

# 11장 순환 신경망

# 학습 목표

- 순환 신경망의 작동 원리를 이해한다.
- 순환 신경망을 이용하여서 사인파를 예측하는 프로그램을 작성해본다.
- 순환 신경망을 이용하여서 주가를 예측하는 프로그램을 작성해본다.





# 순차 데이터(시계열 데이터)

- 순차데이터란, **순서가 있는 데이터**이다. **시간적인 순서**도 가능하고 **공간적인 순서**도 가능하다.
- 주식 가격이나 텍스트 데이터, 오디오 데이터

가

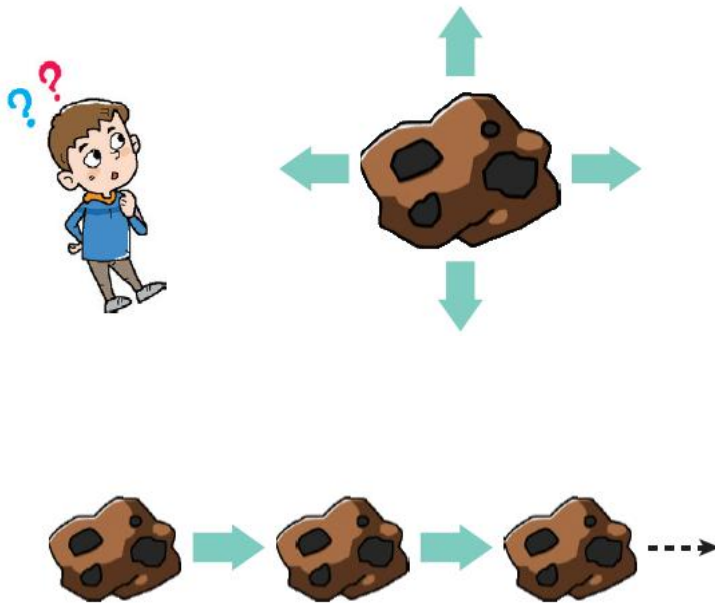




# 순차 데이터

가

- 순차 데이터를 처리하여 정확한 예측을 하려면, 과거의 데이터를 어느 정도 기억하고 있어야 한다.







# 표준 신경망 vs 순환 신경망

- 우리가 앞에서 학습한 표준 신경망을 사용하여도 어느 정도 예측이 가능하지만, 표준 신경망은 멀리 떨어진 과거의 데이터를 잘 기억하지는 못한다.
- 표준 신경망의 구조를 순차 데이터를 잘 처리하게끔 변경하는 것이 필요하다. 이것이 바로 순환 신경망(RNN: recurrent neural network)이다.



# 순환 신경망의 응용 분야

분야	형태	최종 결과물
음성 인식		What are recurrent neural network?
감정 분석	"It is my favorite time travel sci-fi"	
자동 번역	"순환 신경망이란 무엇인가?"	"What is Recurrent Neural Network?"



# 구체적인 예

- 문장 중의 빈칸을 예측하는 문제를 생각해보자.

“난 주말이면 영화를 보기 위해 우리 동네의 \_\_\_\_\_에 간다.”

이전 단어로부터 새로운 단어를 예측한다.

- 순환 신경망의 기능
  - 가변 길이의 입력을 처리할 수 있어야 한다.
  - 장기 의존성을 추적할 수 있어야 한다.
  - 순서에 대한 정보를 유지해야 한다.
  - 시퀀스 전체의 파라미터를 공유할 수 있어야 한다.



# 순환 데이터의 이해

- 순환 데이터: 본격적으로 순환 신경망을 학습시키는 데 사용되는 데이터



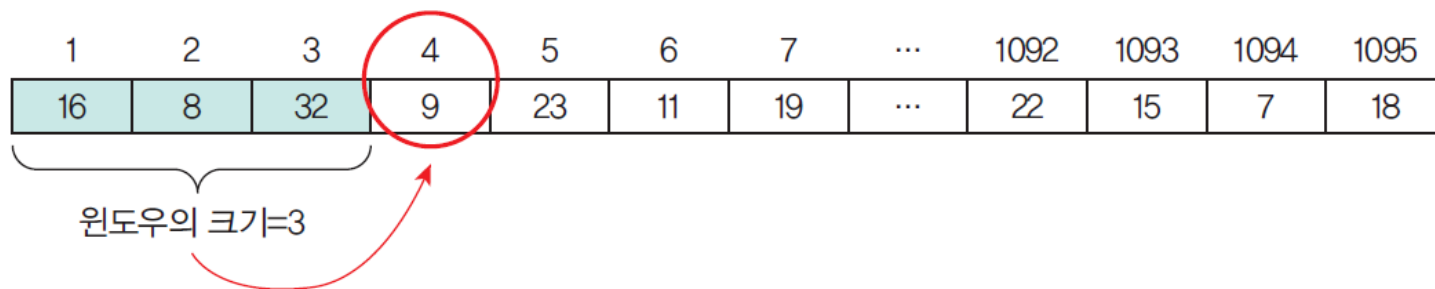
1	2	3	4	5	6	7	...	1092	1093	1094	1095
16	8	32	9	23	11	19	...	22	15	7	18





# 순환 데이터

- 순환 신경망을 학습시키려면 위의 데이터를 일정한 길이로 잘라서 여러 개의 훈련 샘플을 만들어야 한다

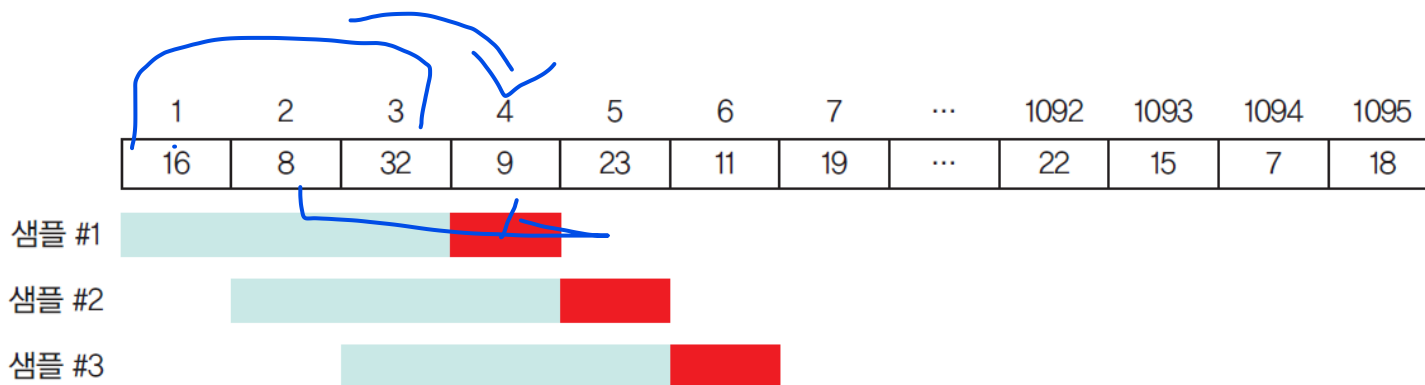




# 순환 데이터

- 전체 데이터를 다음과 같이, 크기가 3인 샘플과 정답으로 분리하게 된다

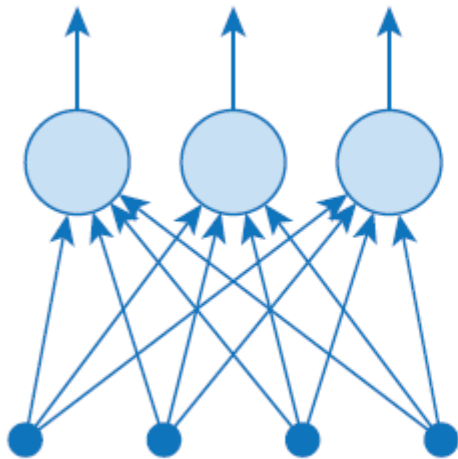
샘플 번호	x(입력)	y(정답)
1	[16, 8, 32]	[9]
2	[8, 32, 9]	[23]
3	[32, 9, 23]	[11]
...	...	...
1092	[22, 15, 7]	[18]



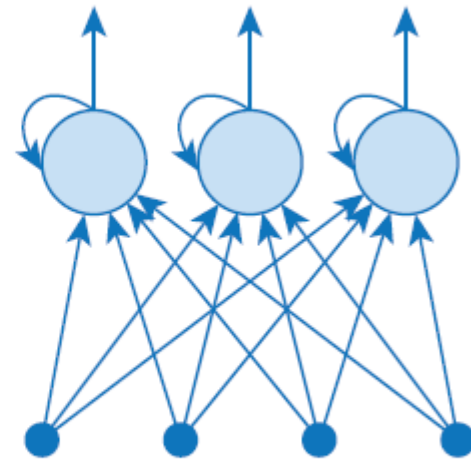


# RNN의 구조

- 피드-포워드 신경망 (feed-forward neural network) vs RNN



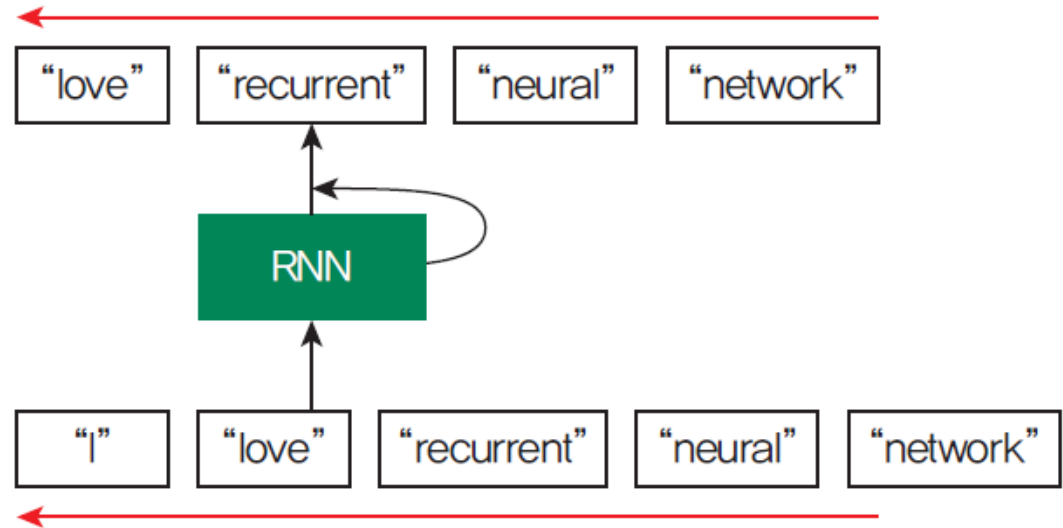
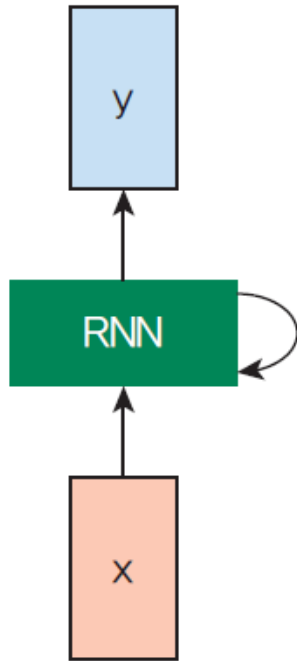
피드-포워드 신경망



순환 신경망

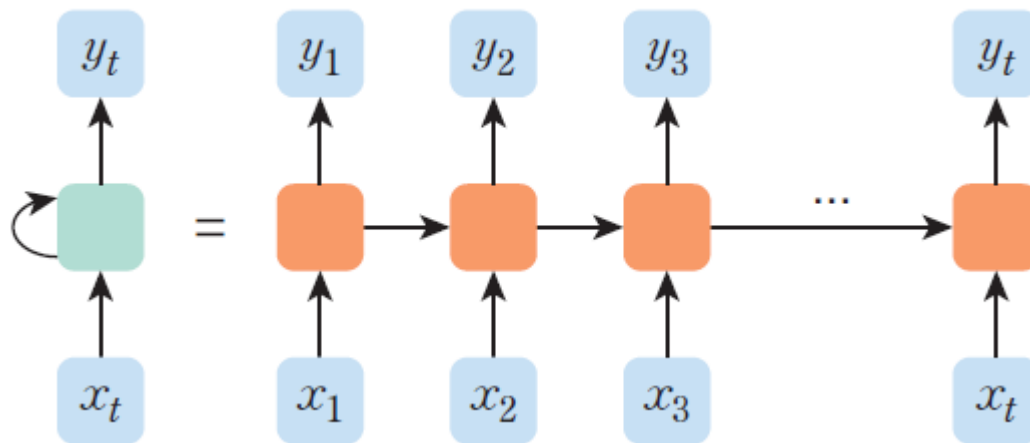


# 다음 단어를 예측하는 RNN





# 순환 신경망을 펼쳐 놓은 그림





**NOTE** 은닉 상태, 입력 벡터, 출력 벡터

지금까지는 은닉층이란 용어를 사용했지만, 순환 신경망에서는 은닉 상태라는 용어를 주로 사용한다. 또 입력 층, 출력층보다는 입력 벡터, 출력 벡터라고 한다. 만약 주가 예측 신경망에서 지난 3일치의 데이터를 가지고, 다음 날짜의 주가를 예측한다면, 입력 벡터의 크기는 3일 것이다.



# RNN의 동작

- 활성화 함수로는 tanh 함수를 주로 사용한다.

새로운 은닉 상태

이전 은닉 상태

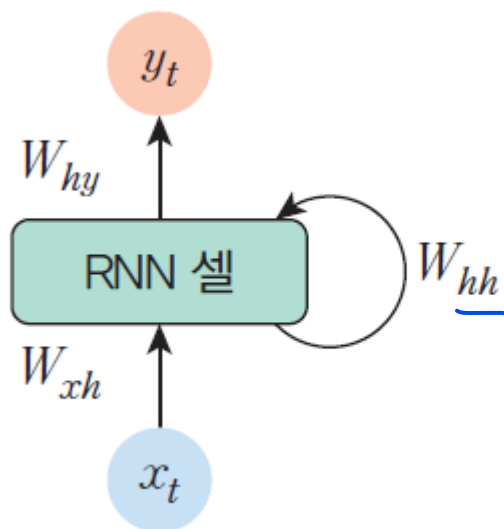
$$h_t = f_W(h_{t-1}, x_t)$$

가중치  $W$ 를 가지는 함수

시간  $t$ 에서의 입력 벡터



# RNN의 동작



- 입력 벡터:  $x_t$
- 출력 벡터:  $y_t = f(W_{hy}h_t)$
- 은닉 상태:  $h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1})$





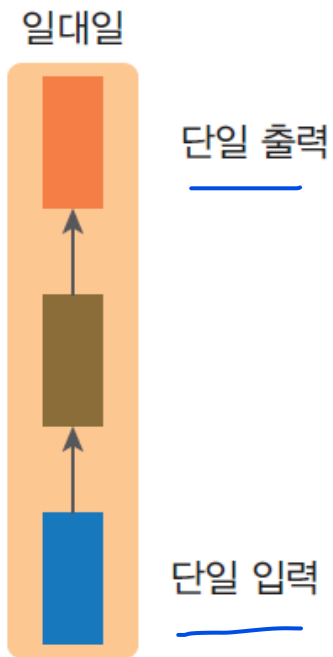
# RNN의 유형

- 일대일 (One to One)
- 일대다 (One to Many)
- 다대일 (Many to One)
- 다대다 (Many to Many)



# 일대일(One to One)

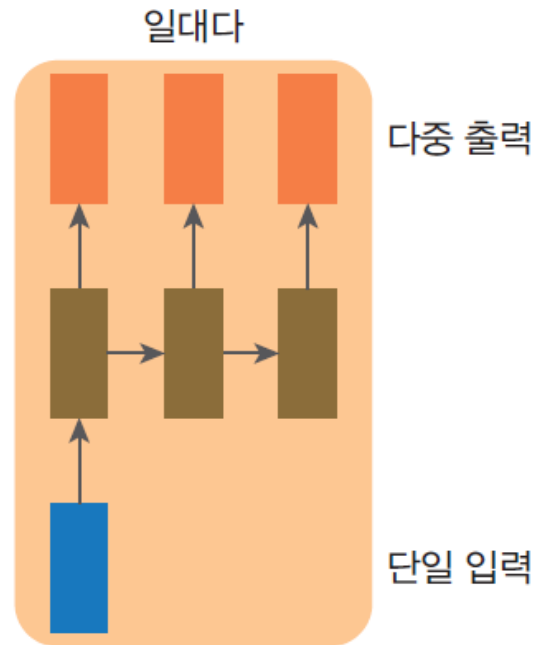
- 단일 입력과 단일 출력이 있는 가장 일반적인 신경망이다. 많은 머신 러닝 문제에 사용된다. 이것을 보통 “Vanilla Neural Network”이라고 한다.





# 일대다(One to Many)

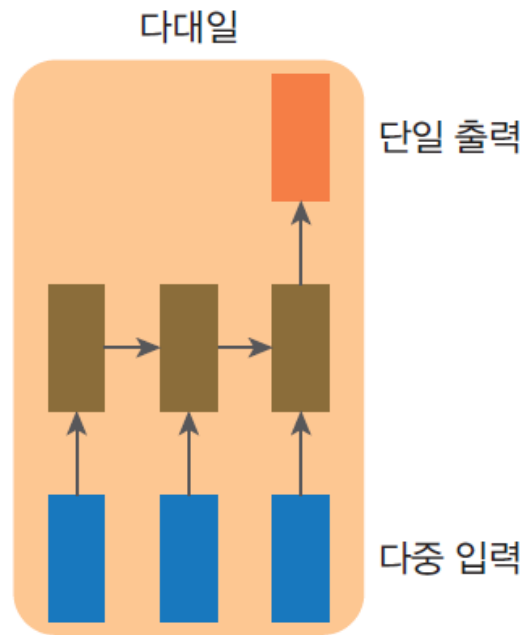
- 하나의 입력을 받아서 많은 수의 출력을 한다. 가장 좋은 예가 이미지 캡션을 생성하는 RNN이다. 하나의 이미지가 입력되면 이미지를 가장 잘 설명하는 캡션들이 생성된다.





# 다대일(Many to One)

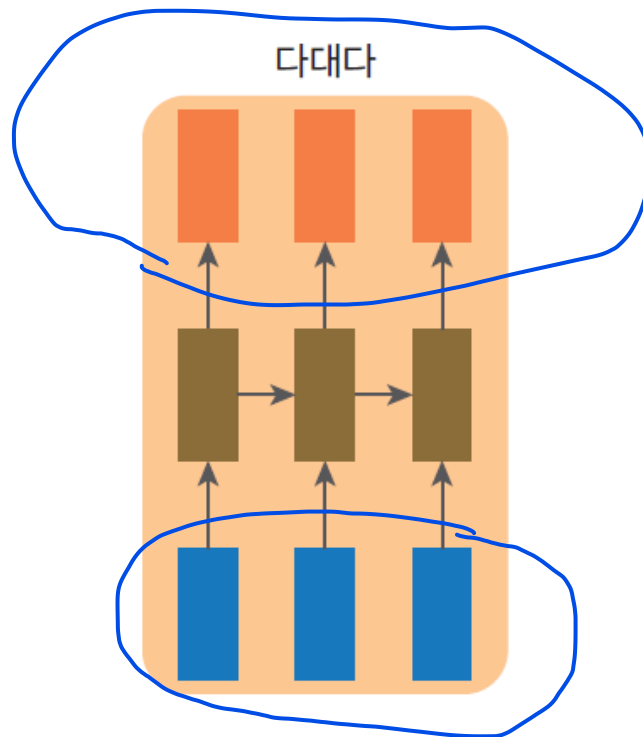
- 이 RNN은 일련의 입력을 받아 단일 출력을 생성한다. 예를 들어서 감정 (Sentiment) 분석 신경망이 이 타입에 속한다. **감성 분석 신경망**은 주어진 문장들이 긍정적 또는 부정적 감정인지를 분류한다.





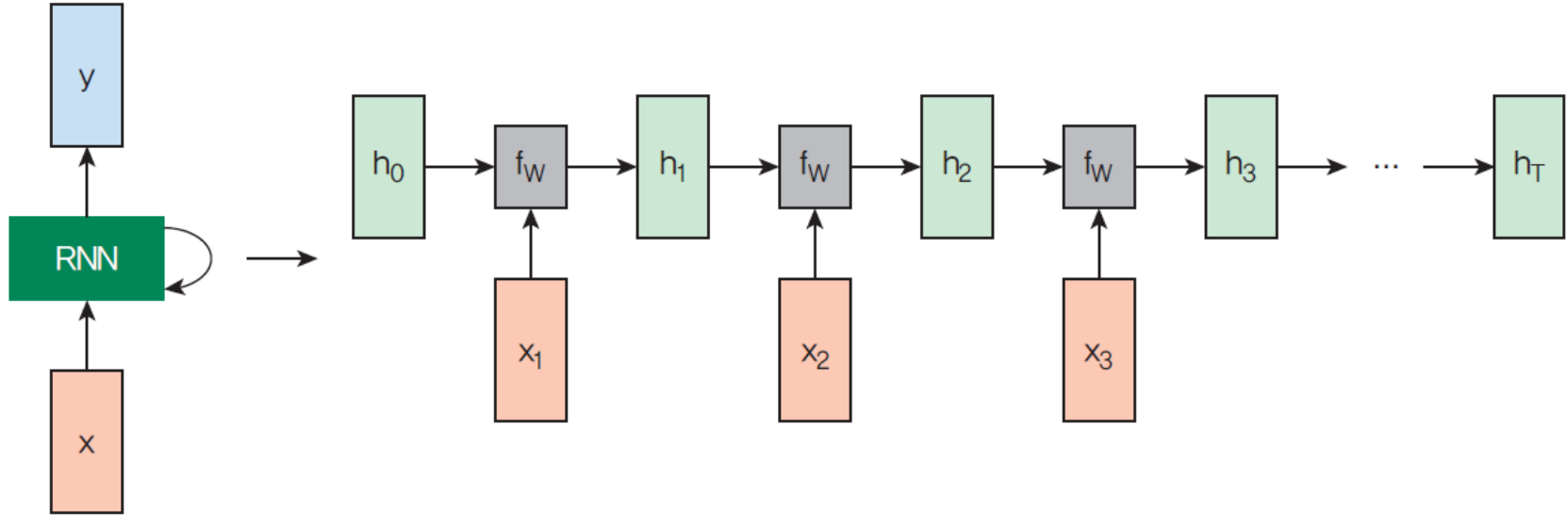
# 다대다(Many to Many)

- 이 RNN은 많은 수의 입력을 받아 많은 수의 출력을 생성한다. 기계 번역  
이 가장 좋은 예이다. 기계 번역에서는 단어들을 다른 단어들로 계속 출력  
하게 된다.





# RNN의 순방향 패스

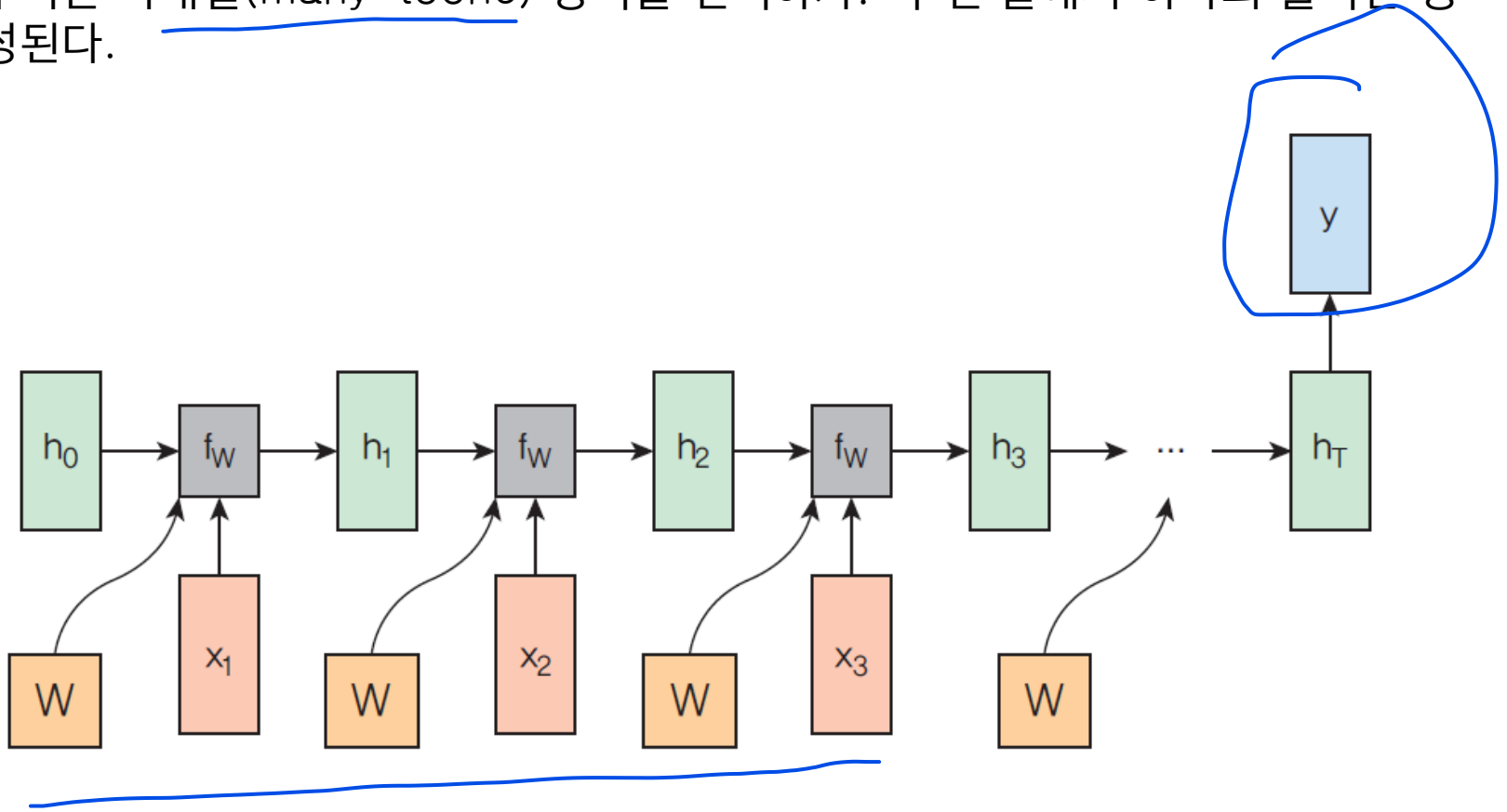


$$h_1 = f(h_0, x_1)$$



# RNN의 순방향 패스

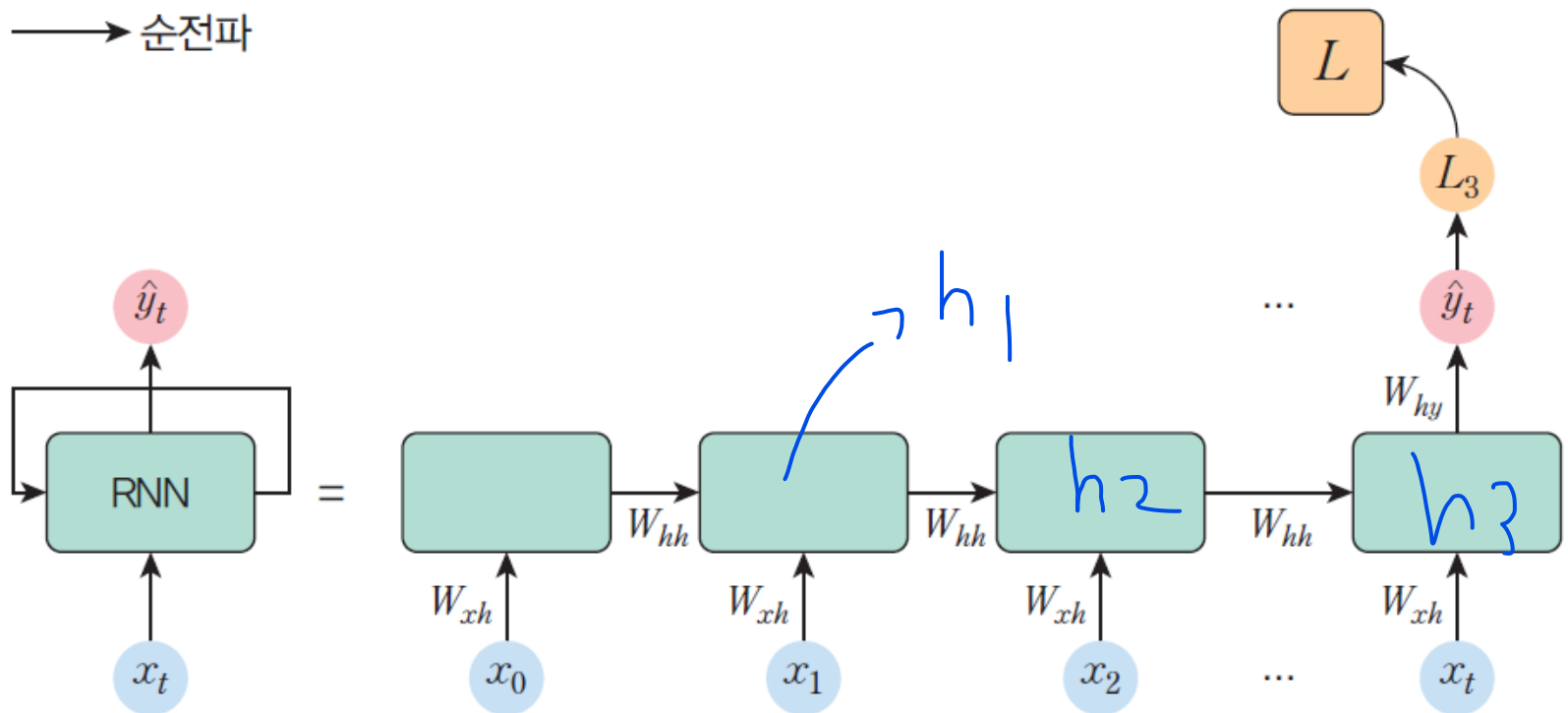
- 우리는 다대일(many-to-one) 방식을 선택하자. 즉 맨 끝에서 하나의 출력만 생성된다.





# RNN의 순방향 패스

→ 순전파







# RNN의 순방향 패스 알고리즘

```
hidden_state = [ 0, 0, 0, 0 ]
```

```
sentence = [ "I", "love", "recurrent", "neural" ]
```

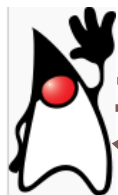
```
for word in sentence:
```

```
    hidden_state = np.tanh(np.dot(Wxh, word) + np.dot(Whh, hidden_state))
```

```
    prediction = f(np.dot(Wxh, word) + np.dot(Why, hidden_state))
```

```
print(prediction) # "network"이 되어야 한다.
```

가상적인 알고리즘



# 케라스에서의 RNN

```
inputs = np.random.random([32, 10, 8]).astype(np.float32)
```

```
simple_rnn = tf.keras.layers.SimpleRNN(4) # 4개의 셀
```

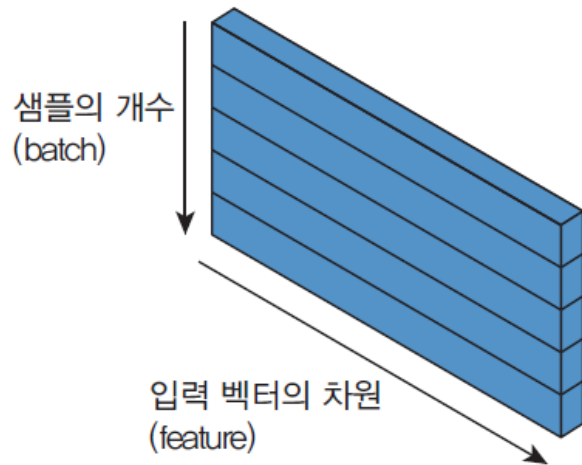
```
output = simple_rnn(inputs) # output은 최종 은닉 상태로 `[32, 4]` 형상이다.
```

- SimpleRNN(4)와 같이 호출하면 셀이 4개인 RNN 레이어가 만들어진다. SimpleRNN의 입력은 [batch, timesteps, feature]과 같은 형상을 가지는 3차원 텐서라고 간주된다.
- 위의 코드에서는 [32, 10, 8]이다. 즉 32개의 샘플이고 각 샘플당 10개의 시계열 데이터가 있다.
- 하나의 데이터는 8개의 실수로 이루어진다

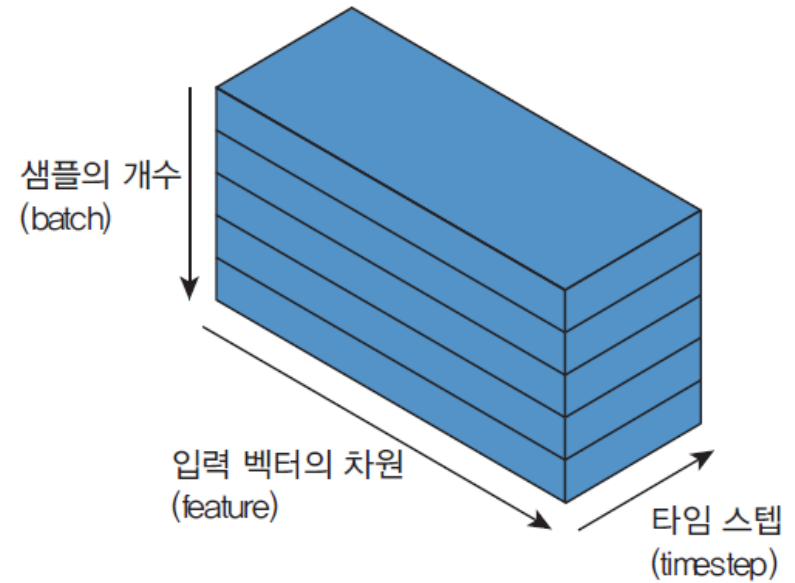


# RNN에서의 입력 형상

피드-포워드 신경망



순환 신경망





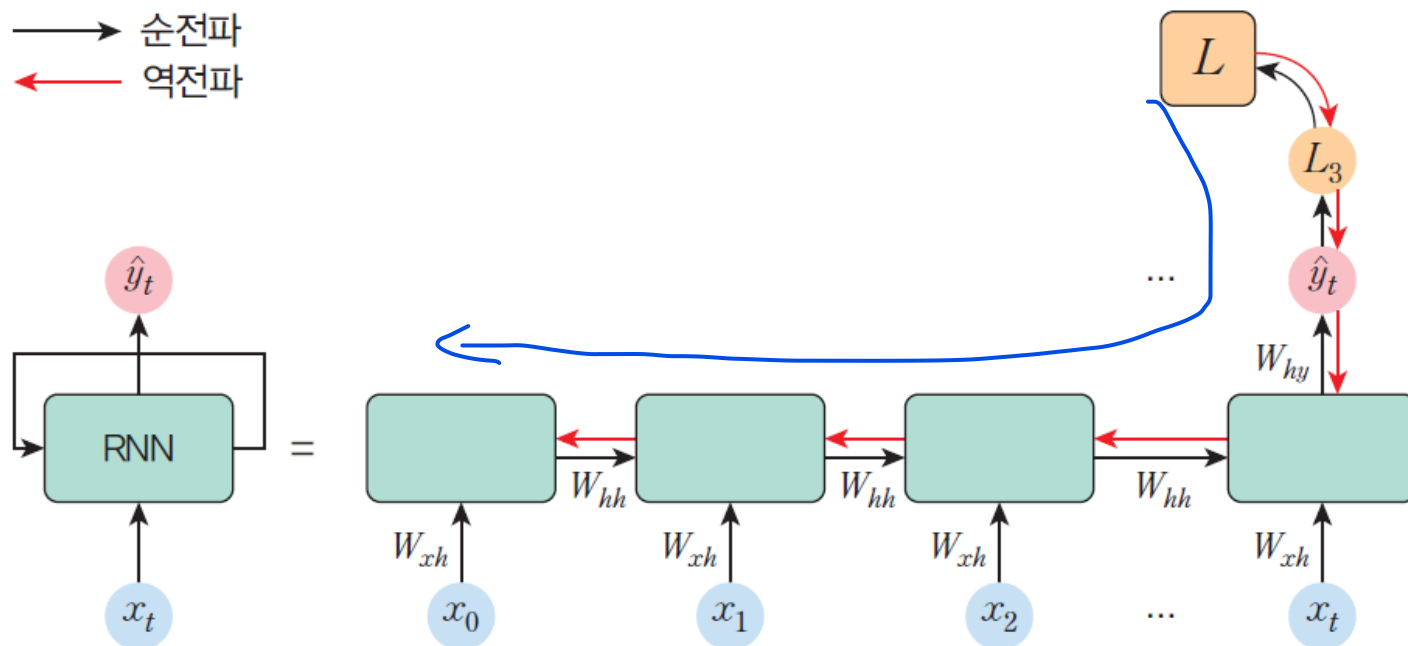
# 메모리 셀의 각 시점에서의 모든 은닉 상태값을 반환받고 싶으면

```
simple_rnn = tf.keras.layers.SimpleRNN(  
    4, return_sequences=True, return_state=True)  
    cell                                true  
whole_sequence_output, final_state = simple_rnn(inputs)  
  
# whole_sequence_output의 형상은 `[32, 10, 4]`.  
# final_state의 형상은 `[32, 4]`.
```



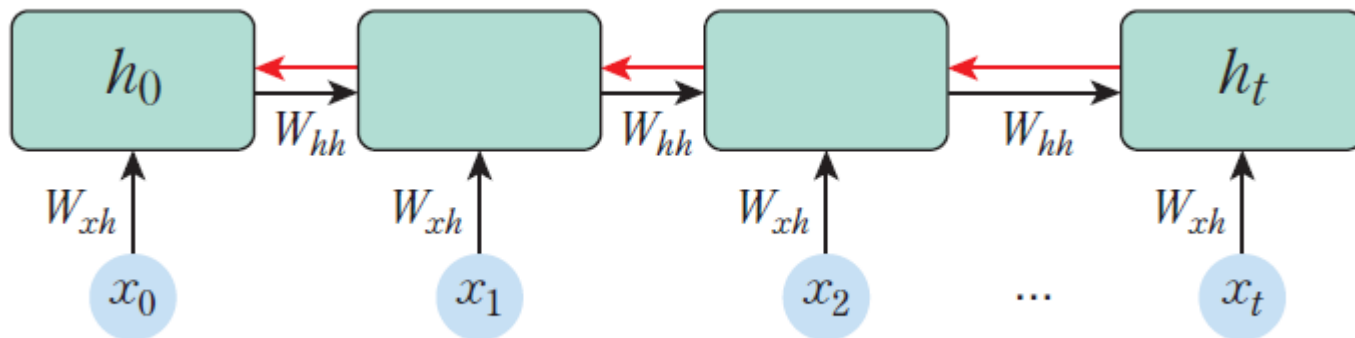
# 시간에 따른 역전파(BPTT)

- 순환 신경망에서는 계층을 통해 오류를 역전파하는 대신, 시간을 거슬러 올라가면서 **그래디언트를 역전파**한다.
- 이것을 BPTT(Backpropagation Through Time)라고 한다.





# 그래디언트 역전파 과정

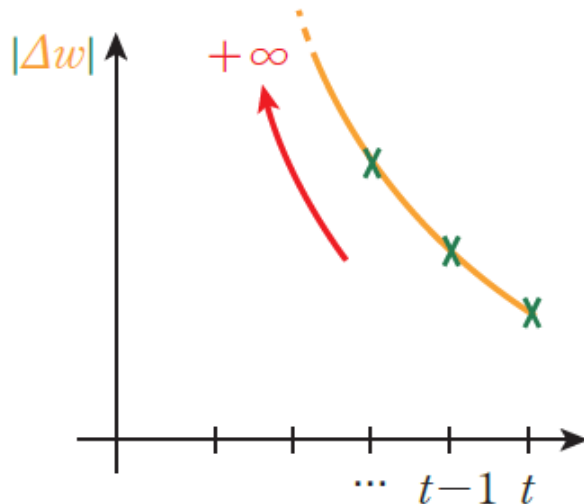




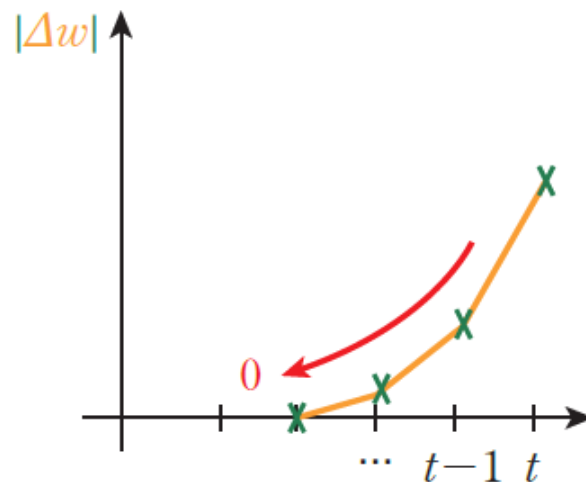
# 문제 발생

- 1보다 작은 값이 여러 번 곱해지는 경우 → **그래디언트가 점점 작아지다가 사라지게 된다.**
- 1보다 큰 값이 여러 번 곱해지는 경우 → **그래디언트가 폭발적으로 증가한다.**

그래디언트 폭증



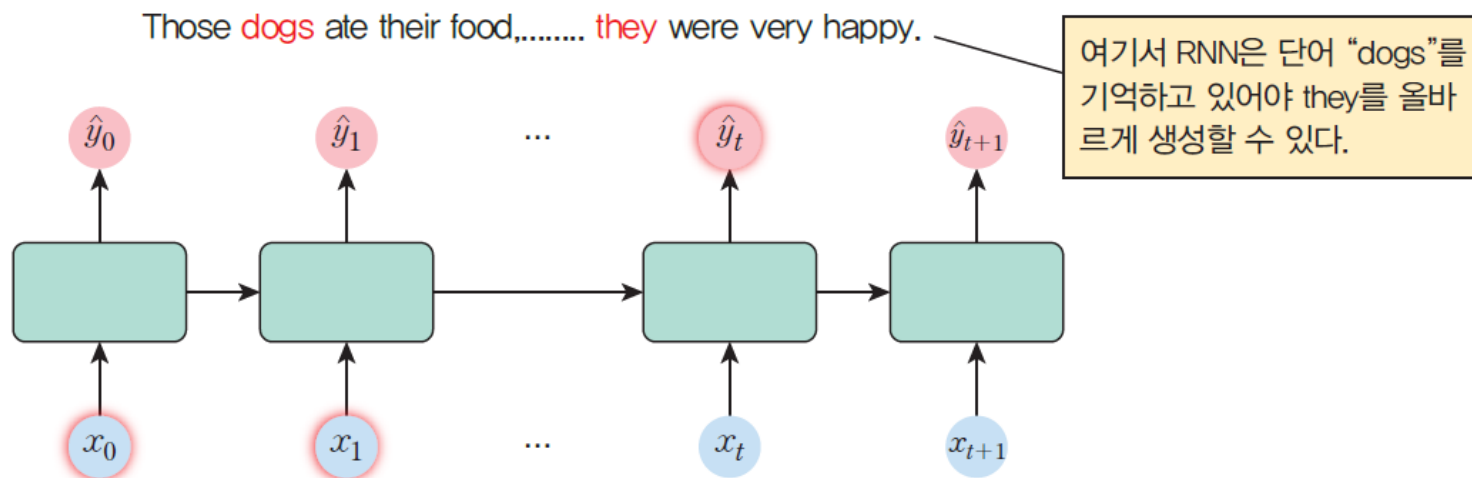
그래디언트 소실





# 그래디언트 소실 문제

- 그래디언트가 소실되면 학습이 진행되어도 먼 거리의 의존 관계는 파악하지 못하고 근거리의 의존 관계만을 중시하게 된다.







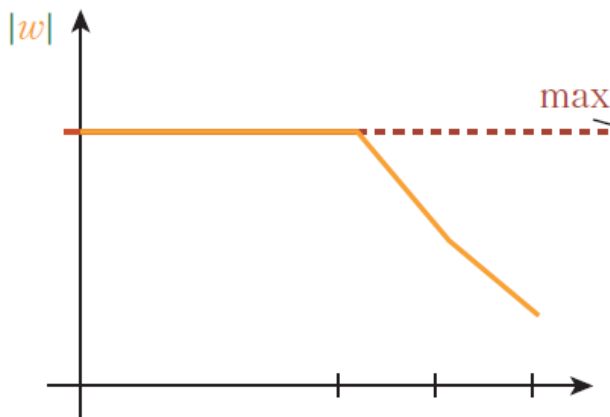
# 그래디언트 소실 방안

- 피드-포워드 신경망과 마찬가지로 활성화 함수를 ReLU로 바꾸면 도움이 된다.
- 또 다른 해결책으로는 가중치를 단위 행렬로 초기화하는 방법이 있다. 바이어스는 0으로 초기화하는 것도 도움이 된다.
- 또 하나의 해결책은 보다 복잡한 순환 유닛인 LSTM이나 GRU 같은 Gated Cell을 사용하는 것이다.
- 이들 게이트들은 어떤 정보를 지나가게 할 것인지를 제어할 수 있어서 장기적으로 기억할 수 있다.



# 그래디언트 폭증 방안

- 그래디언트 폭증은 그래디언트가 너무 커지는 것이다. 이때는 그래디언트를 일정 크기 이상 커지지 못하게 하는 방법이 많이 사용된다.



그래디언트의 절대 크기가 어느 정도 이상 커지지 못하게 하면 그래디언트 폭증 현상을 막을 수 있다.



# 예제: **사인파** 예측 프로그램 ✕

↳ time series

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

def make_sample(data, window):
    train = []
    target = []
    for i in range(len(data)-window):
        train.append(data[i:i+window])
        target.append(data[i+window])
    return np.array(train), np.array(target)

seq_data = []
for i in np.arange(0, 1000):
    seq_data += [[np.sin( np.pi * i* 0.01 )]]
X, y = make_sample(seq_data, 10)
```

# 공백 리스트 생성

# 데이터의 길이만큼 반복  
# i부터 (i+window-1) 까지를 저장  
# (i+window) 번째 요소는 정답  
# 파이썬 리스트를 넘파이로 변환

# 윈도우 크기=10



# 예제: 사인파 예측 프로그램

```
model = Sequential()  
model.add(SimpleRNN(10, activation='tanh', input_shape=(10,1)))  
model.add(Dense(1, activation='tanh'))  
model.compile(optimizer='adam', loss='mse')
```

✓ output

```
history = model.fit(X, y, epochs=100, verbose=1)  
plt.plot(history.history['loss'], label="loss")  
plt.show()
```

...

Epoch 99/100

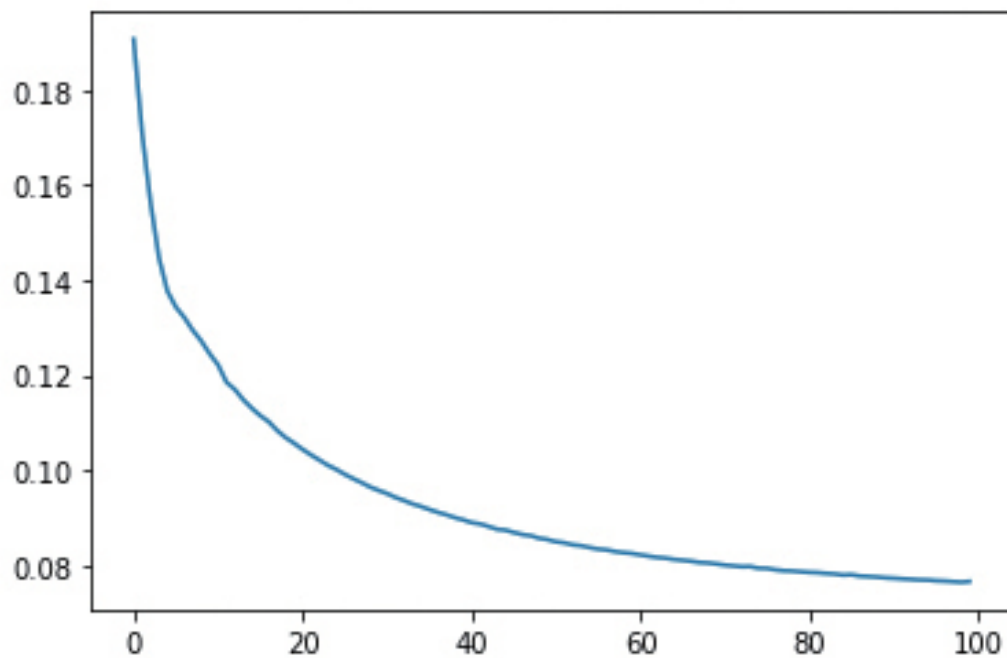
31/31 [=====] - 0s 2ms/step - loss: 5.6981e-04

Epoch 100/100

31/31 [=====] - 0s 3ms/step - loss: 5.4923e-04



# 실행 결과





# 예제: 사인파 예측 프로그램

```
seq_data = []  
for i in np.arange(0, 1000):  
    seq_data += [[np.cos( np.pi * i* 0.01 )]]  
X, y = make_sample(seq_data, 10)  
  
y_pred = model.predict(X, verbose=0)  
plt.plot(np.pi * np.arange(0, 990)*0.01, y_pred )  
plt.plot(np.pi * np.arange(0, 990)*0.01, y)  
plt.show()
```

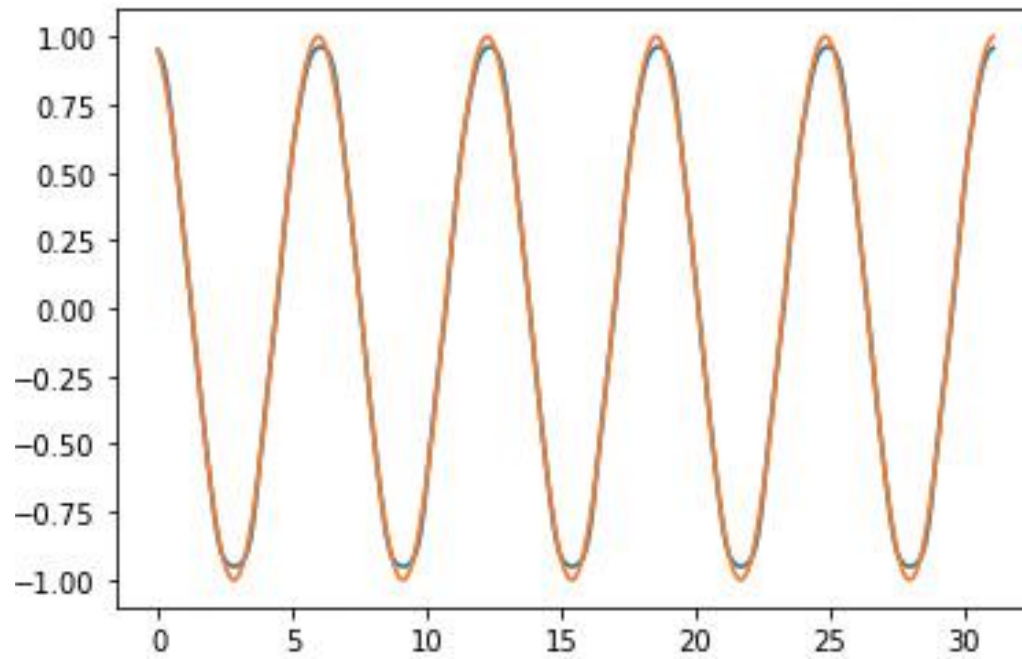
# 테스트 샘플 생성

# 윈도우 크기=10

# 테스트 예측값



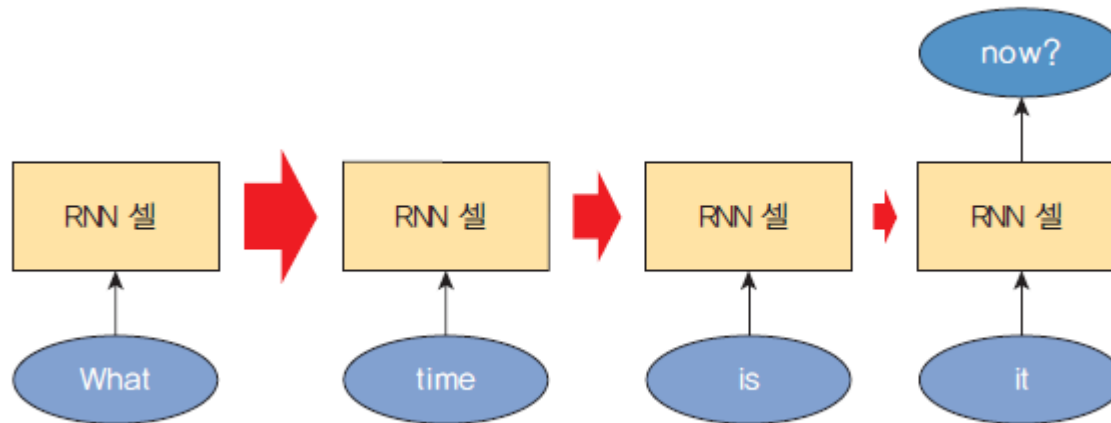
# 실행 결과





# LSTM 신경망

- RNN에서는 그래디언트 소실 현상 때문에 초반의 입력은 뒤로 갈수록 점점 손실된다.
- RNN이 초반의 입력을 기억 못 한다면 다음 단어를 엉뚱하게 예측할 것이다. 이것을 장기 의존성 문제라고 한다.

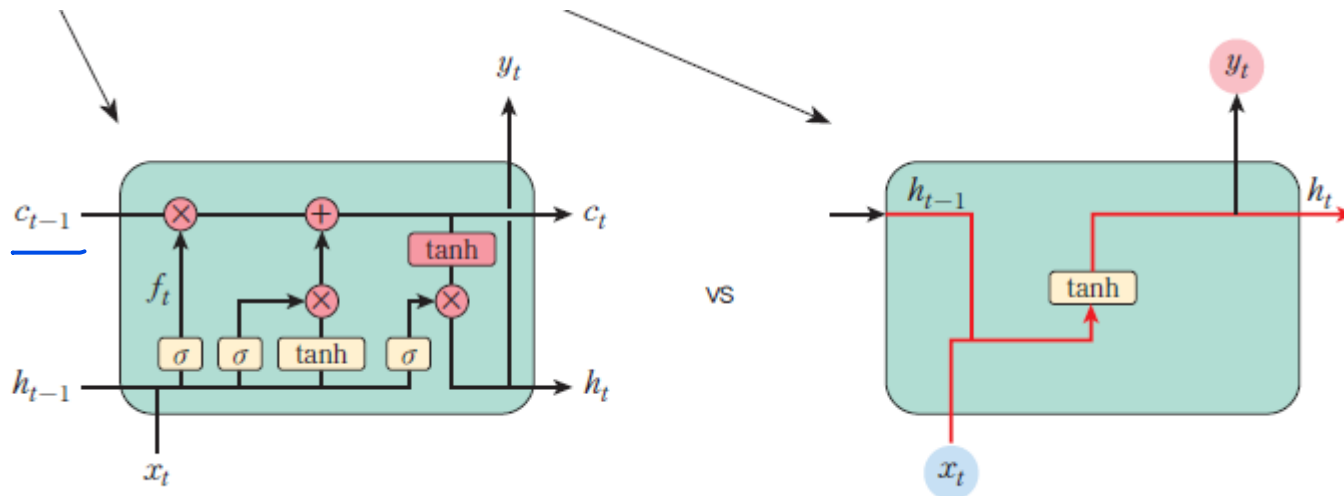






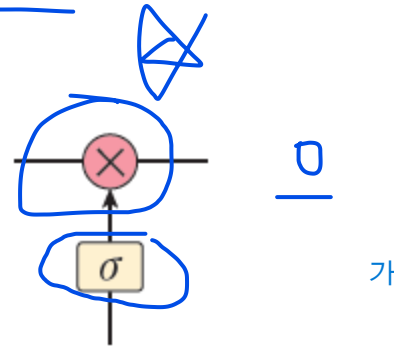
# LSTM(Long short-term memory)

- LSTM은 기존 RNN을 훈련할 때 발생할 수 있는 그래디언트 소실 문제를 해결하기 위해 개발되었다.
- LSTM 유닛은 셀, 입력 게이트, 출력 게이트, 삭제 게이트로 구성된다.
- 셀은 임의의 시점에 대한 값을 기억하고 세 개의 게이트(입력 게이트, 망각 게이트, 출력 게이트)는 셀로 들어오고 나가는 정보의 흐름을 조절한다.



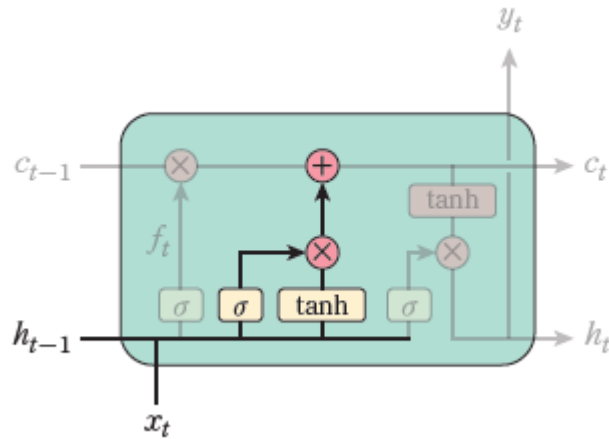


- LSTM의 주된 빌딩 블록은 게이트(gate)이다. LSTM 안의 정보들은 게이트를 통하여 추가되거나 삭제된다.



# 저장 연산은 셀 상태에 관련있는, 새로운 정보를 저장

=

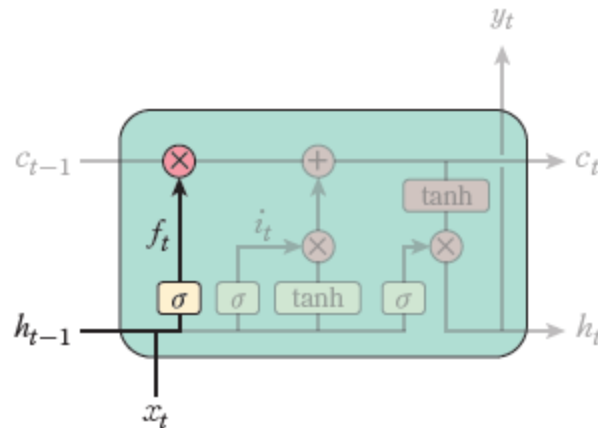


$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1})$$
$$g_t = \tanh(W_{xg}x_t + W_{hg}h_{t-1})$$



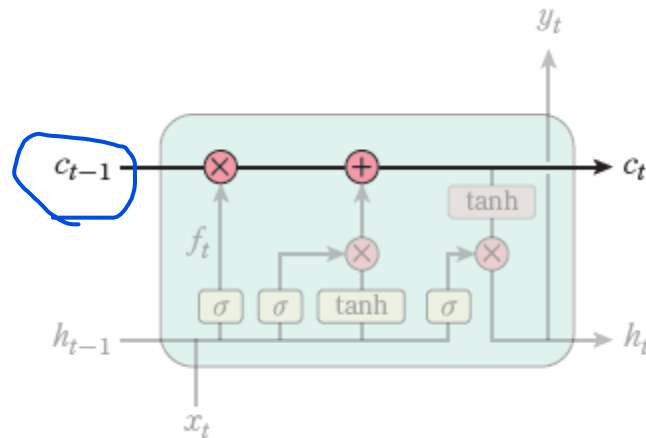
# 삭제 연산은 예전 상태 중에서 일부 관련이 없는 정보를 지우는 것

- 삭제 게이트는 기억을 삭제하기 위한 게이트이다. 현재 시점의 입력  $x_t$ 와 이전 은닉상태  $h_{t-1}$ 이 시그모이드 함수를 거친다. 0과 1 사이의 값이 출력되는데 이 값이 바로 셀 상태에서 삭제를 결정하는 값이다. 0에 가까우면 기억이 많이 삭제 될 것이고 1에 가까우면 정보들이 많이 살아남는다.



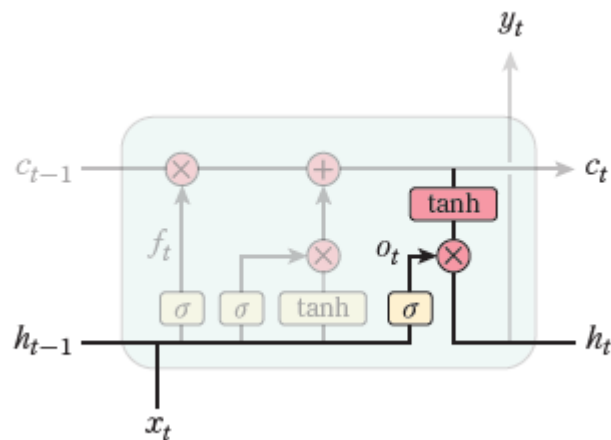
# 업데이트 연산은 선택적으로 셀 상태 값을 업데이트하는 것

- LSTM에서 장기 기억이 저장되는 곳이 셀 상태  $c_t$ 이다. 여기서는 셀 상태를 계산한다. 먼저 셀상태에 삭제 게이트를 통하여 들어온 값을 곱해서 일부 기억을 삭제한다. 여기에 입력 게이트를 통한 값을 더해준다. 즉 입력 게이트에서 선택된 기억이 추가되는 것이다.
- 삭제 게이트는 이전 시점의 입력을 얼마나 기억할지를 결정하고, 입력 게이트는 현재 시점의 입력을 얼마나 기억할지를 결정한다.



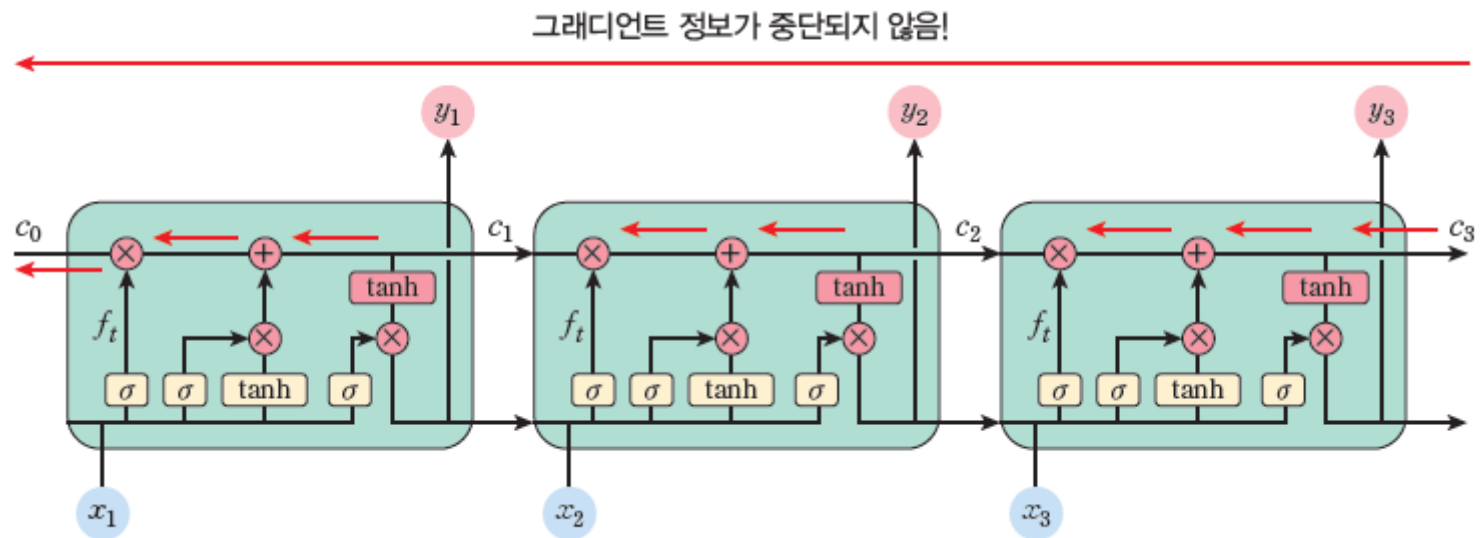
# 출력 연산

- 출력 게이트는 현재 시점의 입력값과 이전 시점의 은닉 상태에 시그모이드 함수를 적용한다. 시그모이드 함수의 결과 값은 다음 시점의 은닉 상태를 결정하는 일에 사용된다.





# 그래디언트 소실 문제 해결





# 케라스에서의 LSTM

```
inputs = tf.random.normal([32, 10, 8])
```

```
lstm = tf.keras.layers.LSTM(4) # 4개의 셀을 가진다.  
output = lstm(inputs)
```

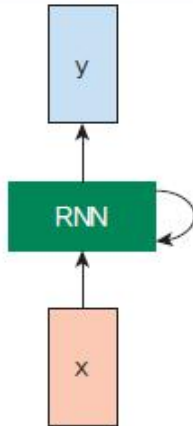
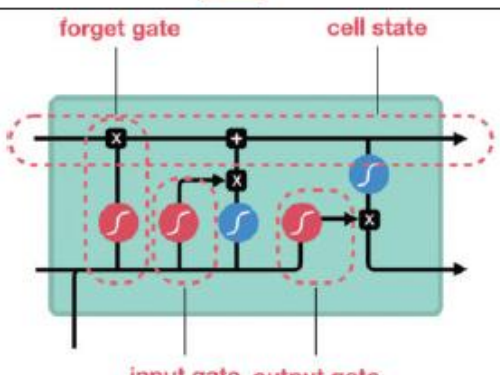
```
print(output.shape)  
print(output)
```

```
(32, 4)
```





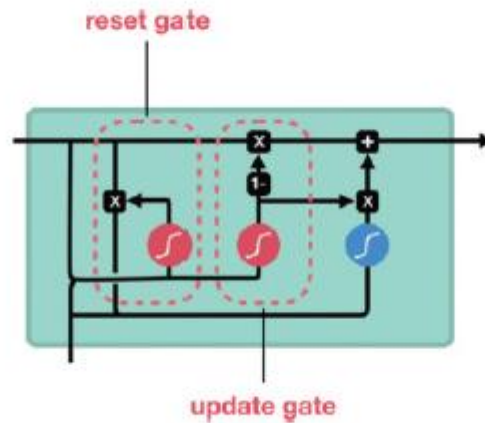
# 예제: Keras를 이용한 주가 예측

클래스	구조	설명
<code>keras.layers.SimpleRNN</code>		기본 순환 신경망이다. 앞 절에서 다룬 바 있다.
<code>keras.layers.LSTM</code>		1997년 Hochreiter & Schmidhuber 이 처음 제안한 순환 신경망 모델로서 앞 절에서 다룬 바 있다.



# 예제: Keras를 이용한 주가 예측

keras.layers.GRU



Cho et al. 등이 2014년도에서 처음 제안한 순환 신경망 모델이다. 이 책에서는 다루지 않았다.



# 예제: Keras를 이용한 주가 예측

```
import FinanceDataReader as fdr
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

samsung = fdr.DataReader('005930', '2016')
print(samsung)

openValues = samsung[['Open']]

from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range = (0, 1))
scaled = scaler.fit_transform(openValues)

TEST_SIZE = 200
train_data = scaled[:-TEST_SIZE]
test_data = scaled[-TEST_SIZE:]
```



# 예제: Keras를 이용한 주가 예측

```
def make_sample(data, window):  
    train = []  
    target = []  
    for i in range(len(data)-window):  
        train.append(data[i:i+window])  
        target.append(data[i+window])  
    return np.array(train), np.array(target)  
  
X_train, y_train = make_sample(train_data, 30)
```



# 예제: Keras를 이용한 주가 예측

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM

model = Sequential()
model.add(LSTM(16,
               input_shape=(X_train.shape[1], 1),
               activation='tanh',
               return_sequences=False)
)
model.add(Dense(1))

model.compile(optimizer = 'adam', loss = 'mean_squared_error')
model.fit(X_train, y_train, epochs = 100, batch_size = 16)
```



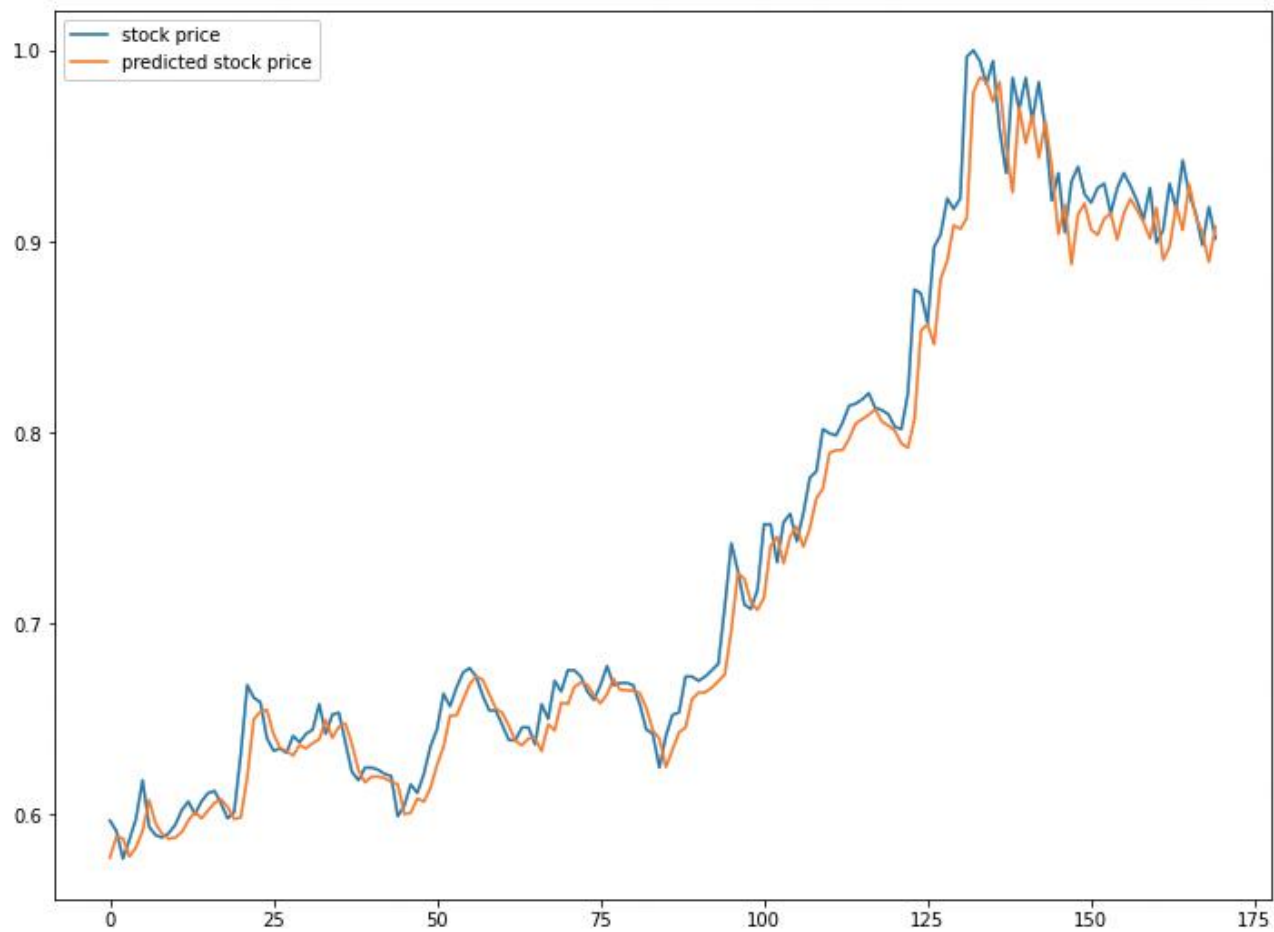
# 예제: Keras를 이용한 주가 예측

```
X_test, y_test = make_sample(test_data, 30)
pred = model.predict(X_test)

import matplotlib.pyplot as plt
plt.figure(figsize=(12, 9))
plt.plot(y_test, label='stock price')
plt.plot(pred, label='predicted stock price')
plt.legend()
plt.show()
```



# 실행 결과





# Summary

- RNN(Recurrent Neural Network)은 시계열 또는 자연어와 같은 시퀀스 데이터를 모델링하는 데 강력한 신경망이다.
- RNN은 시간을 통하여 오차를 역전파하여 학습한다. 이것을 시간에 따른 역전파(BPTT)라고 한다.
- Keras에는 세 가지 내장 RNN 레이어가 있다. `keras.layers.SimpleRNN`는 이전 시간 단계의 출력이 다음 시간단계로 공급되는 완전히 연결된 RNN이다. `keras.layers.GRU`는 Cho et al. 등에 의하여 2014년에서 처음으로 제안되었다. `keras.layers.LSTM`은 1997년 Hochreiter & Schmidhuber에 의하여 처음으로 제안되었다.
- RNN은 음악 생성이나 기계 번역 등에 사용할 수 있고 주가 예측이나 텍스트 처리, 기후 예측과 같은 분야에도 사용할 수 있다.