Deep Learning
시계열 모형

첨단 SW 융합학부 / 데이터사이언스학과

김 승 환 교수

swkim4610@inha.ac.kr

4. Neural Network

4.1 Neural Network?

4.2 XOR 문제

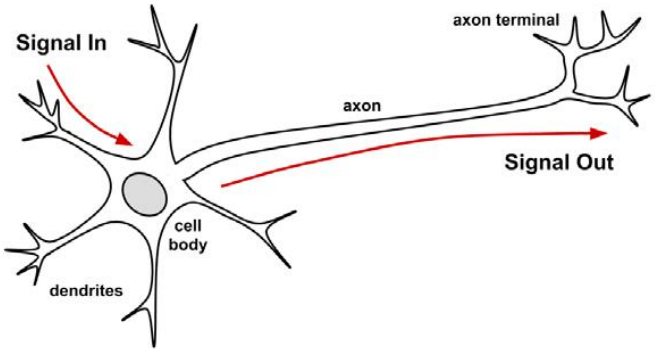
4.3 Backpropagation

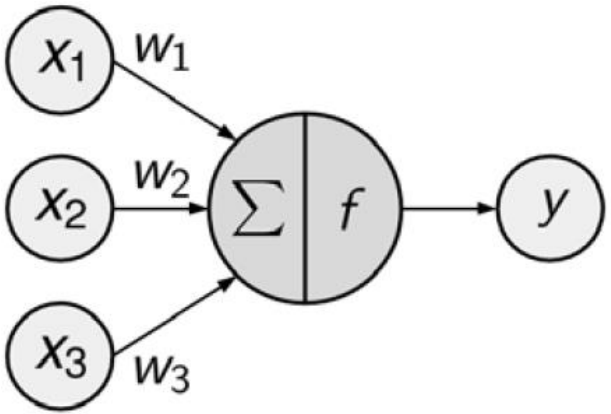
4.4 tensorflow - XOR

4.1 Neural Network

신경망 모형(Warren McCulloch and Walter Pitts, 1943)은 인간의 뇌를 수학적 모형으로 표현하여 인간처럼 판단을 수행하고자 하는 아이디어로 출발하였다.

이 아이디어는 $f(\sum w_i x_i + b)$ 의 값이 y 값이 되도록 미지수 w 값을 구하고자 하는 것이다.





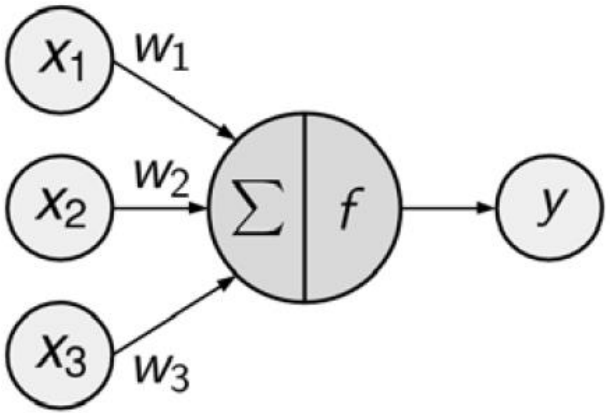
입력 X1, X2, X3 값에 각 가중치를 곱하는데 가중치가 크다는 것은 해당 자극이 신경을 활성화하는데 중요한 자극이라는 의미이다. 즉, 원하는 결과가 어떤 자극에 의해 나타나는지를 가중치로 구하는 것이다. f는 활성함수(Activation function)이라고 한다.

4.1 Neural Network

신경망 모형에서 활성 함수를 Sigmoid를 사용할 경우, 신경망 모형은 Logistic Regression(Cox, 1958) 모형이 된다. 서로 접근방법은 달랐지만 McCulloch-Pitts의 신경망 모형과 Cox의 로지스틱 회귀는 결국 같은 모형이다.

y는 종속변수이고, X1, X2, X3는 독립변수이다.
일반적으로 신경망에서는 X1~X3를 Feature, y를 hypothesis, t는 target 이라고 부른다.

※ 딥러닝 모델은 train data를 전부 외우는 것이다.
=> 즉, '딥러닝 모델'에는 '로직'이 없는 것이다.



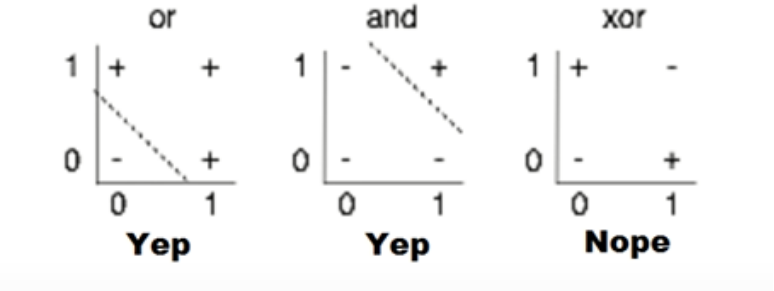
$$Y = S(w_1x_1 + w_2x_2 + w_3x_3 + b)$$

$$= \frac{1}{1+exp(-(w_1x_1+w_2x_2+w_3x_3+b))}$$

· 이에 따라, 'NNAR'에는 '시간' 개념이 없다.
ex) x_1과 x_2의 위치를 서로 바뀌어도 결과값은 바뀌지 않음.

4.1 Neural Network

초기 신경망 모형은 Linear 한 모형으로 or, and 는 풀 수 있으나 xor는 풀 수 없어 많은 사람들이 xor문제에 집중하였다. 이후, 1969년 Minsky는 Hidden Layer를 사용하는 Multiple layer perceptron을 사용해 xor문제를 풀 수 있음을 증명했는데 문제는 MLP의 가중치를 구할 수 없다는 문제에 직면했다.



이후, 1974년 Backpropagation 알고리즘이 나와 MLP의 가중치를 구할 수 있게 되었으나 그 당시 주목받지 못하다가 1986년 Hinton에 다시 동일한 내용의 논문을 발표하면서 부터 다시 주목받기 시작했다.

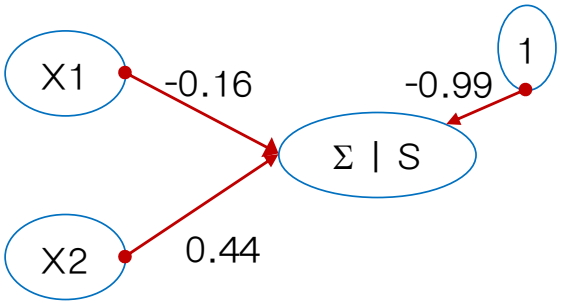
이후, 10년 정도 Neural Network이 유행했으나 많은 단점을 노출하면서 다시 시들해 졌다.

단점으로는 대표적으로 Over-fitting과 layer의 수가 커질수록 학습이 안되는 문제가 있다.

2006, 2007년에 문제에 대한 솔루션이 나오면서 신경망 모형은 다시 Deep Learning이란 이름으로 Re-Branding 하였다. 이후, 빅데이터 등 호재를 만나면서 특히, Image인식 분야에서 본격적으로 딥러닝의 시대를 열었다.

4.1 Neural Network

먼저, 임의의 W 값으로 아래 표와 같은 연산을 수행하여 error를 구한다.
이후, 이 오차를 줄이는 방향으로 W 값을 update 한다. 아래의 과정을 만족된 해가 나올 때까지 반복한다.



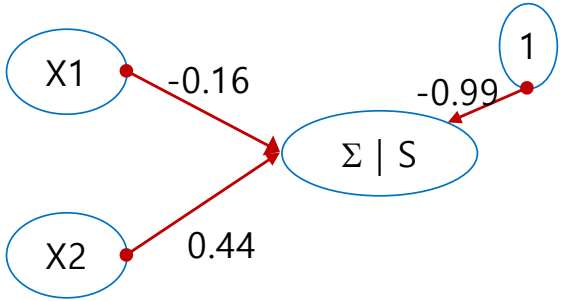
$$\begin{pmatrix} 0.12 \\ 0.73 \\ -0.61 \end{pmatrix} = \begin{pmatrix} -0.16 \\ 0.44 \\ -0.99 \end{pmatrix} - \begin{pmatrix} -0.28 \\ -0.29 \\ -0.38 \end{pmatrix}$$

$W = W - \lambda \times$ 기울기방향

x1	x2	sum	Y	T	error
0	0	0*-0.16+0*0.44-0.99=-0.99	1/(1+exp(0.99)) = 0.27	0	0.27
0	1	0*-0.16+1*0.44-0.99=-0.55	1/(1+exp(0.55)) = 0.36	1	-0.64
1	0	1*-0.16+0*0.44-0.99=-1.15	1/(1+exp(1.15)) = 0.24	1	-0.76
1	1	1*-0.16+1*0.44-0.99=-0.71	1/(1+exp(0.71)) = 0.33	1	-0.67

4.1 Neural Network

전 페이지를 복습하는 의미에서 아래의 표를 완성해보세요~



x1	x2	sum	Y	T	error
0	0				
0	1				
1	0				
1	1				

4.1 Neural Network

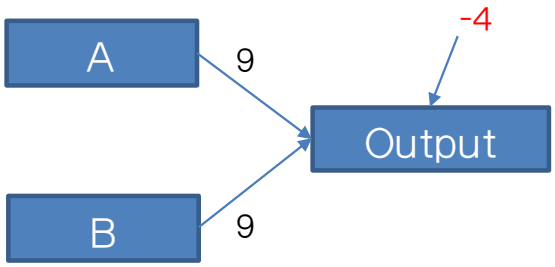
1943 년에 제안된 초기 신경망 모형은 or, and, xor 문제를 푸는 것이었다.
or, and는 해결하였으나 xor를 만족하는 해를 구하지 못했다.
이후, 1969년 Minsky는 Hidden Layer를 사용해 xor 문제를 풀 수 있음을 증명했는데 그 당시, 가중치를 구할 수 없다는 문제에 직면해 실용화는 어려웠다.

AND		
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR		
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

exclusive-OR



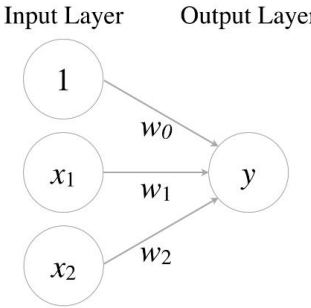
$$Y = S(w_1 x_1 + w_2 x_2 + b) > 0.5$$

$$(w_1 x_1 + w_2 x_2 + b) > 0$$

DIY 문제: OR 문제가 해결됨을 보이세요~

4.1 Neural Network

1958년 Rosenblatt는 AND/OR 문제를 해결하는 Perceptron 기계를 개발하였다.



$$y = \begin{cases} 0 & (w_0 + w_1 x_1 + w_2 x_2 \leqq 0) \\ 1 & (w_0 + w_1 x_1 + w_2 x_2 > 0) \end{cases}$$

위에서 가중치 w를 구해보자.

아래 식에서 t는 true 값이고 f(net)은 네트워크를 통해 계산된 y값이다. t-f(net) 은 오차다.

w는 아래의 식으로 t-f(net) 의 오차가 양수면 f(net) 이 커져야 하므로 가중치에 일정량을 더하고 음수면 f(net) 이 작아져야 하므로 가중치에 일정량을 뺀다.

여기서, 그리스문자 에타 η는 상수로 학습율(Learning Rate) 라고 부른다.

학습률은 w값이 목적값으로 가는 속도를 조절하는 상수역할을 하는데 에타가 크면 빨리 해로 가지만 정확한 해를 구하기 어렵고 에타가 작으면 해로 느리게 가지만 정교한 해를 구할 수 있다.

$$w_i = w_i + \eta x_i (t - f(net))$$

4.1 Neural Network

```
def f(x):
    if x > 0:
        return 1
    else:
        return 0

import numpy as np
x = np.array([[1,0,0], [1,0,1], [1,1,0], [1,1,1]])
t = np.array([0,0,0,1]) # AND
#t = np.array([0,1,1,1]) # OR
eta = 0.1
w = np.array([0.,0.,0.])

for iter in range(10):
    for i in range(len(x)): # 4가지 경우에 대해 루프
        fnet = f(np.dot(x[i], w))
        e = t[i] - fnet
        w[0] = w[0] + eta * e * x[i][0]
        w[1] = w[1] + eta * e * x[i][1]
        w[2] = w[2] + eta * e * x[i][2]
    print(w)
```

$$w_i = w_i + \eta x_i (t - f(net))$$

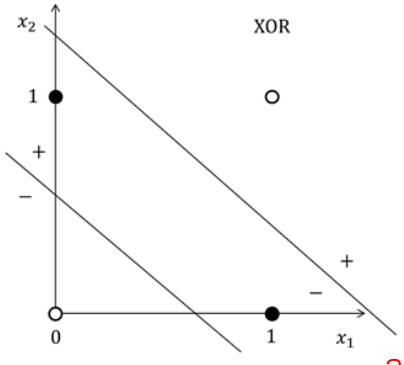
learning rate. (학습률)

· 학습률이 높으면 가중치 변화가 많이 발생함

4.1 Neural Network

학습률이 낮으면 가중치 변화가 적게 발생한다.

xor 문제는 아래의 그림처럼 h1, h2 두개 선을 사용하여 해결할 수 있다.



x_1	x_2	h_1	h_2	y
0	0	0(-)	0(-)	0
0	1	1(+)	0(-)	1
1	0	1(+)	0(-)	1
1	1	1(+)	1(+)	0

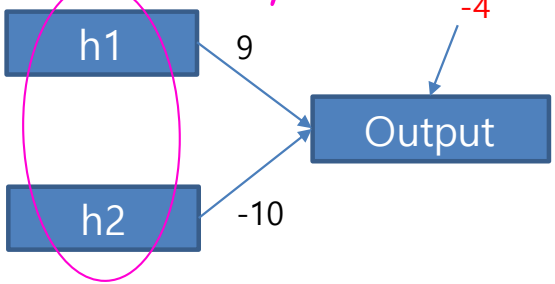
$Y = S(w_1h_1 + w_2h_2 + b) > 0.5$

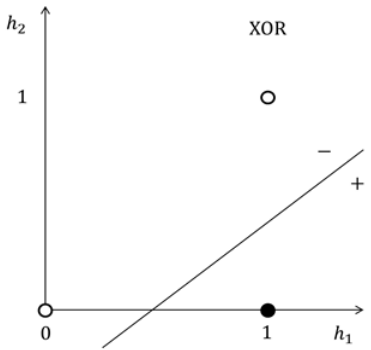
$(w_1h_1 + w_2h_2 + b) > 0$

or

and

hidden layer : '숨겨진 층'에서 실제 값을 보는 것은 아님





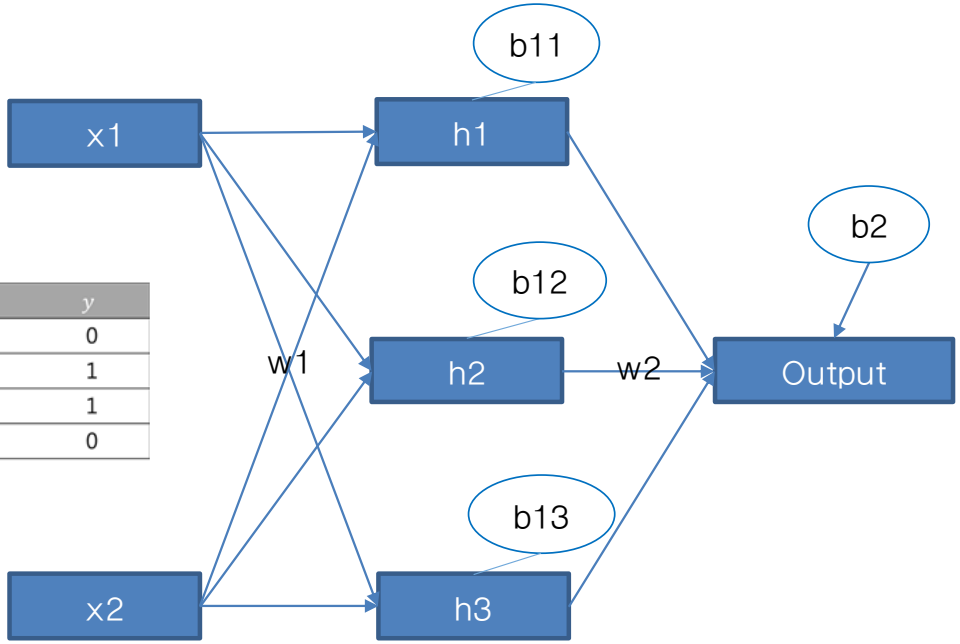
DIY 문제: XOR 문제가 해결됨을 보이세요~

4.2 Neural Network-XOR

신경망 모형 설계

- ✓ xor 문제는 입력이 두개이고 출력이 하나임
- ✓ 히든 레이어는 한 개, 노드는 3개로 가정함(히든 레이어와 노드 수는 자유롭게 결정 가능)
- ✓ w1은 총 6개의 선으로 2 by 3 행렬이고 w2는 3개의 선으로 3 by 1 행렬이다.
- ✓ b1은 1 by 3 행렬이고, b2는 1 by 1 행렬이다.
- ✓ H=x ·w1+b1 이고, output=H ·w2+b2로 계산한다.

x_1	x_2	h_1	h_2	y
0	0	0(-)	0(-)	0
0	1	1(+)	0(-)	1
1	0	1(+)	0(-)	1
1	1	1(+)	1(+)	0



4.2 Neural Network-XOR

신경망 모형 설계

- ✓ xor 문제는 입력이 두개이고 출력이 하나임
- ✓ 히든 레이어는 한 개, 노드는 3개로 가정함(히든 레이어와 노드 수는 자유롭게 결정 가능)
- ✓ w1은 총 6개의 선으로 2 by 3 행렬이고 w2는 3개의 선으로 3 by 1 행렬이다.
- ✓ b1은 1 by 3 행렬이고, b2는 1 by 1 행렬이다.
- ✓ H=x ·w1+b1 이고, output= H ·w2+b2로 계산한다.

DIY 문제: 이 신경망 모형을 그려보세요~~~

4.2 Neural Network-XOR

```
# xor.ipynb
import math
import numpy as np

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

x=np.array([[0,0], [0,1], [1,0], [1,1]])
w1=np.array([[-2,  5,  4], [ 3,  6,  3]])
b1=np.array([2, -2, -5])
w2=np.array([[-4], [ 8], [-8]])
h=Sigmoid(np.dot(x,w1)+b1)
y=Sigmoid(np.dot(h,w2))

print (y)
```

Hidden Layer가 추가됨으로써 xor문제가 풀림을 알 수 있다.
이제, xor문제에서 주어진 가중치 W, b를 구하는 것이 문제이다.
여기서, np.dot()는 행렬 곱셈을 수행하는 함수이다.

4.4 Tensorflow- XOR

XOR 룰을 Tensorflow로 코딩한 결과이다. 학습해보자.

cost 함수는 cross entropy 함수이고, 로지스틱 회귀와 마찬가지로 Gradient Descent Algorithm을 사용하여 진행한다.

optimizer='sgd' 라고 하면 learning rate를 지정할 수 없어서 sgd = tf.keras.optimizers.SGD(learning_rate=0.1)
와 같이 sgd 변수를 만들고 진행하자.

```
# XOR_NN_TF2.ipynb
import tensorflow as tf
import numpy as np
x = np.array([[0,0],[0,1],[1,0],[1,1]]).astype('float32')
y = np.array([[0],[1],[1],[0]]).astype('float32')
from tensorflow.keras import layers
model = tf.keras.Sequential()
model.add(layers.Dense(3, activation='sigmoid', input_dim=2))
model.add(layers.Dense(1, activation='sigmoid'))
sgd = tf.keras.optimizers.SGD(learning_rate=0.1)
#model.compile(optimizer='sgd', loss='binary_crossentropy',metrics=['accuracy'])
model.compile(optimizer=sgd, loss='binary_crossentropy',metrics=['accuracy'])
model.fit(x, y, epochs=10000,  batch_size=4, verbose=0)
model.evaluate(x, y)
predicted = model.predict(x)
print(predicted)
```

This model has sequential layer.

metrics: 측정방식.

8. Recurrent Neural Network

8.1 Recurrent Neural Network?

8.4 Long Short Term Memory

8.5 Stacked RNN

8.6 RNN Example(Stock Data)

8.7 RNN Example(etc)

8.1 Recurrent Neural Network

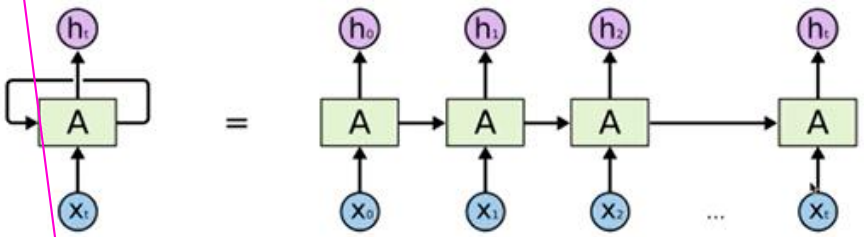
- 우리 주변에 지금 현재의 상태가 다음 상태를 만들어 내는 문제들이 많이 있다.
  예를 들어, 검색어 자동 완성기능을 보면 검색어 몇 자를 입력하면 그 다음 내용이 자동으로 만들어 진다.
  이 원리는 지금 나온 단어와 그 다음 단어는 연관성을 이용하는 것이다.
  RNN은 기존의 신경망 모형을 확장하여 Time Series Forecasting이 가능하도록 만든 모형이다.
- 예를 들어, 현재 내 모습은 "과거의 내 모습 + 현재의 상황" 에 의해 만들어 진다는 상식적인 접근이다.
- 이렇게 시간에 따라 변하는 데이터를 시계열 데이터라고 하고 순서가 중요하며 등 간격 이어야 한다.
- 시계열을 설명하는 가장 간단한 모형은 아래와 같다.
- 아래 첫 번째 모형은 AR(1) 모형이라고 한다. AR은 Autoregressive의 약어다.
- 두 번째 모형은 ARMA(1,1) 모형이다.
- 통계학에서 아래와 같은 형태를 확장하여 ARIMA 모형이라고 하는데 이는 선형모형임을 알 수 있다.

$$Y_t = \phi Y_{t-1} + \epsilon_t$$

$$Y_t = \phi Y_{t-1} + \theta \epsilon_{t-1} + \epsilon_t$$

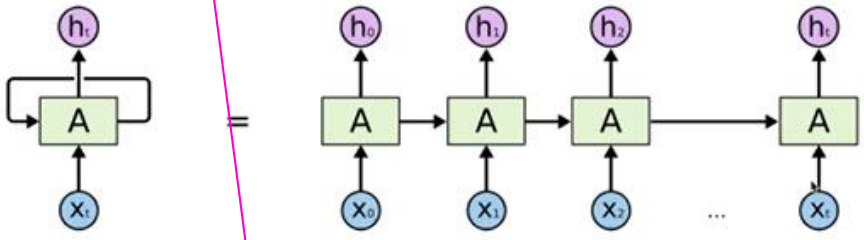
8.1 Recurrent Neural Network

- 아래 그림은 Simple RNN 모형을 표현한 것이다.
- 모형은 입력과 히든 레이어 그리고 출력으로 나눠진다. 히든 레이어가 존재하고 비선형 활성함수를 사용하여 비선형 모형이다.
- 아래 그림에서 순환하는 모양의 화살표가 있는데 이 때문에 순환(Recurrent) 신경망이라고 부른다.
- 순환과정은 우측 그림과 같이 펼칠 수 있다.
  즉, t 시점의 h 는 t 시점의 X와 t-1 시점의 h 가 만든다.



8.1 Recurrent Neural Network

- 이 과정을 수식으로 표현하면 아래와 같다.



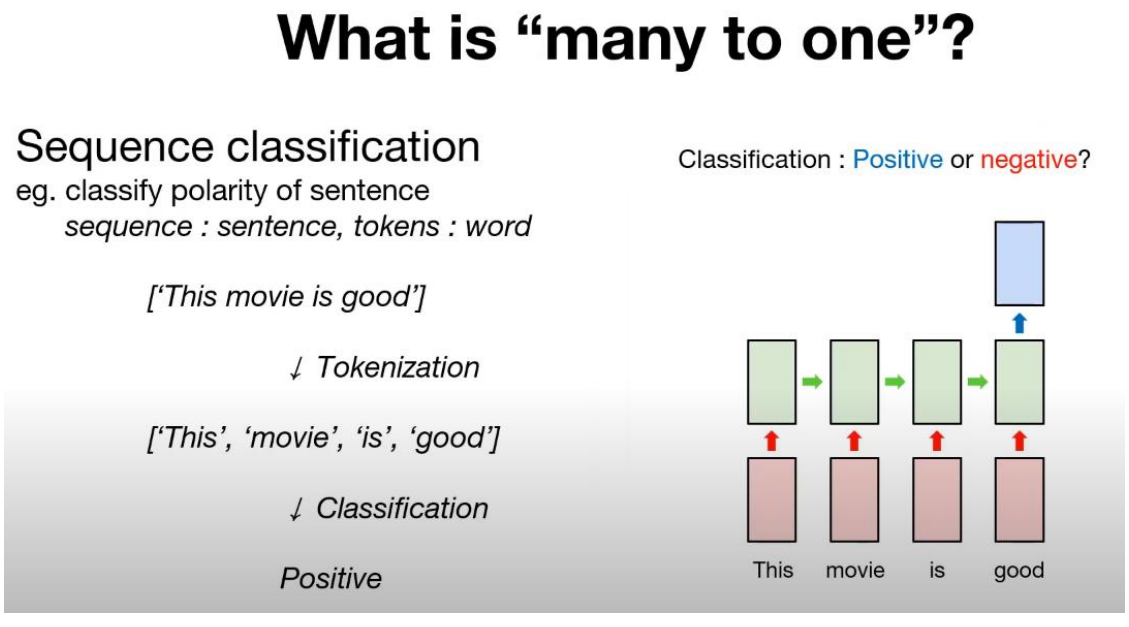
- 여기서, h_t는 current state 이고 h_{t-1}은 old state, x_t는 current state에 영향을 주는 변수다.
- 추정된 y_t와 정답 y_t의 차이를 Cost 함수로 만들어 이를 최소화하는 Weight를 만들어주면 RNN 모형이 완성되는 것이다.
- 순환과정에 사용되는 모든 가중 값은 시점 t에 상관없이 일정한 값이다. 즉, 가중 값 인덱스에 t가 없다.
- y_t에 대한 추정은 아래의 식으로 이루어진다.

$$h_t = f(h_{t-1}, x_t) = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$y_t = W_{hy}h_t$$

- 활성함수 f()는 tanh 혹은 ReLU 함수를 사용한다.

8.3 many to one RNN

many to one RNN은 문장을 판별하는 문제로 예를 들어, 영화평을 입력하면 긍부정을 판단하는 것이다. 문장은 tokenization을 통해 단어들의 모음으로 변환되고 이는 다시 one-hot encoding을 통해 수치로 변환된다.



8.3 many to one RNN

- 아래와 같이 문자 단위로 g→o→o→d 를 입력하면 1 을 출력하고, b→a→d 는 0을 출력 하도록 RNN 모형을 만들어 보자.

```
words = ['good', 'bad', 'worse', 'so good']
y_data = [[1],[0],[0],[1]]
```

- words의 모든 문자를 join 하고 set에 저장한 후, 다시 리스트에 저장한다. 이후, <pad> 라는 "0" 에 해당하는 문자를 삽입한다. 이후, index → character, character → index로 반환하는 딕셔너리를 생성한다.

```
# creating a token dictionary
char_set = ['<pad>'] + sorted(list(set(''.join(words))))
idx2char = {idx : char for idx, char in enumerate(char_set)}
char2idx = {char : idx for idx, char in enumerate(char_set)}

print(char_set)
print(idx2char)
print(char2idx)
```

```
['<pad>', ' ', 'a', 'b', 'd', 'e', 'g', 'o', 'r', 's', 'w']
{0: '<pad>', 1: ' ', 2: 'a', 3: 'b', 4: 'd', 5: 'e', 6: 'g', 7: 'o', 8: 'r', 9: 's', 10: 'w'}
{'<pad>': 0, ' ': 1, 'a': 2, 'b': 3, 'd': 4, 'e': 5, 'g': 6, 'o': 7, 'r': 8, 's': 9, 'w': 10}
```

rnn_many-to-one_TF2.ipynb

8.3 many to one RNN

- “good” 는 [6,7,7,4] 가 되고, “bad” 는 [3,2,4] 이 된다.

```
# converting sequence of tokens to sequence of indices
x_data = list(map(lambda word : [char2idx.get(char) for char in word], words))
x_data_len = list(map(lambda word : len(word), x_data))

print(x_data)
print(x_data_len)
```

```
[[6, 7, 7, 4], [3, 2, 4], [10, 7, 8, 9, 5], [9, 7, 1, 6, 7, 7, 4]]
[4, 3, 5, 7]
```

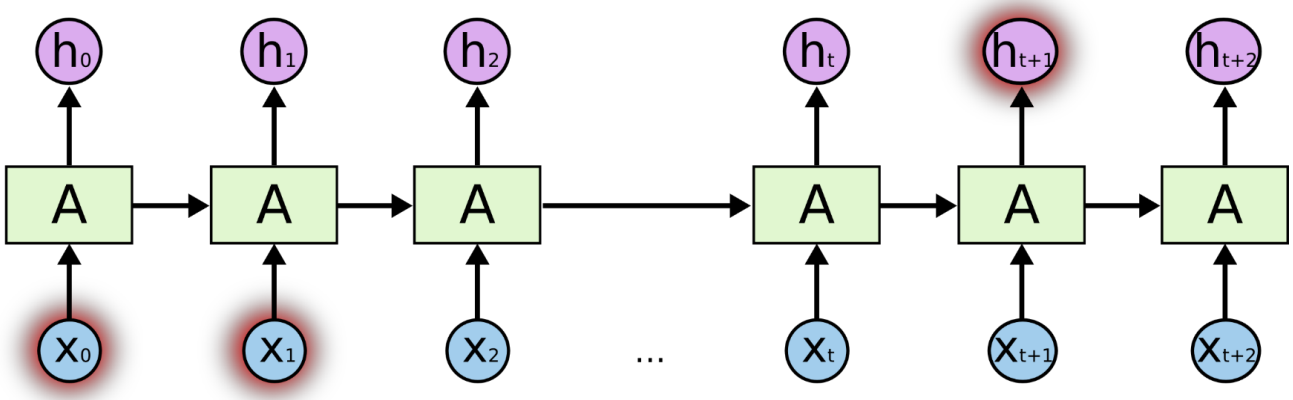
- x_data의 길이가 다 다르다. 이 문제를 해결하기 위해 padding을 사용한다.

```
max_sequence = 10
x_data = pad_sequences(sequences = x_data, maxlen = max_sequence,
                       padding = 'post', truncating = 'post')
y_data = np.array(y_data)
# checking data
print(x_data)
print(y_data)
```

```
[[ 6  7  7  4  0  0  0  0  0  0]
 [ 3  2  4  0  0  0  0  0  0  0]
 [10  7  8  9  5  0  0  0  0  0]
 [ 9  7  1  6  7  7  4  0  0  0]]
[[1]
 [0]
 [0]
 [1]]
```

8.4 Long Short Term Memory

- 문장에서 예를 들어 "cloud in the " 라고 하면 그 다음 단어가 "sky" 일 것이다.
- "I grew up in France. I speak fluent French" 에서 French는 France와 관련이 있다.
- RNN은 과거 정보로 현재정보를 예측하는 구조이므로 제대로 동작하려면 과거 정보를 모두 가지고 있어야 한다.
- 이 때, Sequence 길이가 커져 DNN의 치명적 문제점인 Gradient Vanishing 이슈가 발생한다.

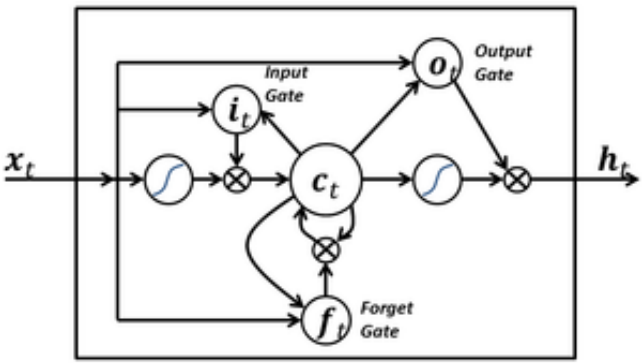


8.4 Long Short Term Memory

- LSTM은 Long Short Term Memory의 약자로 장단기 메모리 문제이다.
- LSTM은 RNN의 치명적 단점인 Gradient Vanishing 문제를 해결하기 위해 만들어진 개념이다.
- LSTM은 아래의 RNN 모델에서 히든 레이어 h_t 를 C_t 로 변형하는 것이다.
- 과거 State를 얼마나 Forget 할 것인가와 현재 state를 얼마나 중요한 정보로 입력할지를 결정하는 forget, input gate를 통해 과거와 현재 정보의 중요도를 결정한다. 예를 들어, "he is ..." 은 주어로 볼 때, 남자이다. 이후, "She is " 가 나오면 앞에서 "he" 로 시작한 문장은 forget해야 한다. 이러한 방식으로 이전 State를 다음 State로 넘길 때, 적절하게 forget하여 히든 레이어의 수가 너무 많아지는 것을 방지함과 동시에 과거 중요정보를 선택적으로 가중 값을 올려주는 방식이다.
- LSTM의 변형형태 들이 많이 나와 있는데 뉴욕대학의 조경현 교수가 제안한 GRU(Gated Recurrent Unit) 도 많이 사용된다.

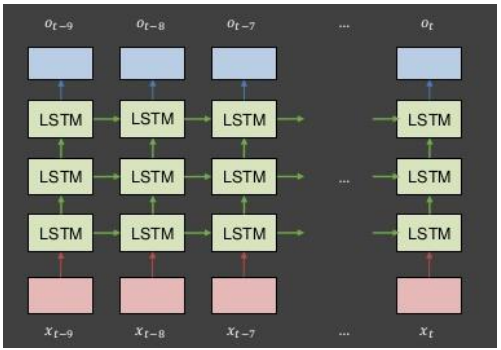
$$h_t = f(h_{t-1}, x_t) = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$\hookrightarrow C_t = f_t \cdot C_{t-1} + i_t \cdot h_t$$



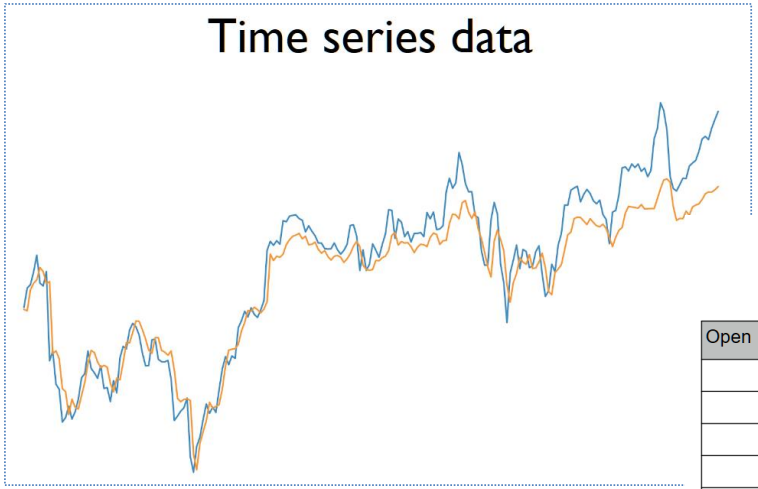
8.5 Stacked RNN

- 좀 더 긴 문장을 학습해보자. 이 때, 필요한 것이 Batch와 Stacked RNN이다.
  많은 문장을 학습할 때, 글 전체를 하나의 Sequence로 학습하는 것은 적절치 않다.
  하나의 문장은 나름 연관된 구조를 가지고 있지만, 글 전체로 보면 앞 문장과 뒤 문장의 연관성이 떨어진다.
- 이럴 때, Batch 학습을 한다.
- Batch 학습은 독립적으로 인정되는 단위로 끊어서 학습하는 방법이다.
  만약, 10자 씩 끊어서 학습한다면 "That, poor", "hat, poor ", "at, poor c"와 같이 여러 개의 문장이 나올 것이다. Batch 학습을 하면 모형의 정확도가 급속도로 떨어진다.
- 모형의 정확도를 올리기 위해서는 Cell의 수를 크게 해야 한다.
  정확도를 올리려면 Recurrent 층을 여러 개로 확장하는 Stacked RNN을 사용한다.



8.6 RNN Example(Stock Data)

- 통계학의 분야로 시계열 분석이 있다. 시계열은 아래와 같이 시간에 따라 달라지는 값을 분석하는 것이다. 분석의 주된 목적은 미래에 대한 예측이다.
- 이러한 분석은 many to one RNN을 사용해서도 구현이 가능하다.



Time series data

Open	High	Low	Volume	Close
828.659973	833.450012	828.349976	1247700	**831.659973**
823.02002	828.070007	821.655029	1597800	**828.070007**
819.929993	824.400024	818.97998	1281700	**824.159973**
819.359985	823	818.469971	1304000	**818.97998**
819	823	816	1053600	**820.450012**
816	820.958984	815.48999	1198100	**819.23999**
811.700012	815.25	809.780029	1129100	**813.669983**
809.51001	810.659973	804.539978	989700	**809.559998**
807	811.840027	803.190002	1155300	**808.380005**

8.6 RNN Example(Stock Data)

- 이 데이터를 7일 단위로 학습한다면 입력길이는 데이터의 dimension 50이고 Sequence 길이 70이다.

Time series data

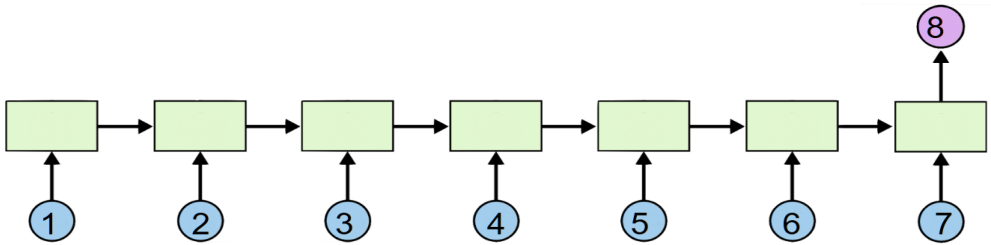
Open	High	Low	Volume	Close
828.659973	833.450012	828.349976	1247700	831.659973
823.02002	828.070007	821.655029	1597800	828.070007
819.929993	824.400024	818.97998	1281700	824.159973
819.359985	823	818.469971	1304000	818.97998
819	823	816	1053600	820.450012
816	820.958984	815.48999	1198100	819.23999
811.700012	815.25	809.780029	1129100	813.669983
809.51001	810.659973	804.539978	989700	809.559998
807	811.840027	803.190002	1155300	808.380005

출력 Y[i]

입력 X[i]

Many to one

입력길이=5



8.6 RNN Example(Stock Data)

- 주가 자료는 양적자료 이므로 신경망 모형에 사용하기 위해 표준화(MinMaxScaler)가 필요하다.
- (Open, High, Low, Volume) 으로 Close를 맞추는 모형을 만들어 보자.
- 주식의 종가(Close)는 오늘의 (Open, High, Low, Volume) 의 영향도 있지만 과거의 (Open, High, Low, Volume, Close) 영향도 받을 것이다.
- 여기서는 7일 간의 과거에 영향을 받는다고 가정했다.

```
def MinMaxScaler(data):
    numerator = data - np.min(data, 0)
    denominator = np.max(data, 0) - np.min(data, 0)
    # noise term prevents the zero division
    return numerator / (denominator + 1e-7)

# Open, High, Low, Volume, Close
xy = np.loadtxt('/gdrive/My Drive/DeepLearning/RNN/stock_daily.csv', delimiter=',')
xy = xy[::-1]  # reverse order
xy = MinMaxScaler(xy)
x = xy
y = xy[:, [-1]]  # Close as label
seq_length = 7   # input sequence
```

rnn_stockPrice_TF2.ipynb

8.6 RNN Example(Stock Data)

- 학습을 위해 과거 7일간의 5개 (Open, High, Low, Volume, Close) 주가와 현재 종가를 각각 _x, _y 로 만들고 이를 dataX, dataY 리스트로 저장한다.

```
# build a dataset
dataX = []
dataY = []
for i in range(0, len(y) - seq_length):
    _x = x[i:i + seq_length]
    _y = y[i + seq_length]  # Next close price
    print(_x, "->", _y)
    dataX.append(_x)
    dataY.append(_y)
```

```
[[0.12705592 0.1390988  0.12324434 0.23008873 0.11649107]
 [0.12288872 0.12276776 0.09067269 0.34481491 0.12791587]
 [0.14089803 0.17638508 0.15387945 0.4378686  0.18649648]
 [0.15782902 0.15470702 0.12791016 0.60962624 0.12692483]
 [0.1207904  0.11770961 0.11162416 0.22934481 0.10512477]
 [0.09905103 0.11924158 0.11722888 0.21129336 0.12316526]
 [0.11405885 0.10952968 0.11353161 0.18323922 0.10022843]] -> [0.09504062]
```

8.6 RNN Example(Stock Data)

- 데이터의 앞부분 70%를 train 데이터로 하고 나머지 30%는 test 데이터로 한다.
- input_shape는 (7,5) 로 지정하고 many2one 을 위해 return_sequences = False로 지정한다.
- 회귀문제이므로 loss는 MSE로 지정한 후, 배치크기 16개로 100회 학습한다.

```
# train/test split
train_size = int(len(dataY) * 0.7)
test_size = len(dataY) - train_size
trainX, testX = np.array(dataX[0:train_size]), np.array(dataX[train_size:len(dataX)])
trainY, testY = np.array(dataY[0:train_size]), np.array(dataY[train_size:len(dataY)])

print(trainX.shape, trainY.shape)
```

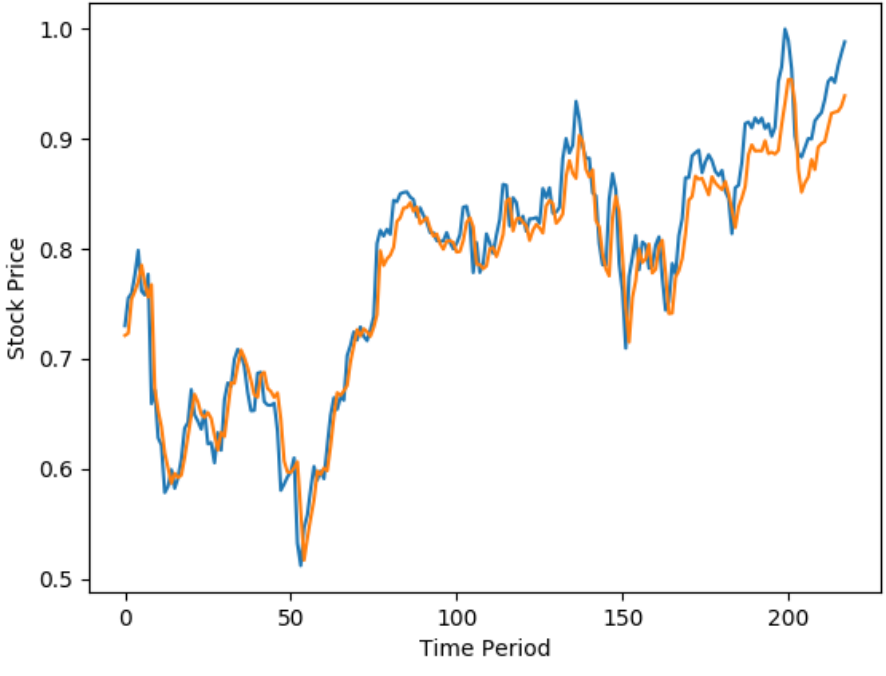
```
(507, 7, 5) (507, 1)
```

```
model = Sequential()
model.add(layers.LSTM(16, input_shape=(trainX.shape[1], trainX.shape[2]), activation='relu', return_sequences=False))
model.add(layers.Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.summary()
```

```
history = model.fit(trainX, trainY, epochs=100, batch_size=16, verbose = 2)
pred = model.predict(testX)
```

8.6 RNN Example(Stock Data)

- 테스트 데이터 셋의 데이터를 검증한 결과, 아래와 같이 훌륭하게 예측됨을 알 수 있다.
- 7개를 주고 그 다음 한 개를 맞추는 방식이다. 이런 방식을 one-step-ahead forecast 라고 한다.



8.7 RNN Example(imdb)

- imdb dataset은 영화 리뷰 데이터로 훈련 데이터와 테스트 데이터는 각각 2만 5천개다.
- 각 단어 중 최빈수 기준으로 1,000개의 단어만 가져오기로 한다.
- 각 sentence에 대해 부정, 긍정 으로 구성되어 있고 긍정과 부정 비율은 5:5 다.
- 각 sentence가 길어서 마지막 부터 80단어만 선택해 학습하기로 하자.

```
[1]  1 from keras.preprocessing import sequence
     2 from keras.datasets import imdb
     3 from keras import layers, models
```

```
Using TensorFlow backend.
```

```
[11] 1 max_features = 10000
     2 (x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
     3 print(x_train[0]) # This is a sequence of expressions of sentiment about a certain movie
```

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100,
```

```
[3]  1 maxlen=80
     2 x_train = sequence.pad_sequences(x_train, maxlen=maxlen) # cutting last 80 words
     3 x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
```

RNN_imdb_TF2.ipynb

8.7 RNN Example(imdb)

- 80개의 스퀀스에 대해 1,000개의 one-hot 벡터를 만들면 입력 크기는 8만개가 된다.
- 천 개의 one-hot vector를 128개의 임베딩 벡터로 바꿔서 메모리도 절약하고 단어의 의미도 표현할 수 있도록 한다.
- 각 단어 sequence를 128개의 LSTM 레이어를 통과하여 many2one 구조의 RNN 모형을 만든다.
- 이 모형의 최종 결과는 83% 정확도를 가진다.

```
x=layers.Input((maxlen,))
h=layers.Embedding(max_features, 128)(x)
h=layers.LSTM(128)(h)
y=layers.Dense(1, activation='sigmoid')(h)
model=models.Model(x,y)

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

```
batch_size=32
epochs=3
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test))
score, acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print('Test performance: accuracy={0}, loss={1}'.format(acc, score))
```