 과거의 정보를 forget gate 에서 계산된 만큼 잊고(또는 기억하고), 현 시점의 정보 값에 입력 게이트의 중요도 만큼 곱해준 것을 더해서 현재 시점 기준의 memory cell 값을 계산.  
※ 곱하기 표시는 모두 pointwise operation 을 나타냄



# LSTM – output gate

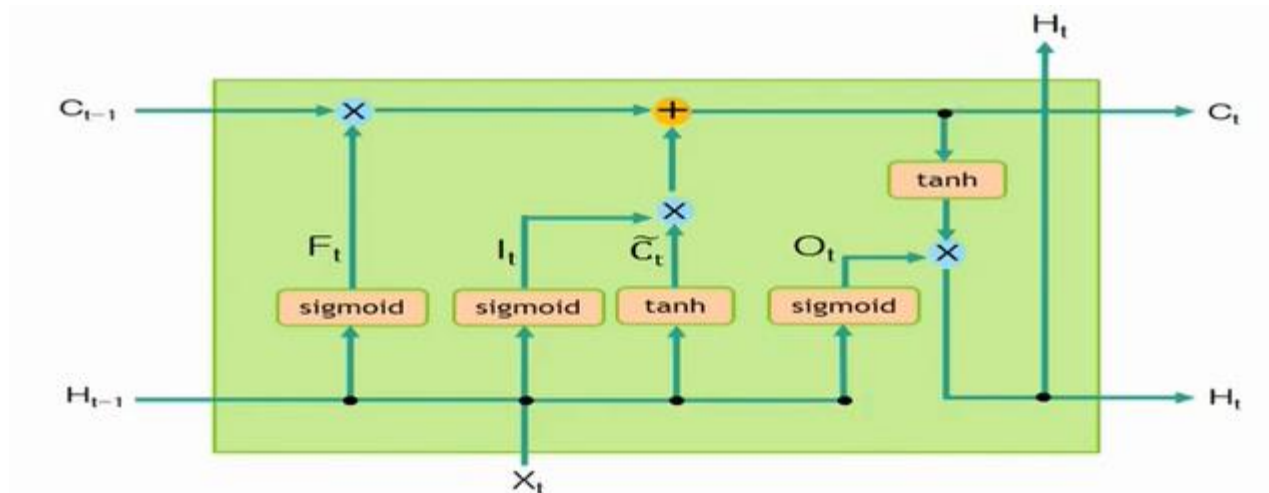


$$H_t = O_t * \tanh(C_t)$$

$$O_t = \text{sigmoid}(U_o X_t + W_o H_{t-1} + b_o)$$

- output gate는 forget gate와 input gate에 의해서 변경된 현재 시점의 memory cell state ( $C_t$ ) 값을, 얼마나 빼내서 다음 레이어로 전달할지 결정하는 단계
- 이때 현재 시점의 LSTM 출력 값  $H_t = O_t * \tanh(C_t)$  수식에서의 곱하기 표시는 pointwise operation 을 나타냄

# LSTM summary

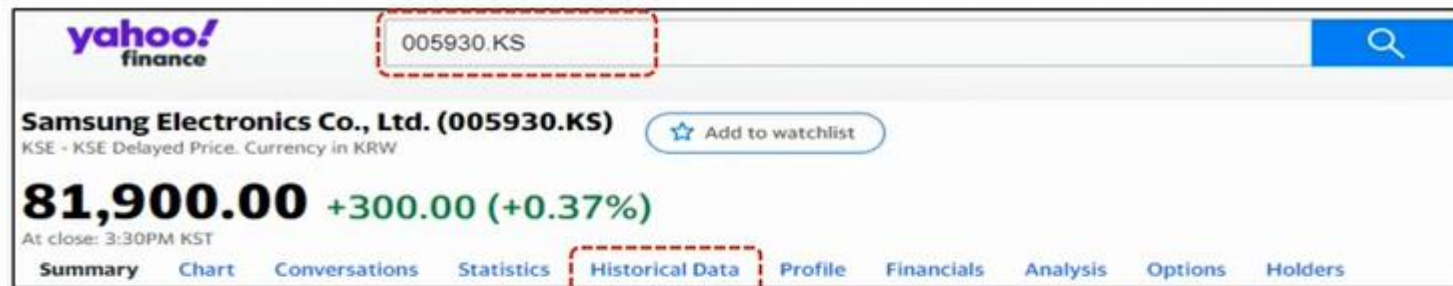


계층	수식		학습 파라미터
LSTM 계층	forget gate	$F_t = \text{sigmoid}(U_f X_t + W_f H_{t-1} + b_f)$	$U_f, W_f, b_f$
	input gate	$\tilde{C}_t = \tanh(U_c X_t + W_c H_{t-1} + b_c)$	$U_c, W_c, b_c$
		$I_t = \text{sigmoid}(U_{in} X_t + W_{in} H_{t-1} + b_i)$	$U_{in}, W_{in}, b_i$
	cell state	$C_t = F_t C_{t-1} + I_t \tilde{C}_t$	—
	output gate	$H_t = O_t * \tanh(C_t)$	$U_o, W_o, b_o$
		$O_t = \text{sigmoid}(U_o X_t + W_o H_{t-1} + b_o)$	
출력 계층	$y_t = \text{activation\_function}(V_{out} H_t + b_{out})$		$V_{out}, b_{out}$

## 4. LSTM 활용한 삼성전자 주가 예측

### ❖ 삼성전자 주식가격(2000-01-04 ~ 2021.06.18)

- 삼성전자 주가 데이터 yahoo finance에서 csv 파일로 다운로드 할 수 있음
- Csv 파일에서 7개의 column(Date, Open, High, Low, Close, Adj Close, Volume) 만 존재하나, 예측의 정확도를 높이기 위해 3일 이동평균선(3MA), 5일 이동평균선(5MA) 데이터를 추가함



005930.KS.csv 다운로드

							새로 추가한 column	
Date	Open	High	Low	Close	Adj Close	Volume	3MA	5MA
2000-01-04	6000.0	6110.0	5660.0	6110.0	4740.119629	74195000.0	NaN	NaN
2000-01-05	5800.0	6060.0	5520.0	5580.0	4328.947754	74680000.0	NaN	NaN
2000-01-06	5750.0	5780.0	5580.0	5620.0	4359.979492	54390000.0	4476.348958	NaN
2000-01-07	5560.0	5670.0	5360.0	5540.0	4297.916992	40305000.0	4328.948079	NaN
2000-01-10	5600.0	5770.0	5580.0	5770.0	4476.349121	46880000.0	4378.081868	4440.662598

# 개발과정 - 시계열 데이터 분석 및 예측



# 데이터 로드 및 분포 확인

## # LSTM 기반의 삼성전자 주가 예측 예제

```
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
```

- 데이터 불러오기

yahoo finance 에서 데이터 다운로드 후 3일(3MA), 5일(5MA) 가격이평선 추가

```
raw_df = pd.read_csv('005930.KS_3MA_5MA.csv') # yahoo finance 로부터 데이터 다운로드
```

```
raw_df.head()
```

예측할 값(수정 종가. Adj Close)

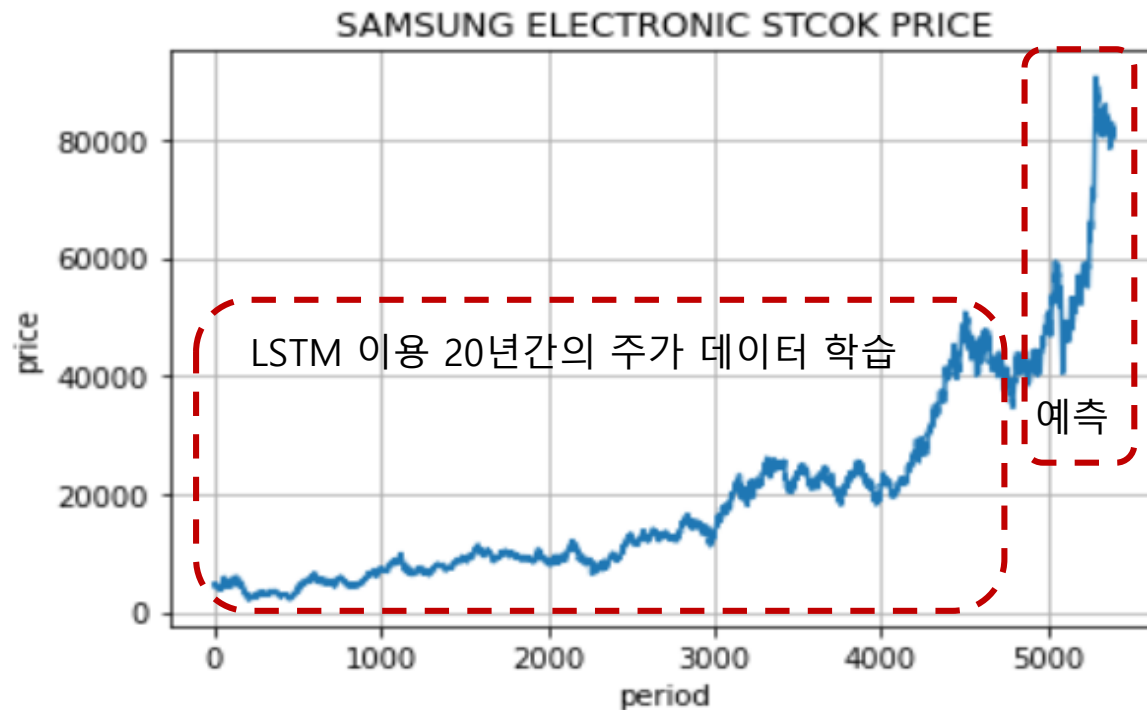
	Date	Open	High	Low	Close	Adj Close	Volume	3MA	5MA
0	2000-01-04	6000.0	6110.0	5660.0	6110.0	4740.119629	74195000.0	NaN	NaN
1	2000-01-05	5800.0	6060.0	5520.0	5580.0	4328.947754	74680000.0	NaN	NaN
2	2000-01-06	5750.0	5780.0	5580.0	5620.0	4359.979492	54390000.0	4476.348958	NaN
3	2000-01-07	5560.0	5670.0	5360.0	5540.0	4297.916992	40305000.0	4328.948079	NaN
4	2000-01-10	5600.0	5770.0	5580.0	5770.0	4476.349121	46880000.0	4378.081868	4440.662598

# 데이터 로드 및 분포 확인

```
plt.title('SAMSUNG ELECTRONIC STCOK PRICE')
plt.ylabel('price')
plt.xlabel('period')
plt.grid()

plt.plot(raw_df['Adj Close'], label='Adj Close')

plt.show()
```





# 데이터 전처리 - Outlier 확인

❖ 통계적으로 비정상적으로 크거나 작은 데이터인 outlier(특이 값)는 딥러닝 학습을 하기 위해서는 적절한 값으로 바꾸거나 삭제하는 등의 처리가 필요

- 판다스 describe()를 통해서 삼성전자 주가 데이터 통계를 확인해보면, 거래량을 나타내는 Volume 최소값이 0임을 알 수 있음 => 주식과 같은 금융데이터에서 Volume(거래량)값이 없는, 즉 0으로 나타나는 곳은 missing value(결측값)인 NaN으로 취급하는 것이 일반적임

데이터 전처리 (Missing Data 처리, 정규화 등)

```
# 통계정보 확인  
raw_df.describe()
```

	Open	High	Low	Close	Adj Close	Volume	3MA	5MA
count	5389.000000	5389.000000	5389.000000	5389.000000	5389.000000	5.389000e+03	5393.000000	5391.000000
mean	22801.887178	23043.991464	22559.285582	22801.909445	19920.044863	2.207838e+07	19934.224878	19925.585966
std	17736.201238	17906.827402	17572.109940	17733.296811	17322.636387	1.564053e+07	17313.297249	17292.697383
min	2540.000000	2760.000000	2420.000000	2730.000000	2117.926025	0.000000e+00	2164.473877	2179.989746
25%	9880.000000	9990.000000	9800.000000	9900.000000	7726.937988	1.185265e+07	7726.939453	7709.871631
50%	15760.000000	15940.000000	15560.000000	15800.000000	12808.596680	1.803935e+07	12802.806970	12809.314840
75%	29280.000000	29560.000000	29000.000000	29280.000000	24566.556640	2.771475e+07	24558.166020	24555.269140
max	90300.000000	96800.000000	89500.000000	91000.000000	90597.414060	1.642150e+08	90033.252600	89562.014060



# 데이터 전처리 Missing value 확인

- 결측치(missing value)는 특정 데이터가 누락된 것을 말하며, outlier와 마찬가지로 이러한 missing value를 제거하거나 적절한 값으로 대체하는 등의 처리가 필요함
- 판다스 `isnull().sum()` 통해서 삼성전자 주가 데이터의 missing value를 확인해보면, 6개의 칼럼(Open, High, Low, Close, Adj Close, Volume)에서 각각 6개 missing value가 있음을 알 수 있음 ⇒ 주식과 같은 금융데이터에서 NaN으로 표시되는 missing value는 평균값이나 중간값 등으로 대체하지 않고 해당되는 행(row) 전체를 삭제하는 것이 일반적임

raw_df.isnull().sum()	
Date	0
Open	6
High	6
Low	6
Close	6
Adj Close	6
Volume	6
3MA	2
5MA	4
dtype: int64	



raw_df.loc[raw_df['Open'].isna()]										
	Date	Open	High	Low	Close	Adj Close	Volume	3MA	5MA	
1304	2005-01-03	NaN	NaN	NaN	NaN	NaN	NaN	6989.931152	6910.412353	
4513	2017-11-16	NaN	NaN	NaN	NaN	NaN	NaN	49382.673830	49719.997070	
4518	2017-11-23	NaN	NaN	NaN	NaN	NaN	NaN	49373.792970	49324.969730	
4542	2018-01-02	NaN	NaN	NaN	NaN	NaN	NaN	44927.763670	44190.302730	
4755	2018-11-15	NaN	NaN	NaN	NaN	NaN	NaN	40569.835940	40775.886720	
5000	2019-11-14	NaN	NaN	NaN	NaN	NaN	NaN	49695.419920	49364.432620	

# 데이터 전처리 – Outlier 및 Missing Value 처리

✓ Volume 값 0 을 NaN 으로 모두 대체

```
# Volume 값 0 을 NaN 으로 모두 대체(replace)
```

```
raw_df['Volume'] = raw_df['Volume'].replace(0, np.nan)
```

```
# 각 column에 0 개수 확인
```

```
for col in raw_df.columns:  
    missing_rows = raw_df.loc[raw_df[col]==0].shape[0]  
    print(col + ': ' + str(missing_rows))
```

```
Date: 0  
Open: 0  
High: 0  
Low: 0  
Close: 0  
Adj Close: 0  
Volume: 0  
3MA: 0  
5MA: 0
```



✓ 모든 Missing Value 삭제

```
raw_df.isnull().sum()
```

Date	0
Open	6
High	6
Low	6
Close	6
Adj Close	6
Volume	122
3MA	2
5MA	4



```
raw_df = raw_df.dropna()
```

```
raw_df.isnull().sum()
```

Date	0
Open	0
High	0
Low	0
Close	0
Adj Close	0
Volume	0
3MA	0
5MA	0

# 데이터 전처리 - 정규화

❖ 날짜를 나타내는 Data 항목을 제외한 숫자로 표현된 column에 대해서 0~1 값으로 정규화 수행

```
# 정규화 (Date 제외한 모든 수치부분 정규화)
```

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scale_cols = ['Open', 'High', 'Low', 'Close', 'Adj Close',
              '3MA', '5MA', 'Volume']
scaled_df = scaler.fit_transform(raw_df[scale_cols])
scaled_df = pd.DataFrame(scaled_df, columns=scale_cols)
print(scaled_df)
```

	Open	High	Low	...	3MA	5MA	Volume
0	0.034868	0.032008	0.036288	...	0.025192	0.025698	0.285359
1	0.037375	0.035517	0.038470	...	0.025634	0.025094	0.363715
2	0.034982	0.031689	0.036518	...	0.026163	0.025343	0.177799
3	0.034868	0.031689	0.036059	...	0.025987	0.025503	0.250704
4	0.036235	0.033177	0.037437	...	0.026163	0.026018	0.300555
...	...	...	...	...	...	...	...
5264	0.891750	0.830923	0.896647	...	0.895299	0.903144	0.064088
5265	0.892890	0.834113	0.897795	...	0.894920	0.900854	0.061199
5266	0.899727	0.841557	0.903537	...	0.897955	0.902457	0.091190
5267	0.895169	0.835177	0.898944	...	0.899472	0.902228	0.085145
5268	0.895169	0.833050	0.896647	...	0.897955	0.901083	0.090683

# 데이터 전처리 – feature column /label column 정의

- ❖ 입력 데이터 feature column, 정답 데이터 label column 정의 후 numpy로 변환하여 데이터 전처리 과정 완료 함

주가예측을 위해 3MA, 5MA, Adj Close 항목을 feature 선정

- 정답은 Adj Close 선정
- 시계열 데이터를 위한 window\_size = 40 선정

```
# 입력 파라미터 feature, label => numpy type
def make_sequene_dataset(feature, label, window_size):
    feature_list = []      # 생성될 feature list
    label_list = []       # 생성될 label list

    for i in range(len(feature)-window_size):
        feature_list.append(feature[i:i+window_size])
        label_list.append(label[i:i+window_size])

    return np.array(feature_list), np.array(label_list)
```

# feature\_df, label\_df 생성

```
feature_cols = [ '3MA', '5MA', 'Adj Close' ]
label_cols = [ 'Adj Close' ]
```

```
feature_df = pd.DataFrame(scaled_df, columns=feature_cols)
label_df = pd.DataFrame(scaled_df, columns=label_cols)
```

# DataFrame => Numpy 변환

```
feature_np = feature_df.to_numpy()
label_np = label_df.to_numpy()
```

```
print(feature_np.shape, label_np.shape)
```

```
(5269, 3) (5269, 1)
```



# 데이터 생성 – 입력 데이터 feature / 정답 데이터 label

## [1] 학습 데이터 X, Y 생성

```
window_size = 40

X, Y = make_sequene_dataset(feature_np, label_np, window_size)
print(X.shape, Y.shape)
```

①

(5229, 40, 3) (5229, 1)

```
# 입력 파라미터 feature, label => numpy type

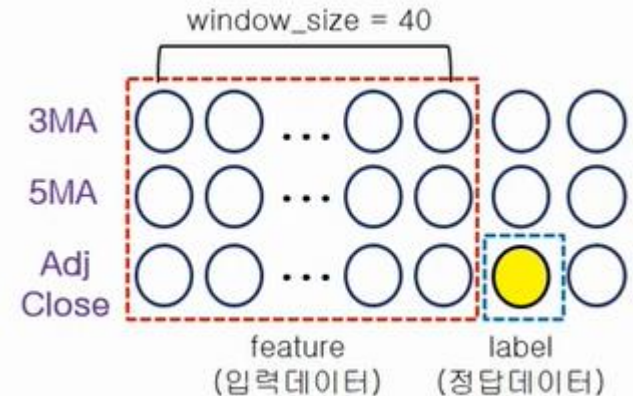
def make_sequene_dataset(feature, label, window_size):

    feature_list = []      # 생성될 feature list
    label_list = []       # 생성될 label list

    for i in range(len(feature)-window_size):

        feature_list.append(feature[i:i+window_size])
        label_list.append(label[i+window_size]) ②

    return np.array(feature_list), np.array(label_list)
```



① 넘파이로 주어지는 시계열 데이터 feature\_np, label\_np로부터, window size에 맞게 RNN 입력 데이터 X, 정답 데이터 Y 생성함, 이때 리턴되는 입력 데이터 X.shape = (batch size, time steps, input dims)

② feature[i:i+window\_size] 슬라이싱 이용하여 [ .., .., .. ] 형상으로 입력 데이터, 즉 feature를 생성함

③ feature\_list = [ .., .., .. ] 이므로 리턴 값 np.array(feature\_list)는 (batch size, time steps, input dims) 형상 가짐

# 데이터 분리 및 모델 구축

## ❖ 트레이닝 데이터/테스트 데이터 분리

### [2] 트레이닝 데이터 / 테스트 데이터 분리

```
split = -200

x_train = X[0:split]
y_train = Y[0:split]

x_test = X[split:]
y_test = Y[split:]

print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
```

```
(5029, 40, 3) (5029, 1)
(200, 40, 3) (200, 1)
```

training data

test data

5029 개

200 개

### [3] LSTM 모델 구축

```
model = Sequential()
model.add(LSTM(128,
               activation='tanh',
               input_shape=x_train[0].shape))
model.add(Dense(1, activation='linear'))
model.summary()
```

LSTM 계층에 tanh를  
활성화 함수로 가지는  
노드 수 128개

input\_shape=(40, 3)

Model: "sequential\_6"

Layer (type)	Output Shape	Param #
lstm_6 (LSTM)	(None, 128)	67584
dense_6 (Dense)	(None, 1)	129

Total params: 67,713

Trainable params: 67,713

Non-trainable params: 0

## [4] 모델 구축 및 컴파일

```
# model 생성
```

```
model = Sequential()
```

```
model.add(LSTM(128, activation='tanh', input_shape=x_train[0].shape))
```

```
model.add(Dense(1, activation='linear'))
```

```
model.compile(loss='mse', optimizer='adam', metrics=['mae'])
```

```
model.summary()
```

Model: "sequential\_6"

Layer (type)	Output Shape	Param #
=====		
lstm_6 (LSTM)	(None, 128)	67584
-----		
dense_6 (Dense)	(None, 1)	129
=====		

Total params: 67,713

Trainable params: 67,713

Non-trainable params: 0



## [5] 모델 학습 적용

```
from tensorflow.keras.callbacks import EarlyStopping

early_stop = EarlyStopping(monitor='val_loss', patience=5)

model.fit(x_train, y_train,
          validation_data=(x_test, y_test),
          epochs=100, batch_size=16,
          callbacks=[early_stop])
```

Epoch 1/100

315/315 [=====] - 3s 5ms/step - loss: 8.4391e-04 - mae: 0.0115 - val\_loss: 8.1629e-04 - val\_mae: 0.0191

Epoch 2/100

315/315 [=====] - 1s 4ms/step - loss: 9.1896e-05 - mae: 0.0062 - val\_loss: 6.6948e-04 - val\_mae: 0.0182

Epoch 3/100

315/315 [=====] - 1s 4ms/step - loss: 7.8710e-05 - mae: 0.0058 - val\_loss: 6.0811e-04 - val\_mae: 0.0168

Epoch 4/100

315/315 [=====] - 1s 4ms/step - loss: 7.3097e-05 - mae: 0.0057 - val\_loss: 4.9352e-04 - val\_mae: 0.0161

Epoch 21/100

315/315 [=====] - 1s 4ms/step - loss: 2.9038e-05 - mae: 0.0036 - val\_loss: 2.0183e-04 - val\_mae: 0.0101

Epoch 22/100

315/315 [=====] - 1s 4ms/step - loss: 2.7251e-05 - mae: 0.0035 - val\_loss: 2.8862e-04 - val\_mae: 0.0140

Epoch 23/100

315/315 [=====] - 1s 4ms/step - loss: 2.8910e-05 - mae: 0.0036 - val\_loss: 2.7316e-04 - val\_mae: 0.0123

Epoch 24/100

315/315 [=====] - 1s 4ms/step - loss: 2.8254e-05 - mae: 0.0035 - val\_loss: 2.5160e-04 - val\_mae: 0.0129

Epoch 25/100

315/315 [=====] - 1s 4ms/step - loss: 2.6528e-05 - mae: 0.0034 - val\_loss: 1.9279e-04 - val\_mae: 0.0099

<tensorflow.python.keras.callbacks.History at 0x7f5ed008bc90>

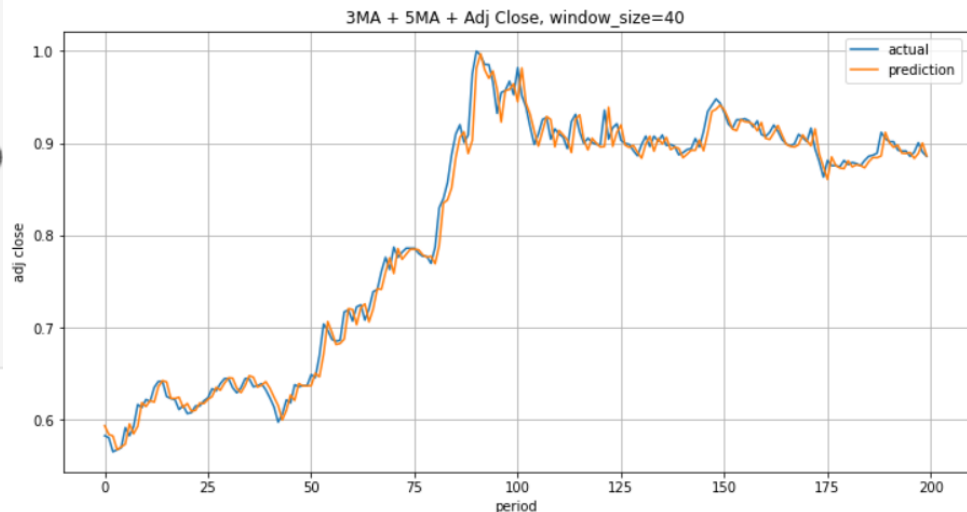
## [6] 예측을 통한 정답과의 비교

- (오차계산 MAPE 사용, 평균절대값백분율오차)

```
pred = model.predict(x_test)

plt.figure(figsize=(12, 6))
plt.title('3MA + 5MA + Adj Close, window size=40')
plt.ylabel('adj close')
plt.xlabel('period')
plt.plot(y_test, label='actual')
plt.plot(pred, label='prediction')
plt.grid()
plt.legend(loc='best')

plt.show()
```



```
# 평균절대값백분율오차계산 (MAPE)
```

```
print( np.sum(abs(y_test-pred)/y_test) / len(x_test) )
```

```
0.01224175273632353
```