순환신경망 RNN

7장 순환신경망

2020.08.15목 2h

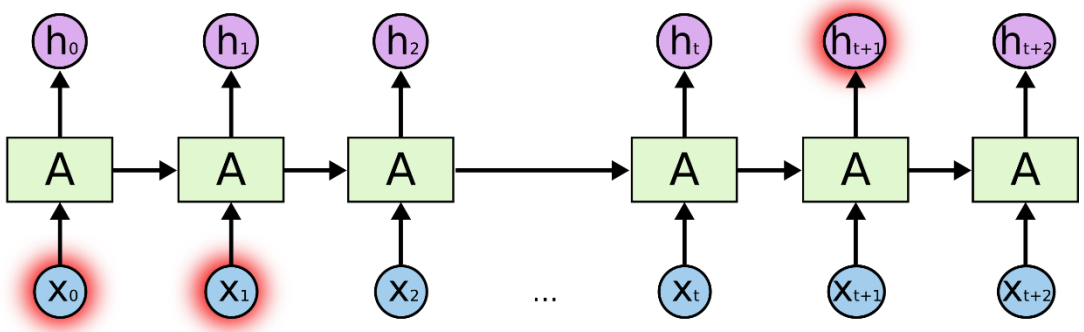
7장 순환 신경망

2 LSTM layers

1.5h

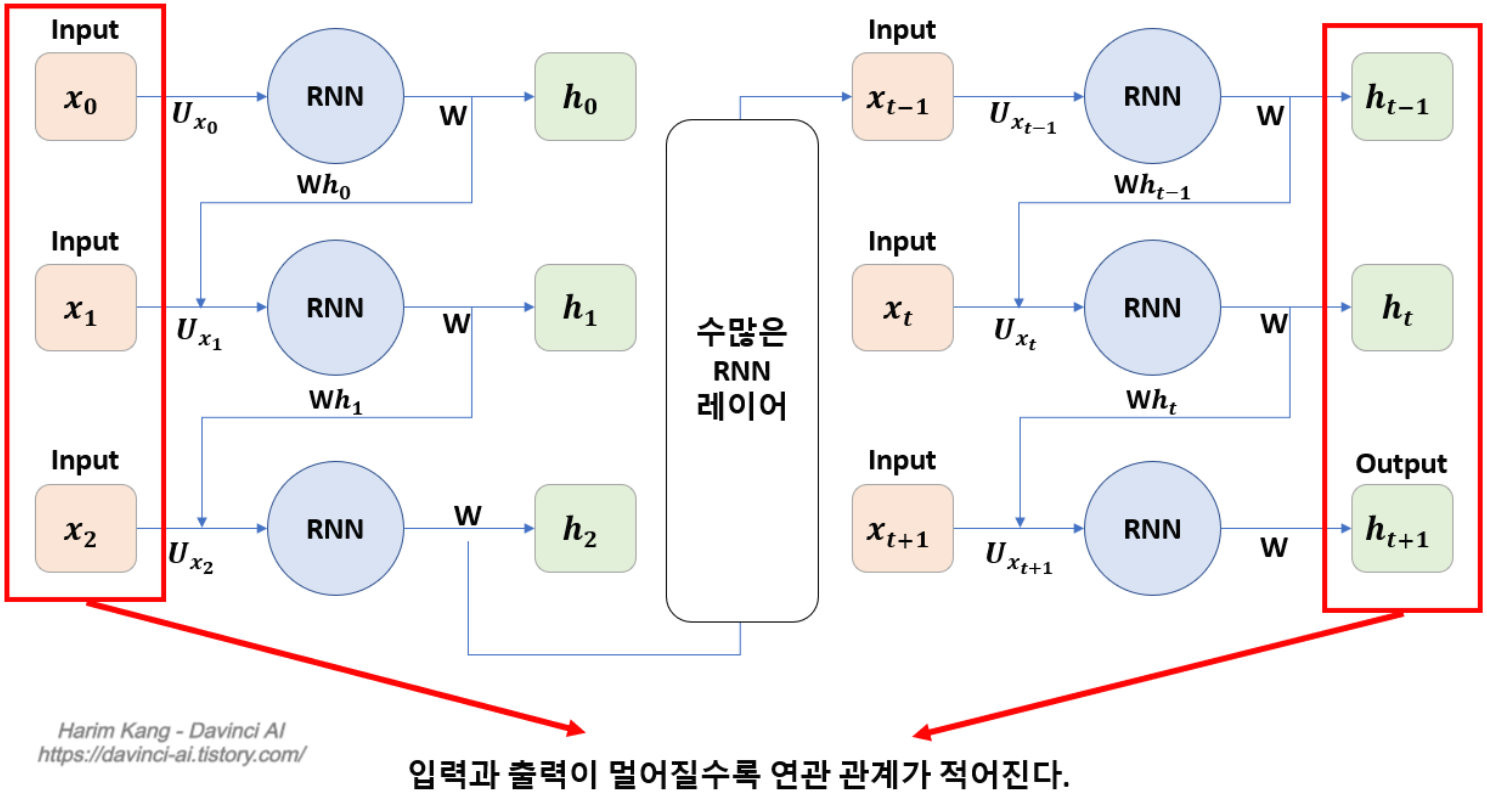
RNN(Recurrent Neural Network) 문제

- **경사 소실 문제**
  - 시간을 많이 거슬러 올라갈수록(long term) 경사를 소실하는 문제가 있음
    - **초기값에 따라서 과거 데이터를 계속 곱할수록 작아지는 문제가 발생**
- **장기 의존성 (Long-Term Dependency) 문제**
  - 입력 데이터가 길어질수록, 즉 데이터의 타임 스텝이 길어질수록 학습 능력이 떨어진다는 점
  - 입력 데이터와 출력 사이의 길이가 멀어질수록 연관 관계가 적어지는 문제
    - **예: 책을 읽을 때, 몇 페이지/챕터 전에 있는 정보를 머리 속에 기억하고 있어야 하는 경우**
  - 시퀀스가 있는 문장에서 문장 간의 간격(gap, 입력 위치의 차이)이 커질수록
    - **RNN은 두 정보의 맥락을 파악하기 어려워 짐**



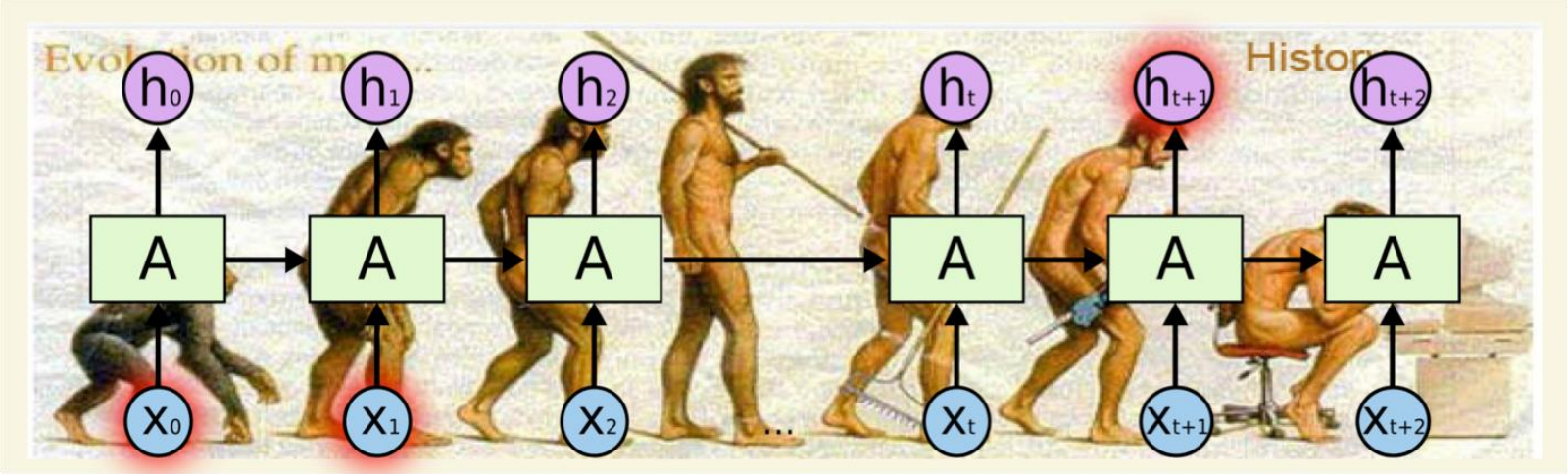
장기 의존성(Long-Term Dependency)문제

- **입력 데이터와 출력 데이터 사이의 길이가 멀어질수록**
  - 연관 관계가 줄어드는 문제



LSTM(Long Short Term Memory) 개요

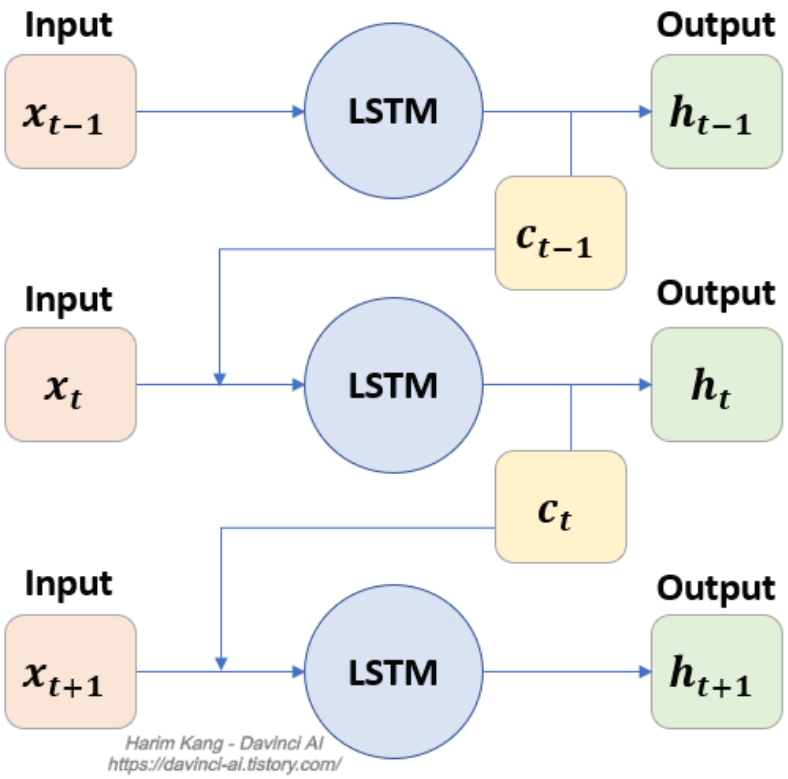
- **1997년, 셉 호흐라이터(Sepp Hochreiter) 유르겐 슈미트후버(Jurgen Schmidhuber)에 의해 제안**
  - 많은 개선을 통해 언어, 음성인식 등 다양한 분야에서 사용
  - RNN의 장기 의존성 문제를 해결할 수 있음
  - 직전 데이터뿐만 아니라, 좀 더 거시적으로 과거 데이터를 고려하여 미래의 데이터를 예측하기 위함

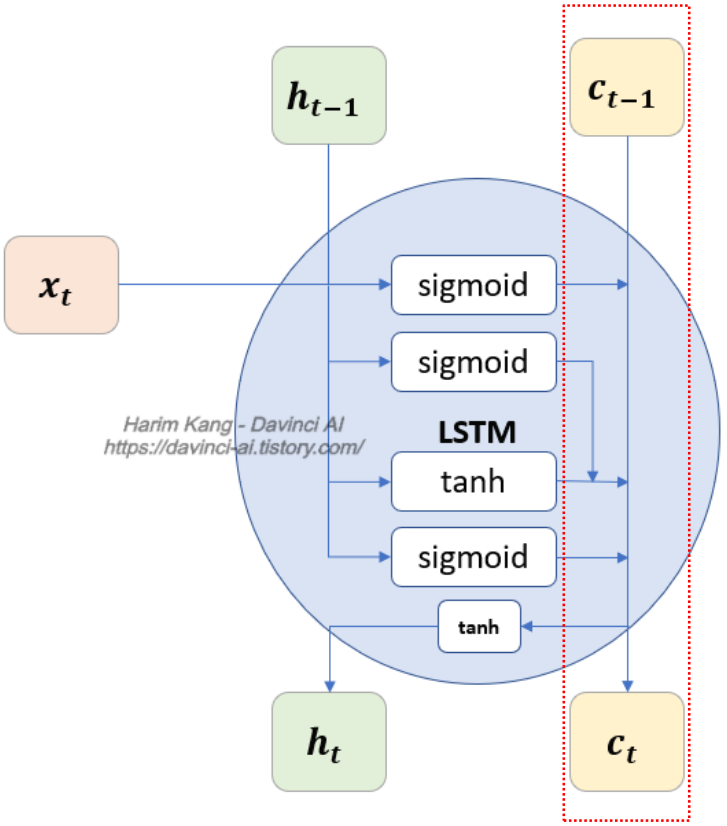


https://dgkim5360.tistory.com/entry/understanding-long-short-term-memory-lstm-kr

LSTM의 구조

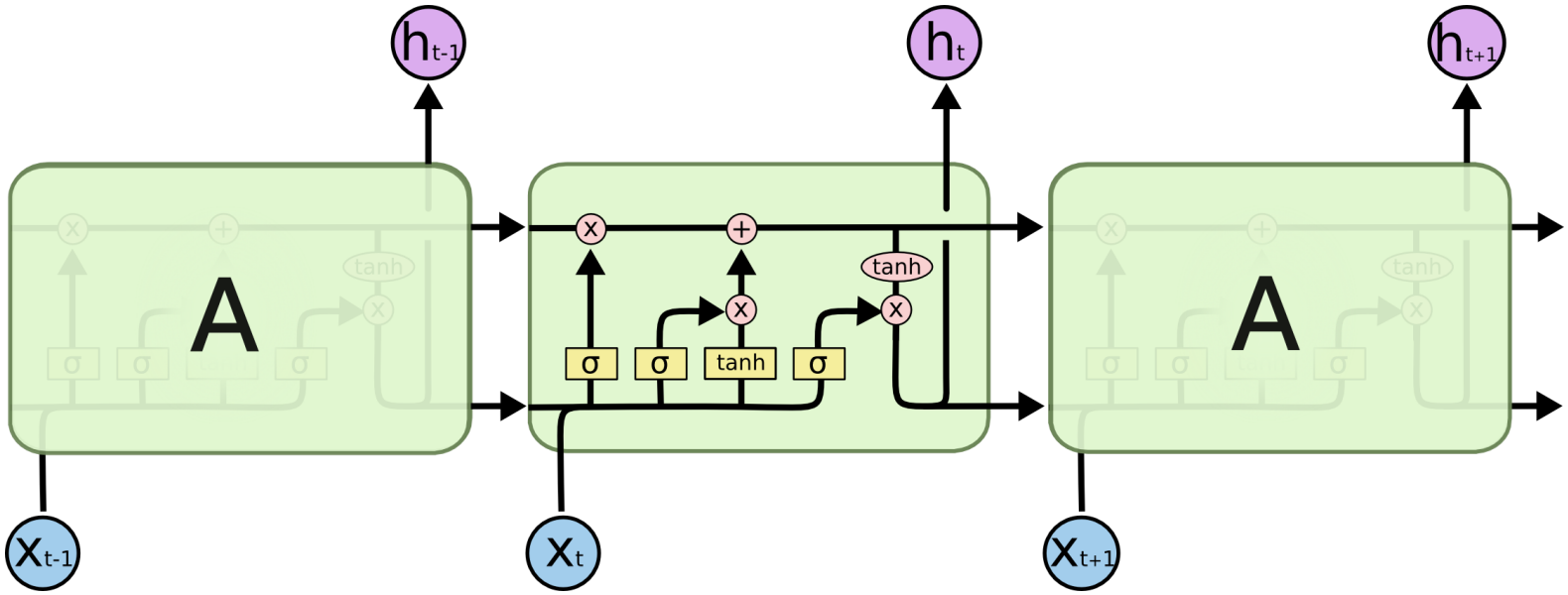
- **셀 상태(cell state) 변수 활용, Ct-1, Ct**
  - 장기 의존성 문제 해결
    - **LSTM 셀 사이에서 셀 상태가 보존되기 때문에**





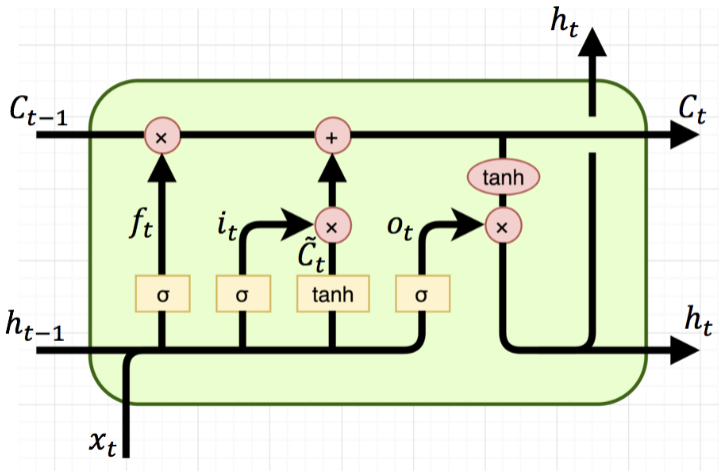
활성화 함수 시그모이드와 tanh

- **시그로이드 함수: (0, 1)**
  - 이 값은 각 컴포넌트가 얼마나 정보를 전달해야 하는지에 대한 척도를 표현
    - **값이 0이라면 "아무 것도 넘기지 말라"가 되고**
    - **값이 1이라면 "모든 것을 넘겨드려라"라는 의미**
- **tanh 함수: (-1, 1)**



LSTM 계산 흐름

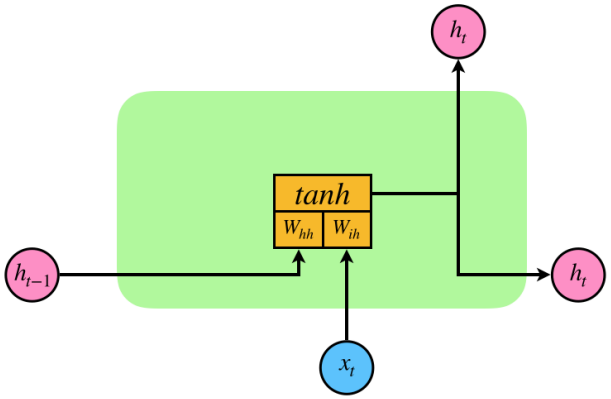
- 장기 의존성 문제를 해결



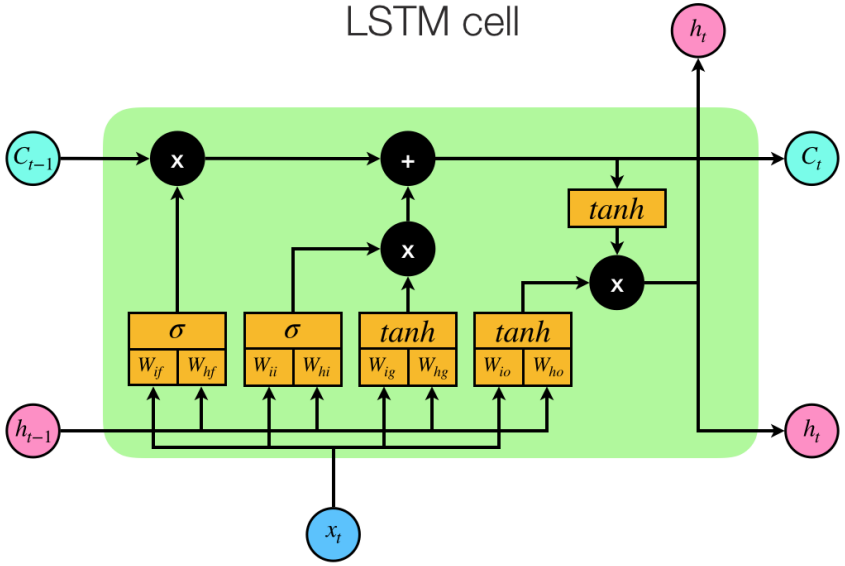
LSTM: solves long-term dependencies

Cell State & Gates

RNN cell

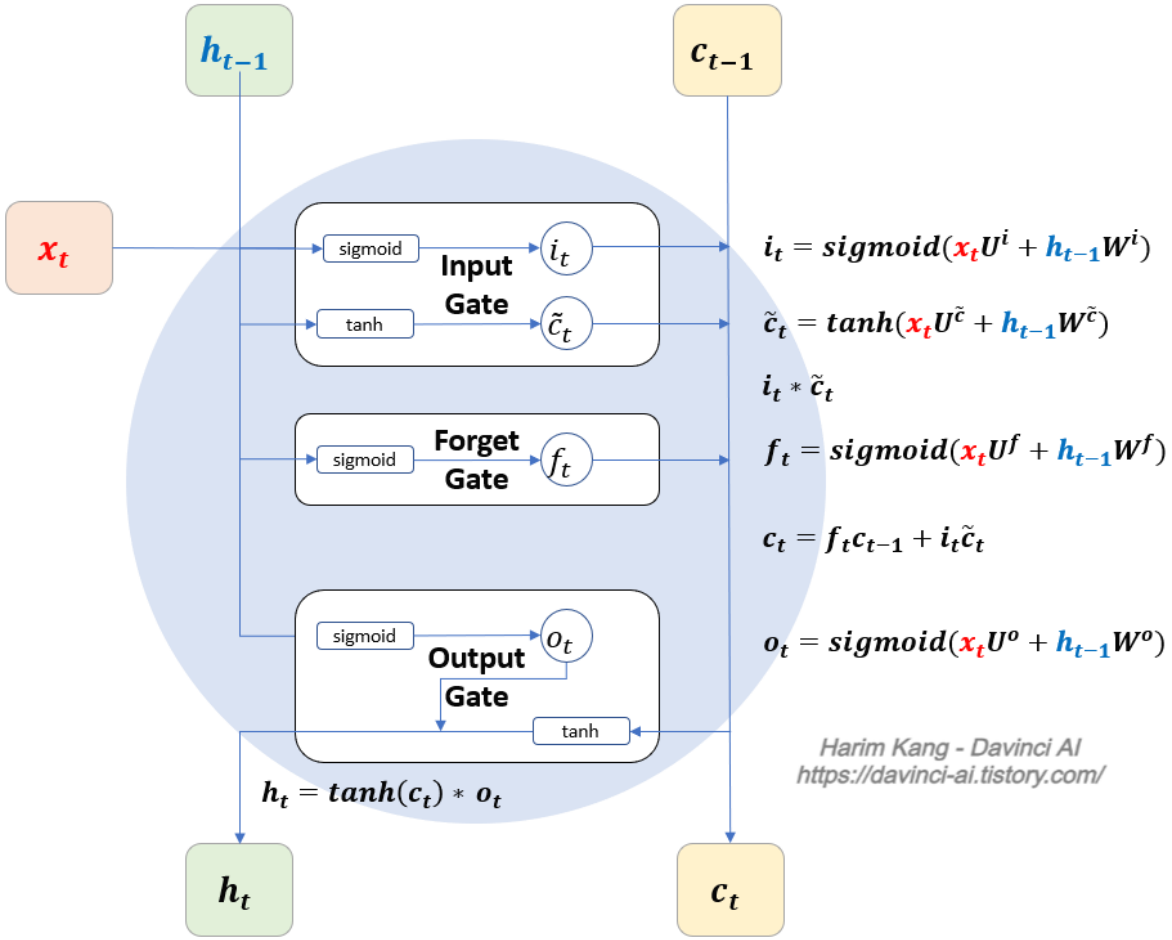


LSTM cell



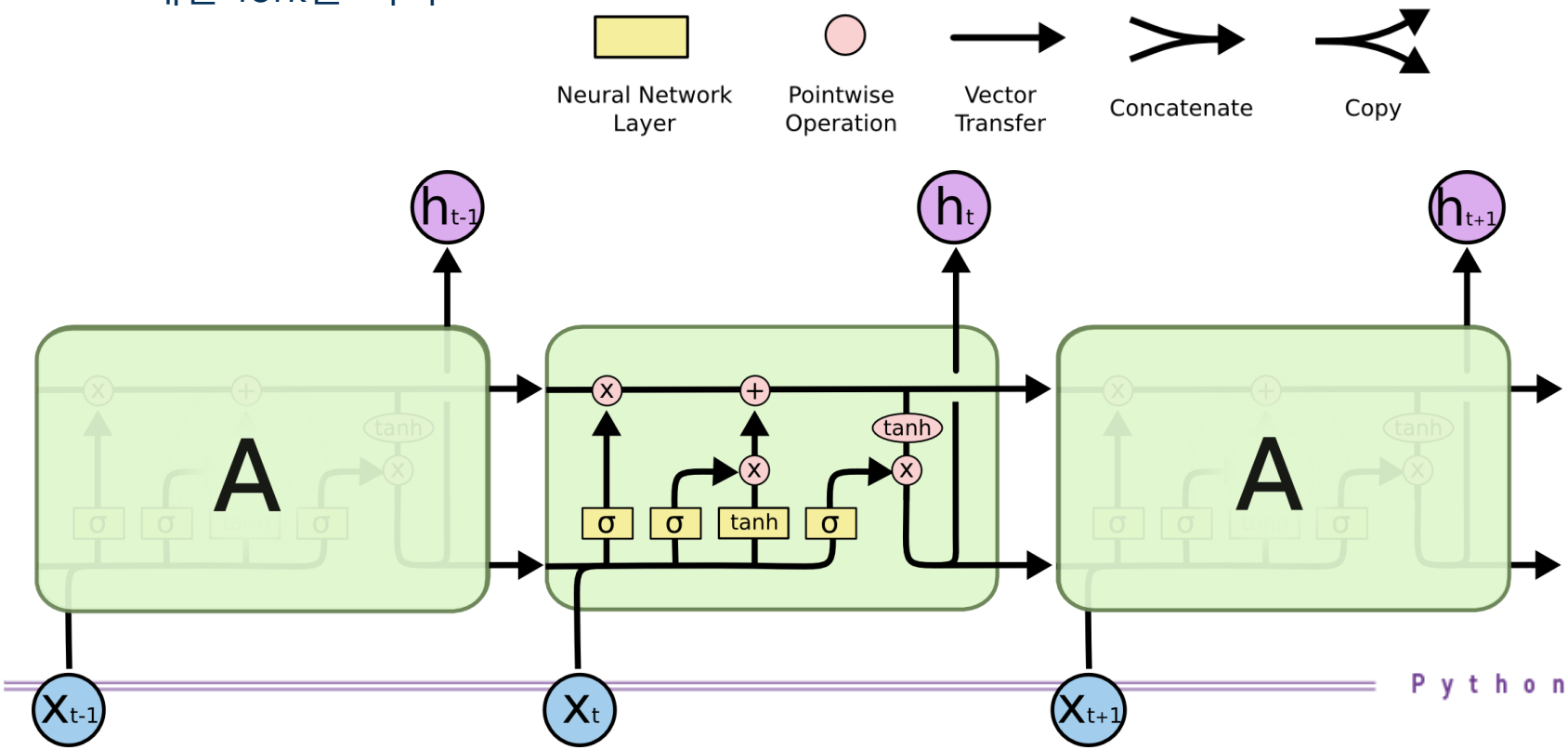
LSTM의 셀의 세부 구조

- 값을 얼마나 통과시킬 것인가를 제어하는 게이트
  - Input Gate
  - Forget Gate
  - Output Gate
- U, W
  - 가중치
- it, ft, ot
  - Input, forget, output 게이트를 통과한 출력
- $\tilde{c}_t$
  - SimpleRNN에도 있던 뉴런의 출력
  - 셀 상태인 Ct가 되기 전의 출력 값



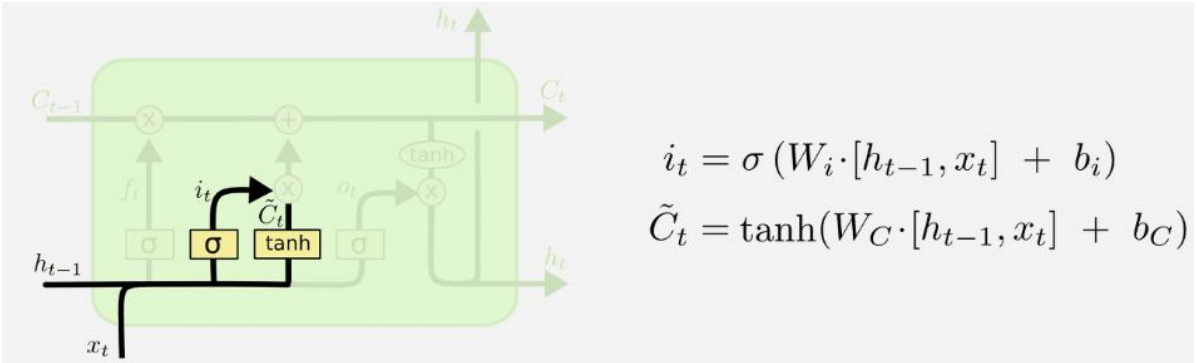
LSTM의 다양한 표현

- **LSTM의 반복 모듈에는 4개의 상호작용하는 layer로 구성**
  - 각 선(line)은 한 노드의 output을 다른 노드의 input으로 보내는 흐름을 표현
  - 분홍색 동그라미는 vector 합과 같은 pointwise operation을 표현
  - 노란색 박스는 학습된 neural network layer
  - 합쳐지는 선은 concatenation을 의미, 갈라지는 선은 정보를 복사해서 다른 쪽으로 보내는 fork를 의미



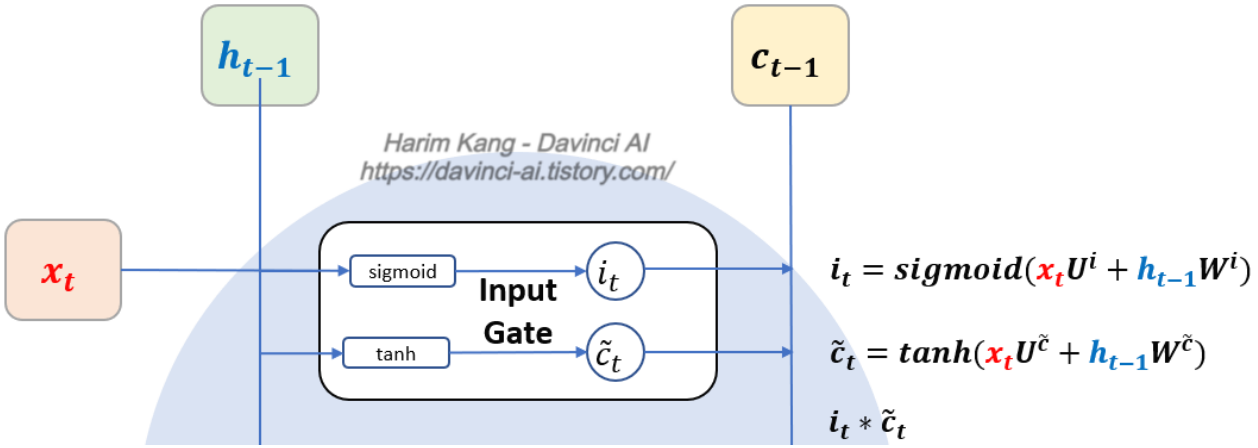
Input gate layer

- 이전 상태의 hidden state(h)와 현재 상태의 input(x)값에 대해
  - 시그모이드와 하이퍼볼릭탄젠트(tanh)로 연산
    - **시그모이드를 지난 값은 0~1의 값을 가지며 gate의 역할을 하게 되고(input gate),**
    - **tanh를 지난 값은 -1~1의 값을 가지며 현재 cell state를 나타내게 표현**



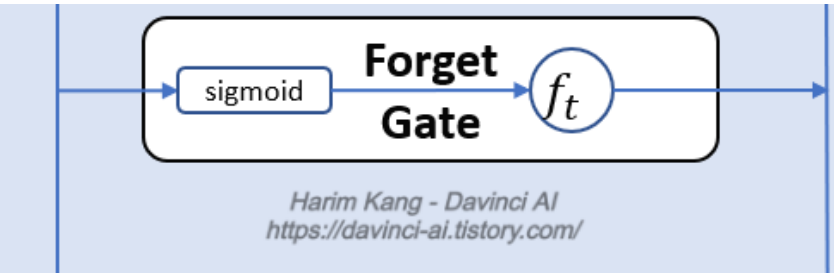
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$



Forget gate layer

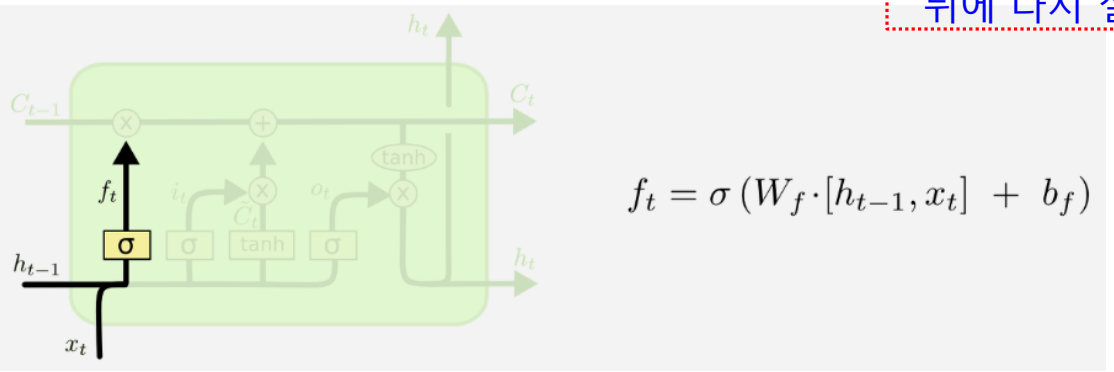
- 이전의 셀 정보를 어느 정도 잊을 지에 대한 게이트
  - 얼마나 잊어버릴 지에 대한 가중치를 사용
    - 이전 state c에 forget gate 출력 값을 곱해서 state를 업데이트
      - input 게이트를 통해 나온 출력 값들을 곱하여 state에 합
    - 가중치
      - forget을 위한 Uf, Wf



$$f_t = sigmoid(x_t U^f + h_{t-1} W^f)$$

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

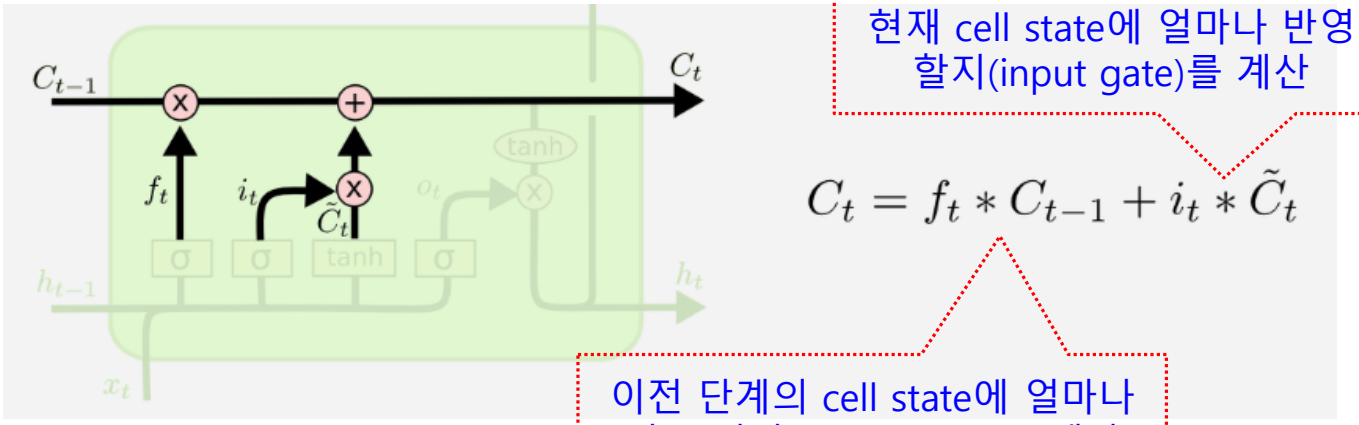
뒤에 다시 설명



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

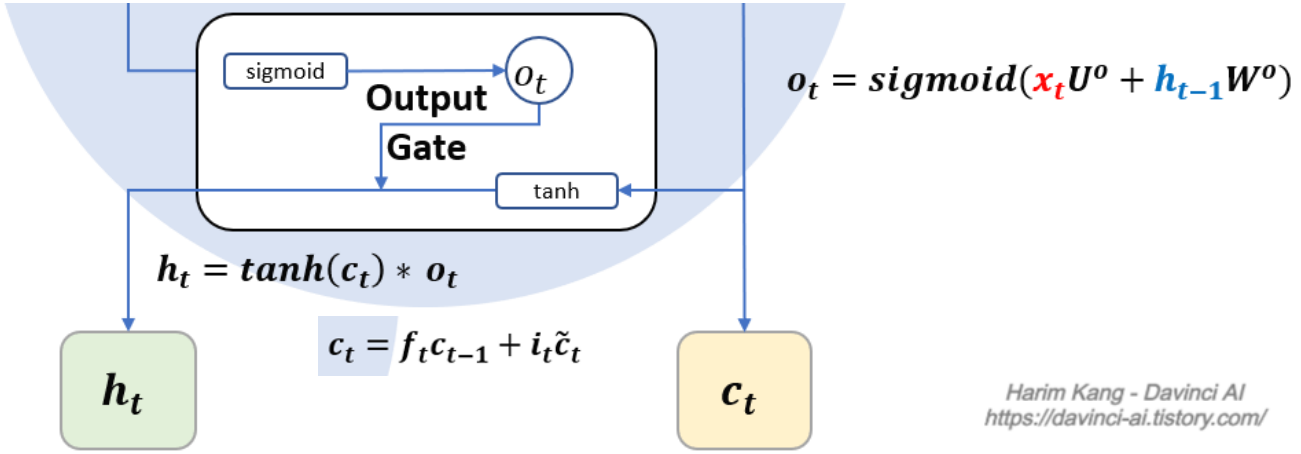
현 상태 수정

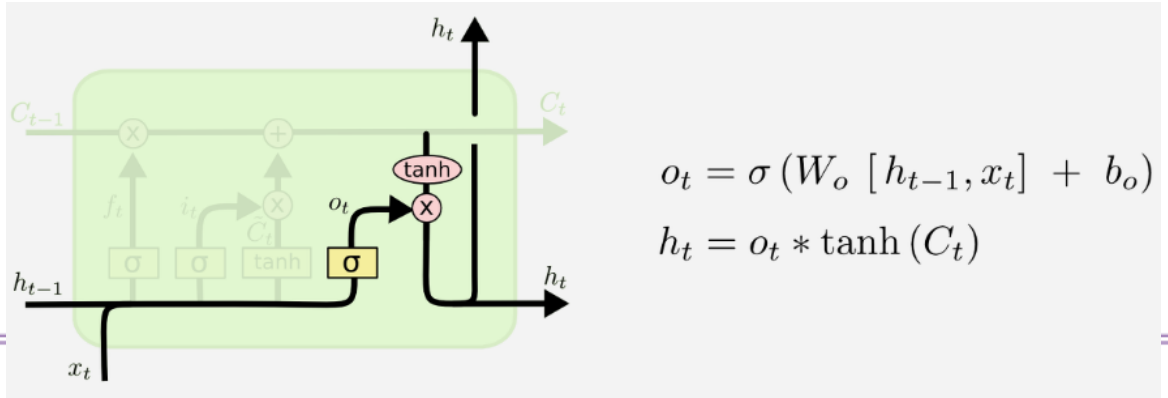
- **현 셀 상태 값 업데이트**
  - cell state의 값
    - **이전 셀 값과 현재 셀의 계산 값 사용**
  - forget gate의 값
    - **이전 타임 스텝의 셀 상태를 얼마만큼 남길지가 결정**
  - input gate의 값을 사용
    - **현 타임 스텝의 셀 상태를 얼마만큼 남길지가 결정**



Output gate layer

- 어떤 값을 출력할지를 결정하는 게이트
  - sigmoid layer: Ot
  - 최종 출력 값을 연산
    - cell state를 tanh 하여 나온 -1과 1 사이의 값을 sigmoid 출력 값과 곱





파일

- **ch7_RNN_study.ipynb**

곱셈 문제(Multiplication Problem)

- **LSTM의 학습 능력을 확인하기 위한 예제**
  - LSTM을 처음 제안한 논문에 나온 실험 여섯 개 중 다섯 번째
  - 고려해야 할 실수의 범위가 100개이고 그 중에서 마킹된 두 개의 숫자만 곱하기
    - **책 p186 그림 7.9**

문제와 정답 데이터 생성

- **데이터**
  - 2560개 학습 데이터
  - 440개 테스트 데이터
  - 1이 마킹된 두 개의 원소의 곱

```
X = []
Y = []
for i in range(3000):
  # 0 ~ 1 범위의 랜덤한 숫자 100개를 만듭니다.
  lst = np.random.rand(100)

  # 마킹할 숫자 2개의 인덱스를 뽑습니다.
  idx = np.random.choice(100, 2, replace=False)

  # 마킹 인덱스가 저장된 원-핫 인코딩 벡터를 만듭니다.
  zeros=np.zeros(100)
  zeros[idx]=1

  # 마킹 인덱스와 랜덤한 숫자를 합쳐서 X에 저장합니다.
  X.append(np.array(list(zip(zeros, lst))))
  # 마킹 인덱스가 1인 값만 서로 곱해서 Y에 저장합니다.
  Y.append(np.prod(lst[idx]))

print(X[0], Y[0])
```

```
[[0.         0.56858055]
 [0.         0.69588629]
 [0.         0.67735798]
 [0.         0.27871065]
 [0.         0.35586034]
 [0.         0.12100308]
 [1.         0.14539513]
 [0.         0.43342875]
 [0.         0.49999051]
 [0.         0.89730127]

 [0.         0.86271116]
 [0.         0.96327191]
 [0.         0.98042503]
 [0.         0.5772363 ]
 [1.         0.21461413]
 [0.         0.95933064]
 [0.         0.22111469]
 [0.         0.24241146]
 [0.         0.75950882]
 [0.         0.51717453]

 [0.         0.3700633 ]
 [0.         0.55055634]
 [0.         0.20978592]] 0.031203849491372 52
```

X (곱하기)

모델 생성

- **뉴런 수 10**
  - return_sequences=True로 설정
    - **RNN 레이어를 겹치기 위해 첫 번째 SimpleRNN 레이어 설정**
      - **return_sequences는 레이어의 출력을 다음 레이어로 그대로 넘겨 줌**

```
[ ]  1 model = tf.keras.Sequential([
     2   tf.keras.layers.SimpleRNN(units=30, return_sequences=True, input_shape=[100,2]),
     3   tf.keras.layers.SimpleRNN(units=30),
     4   tf.keras.layers.Dense(1)
     5 ])
     6
     7 model.compile(optimizer='adam', loss='mse')
     8 model.summary()
```

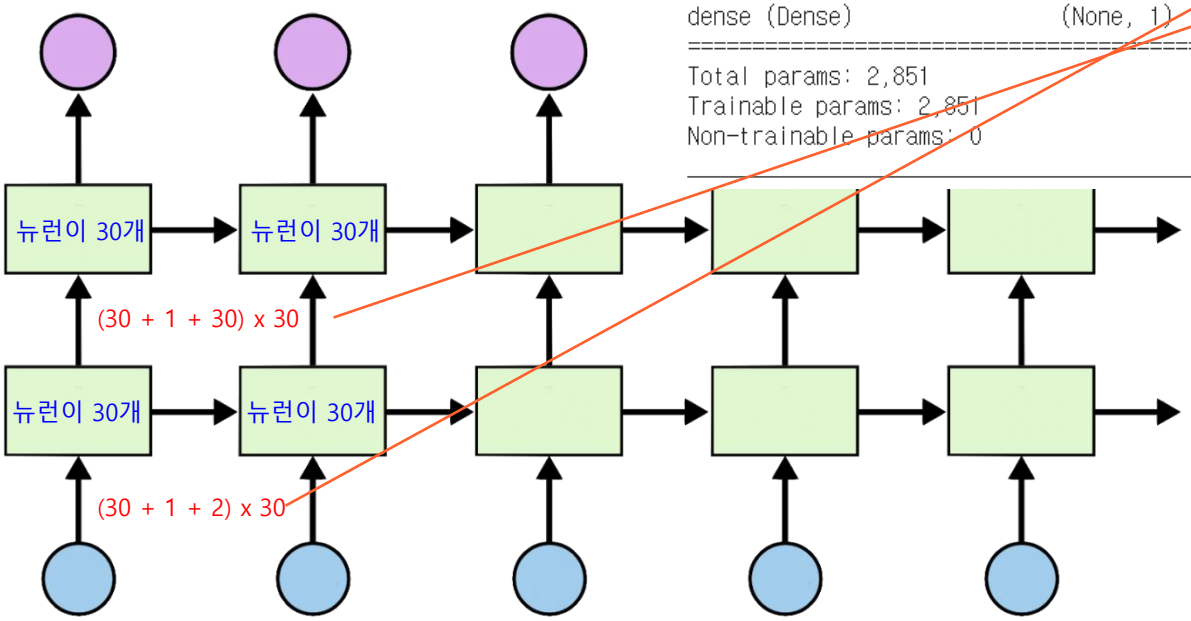
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 100, 30)           990
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 30)                1830
_________________________________________________________________
dense (Dense)                (None, 1)                 31
=================================================================
Total params: 2,851
Trainable params: 2,851
Non-trainable params: 0
_________________________________________________________________
```

다중 RNN(Stacked RNN)

- **RNN layer도 여러 겹 쌓아 다중 RNN을 구현** p188 그림 7.10 참조

```
model = tf.keras.Sequential([
  tf.keras.layers.SimpleRNN(units=30, return_sequences=True, input_shape=[100,2]),
  tf.keras.layers.SimpleRNN(units=30),
  tf.keras.layers.Dense(1)
])
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 100, 30)           990
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 30)                1830
_________________________________________________________________
dense (Dense)                (None, 1)                 31
=================================================================
Total params: 2,851
Trainable params: 2,851
Non-trainable params: 0
_________________________________________________________________
```



학습

- **반드시 GPU 설정**
  - 런 타임 | 런 타임 유형 변경

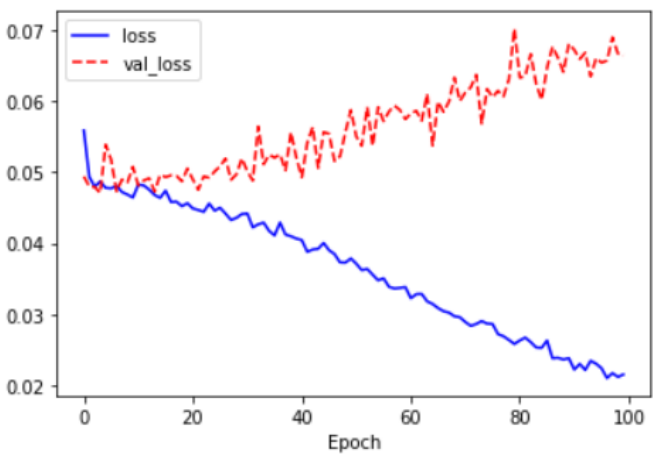
```
X = np.array(X)
Y = np.array(Y)

# 2560개의 데이터만 학습시킵니다. 검증 데이터는 20%로 저장합니다.
History = model.fit(X[:2560], Y[:2560], epochs=100, validation_split=0.2)
```

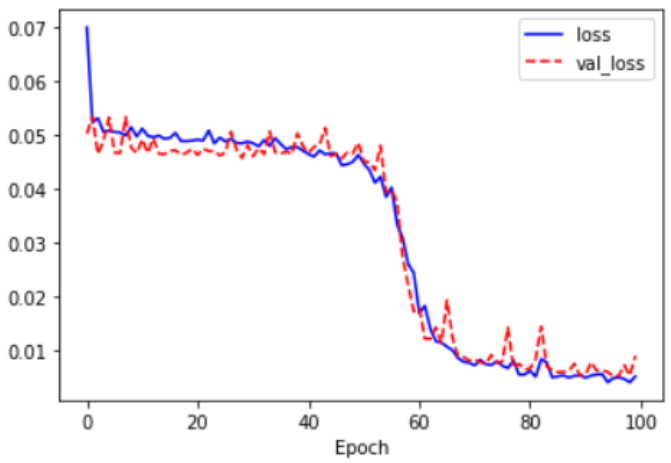
훈련 과정 시각화

- **손실 시각화**
  - 훈련 데이터의 손실(loss)과 검증 데이터의 손실(var_loss)는 감소하지 않고 오히려 증가
  - 잘 되는 경우도 있는 듯

```
[ ]  1 import matplotlib.pyplot as plt
     2 plt.plot(history.history['loss'], 'b-', label='loss')
     3 plt.plot(history.history['val_loss'], 'r--', label='val_loss')
     4 plt.xlabel('Epoch')
     5 plt.legend()
     6 plt.show()
```




```
[11]  1 import matplotlib.pyplot as plt
      2 plt.plot(history.history['loss'], 'b-', label='loss')
      3 plt.plot(history.history['val_loss'], 'r--', label='val_loss')
      4 plt.xlabel('Epoch')
      5 plt.legend()
      6 plt.show()
```



테스트 데이터에 대한 예측

- **논문에서는 오차가 0.04 이상일 때 오답으로 처리**

```
[ ]
model.evaluate(X[2560:], Y[2560:])
prediction=model.predict(X[2560:2560+5])

# 5개 테스트 데이터에 대한 예측을 표시합니다.
for i in range(5):
  print(Y[2560+i], '₩t', prediction[i][0], '₩tdiff:', abs(prediction[i][0] - Y[2560+i]))

prediction = model.predict(X[2560:])
fail = 0
for i in range(len(prediction)):
  # 오차가 0.04 이상이면 오답입니다.
  if abs(prediction[i][0] - Y[2560+i]) > 0.04:
    fail +=1

print('correctness:', (440-fail)/440*100, '%')
```

```
14/14 [==============================] - 0s 16ms/step - loss: 0.0667
0.009316712705671063      -0.05508966    diff: 0.06440637269455123
0.1595728709352136        0.051416673    diff: 0.1081561981690049б
0.343633615267837         0.27744463     diff: 0.06618898440655463
0.04783629085022783б      0.23675384     dif
0.07471709800989841       0.19511518     dif
correctness: 12.727272727272727 %
```

```
14/14 [==============================] - 0s 17ms/step - loss: 0.0095
0.0509529573170782        0.031904534    diff: 0.0190484238116384 4
0.13046067886059912       0.14164282     diff: 0.011182144954746648
0.12109961540418425       0.21557826     diff: 0.09447864263338289
0.004848609688217153      0.14511059     diff: 0.14026198255783845
0.40852896246832054       0.6411967      diff: 0.232667765284365
correctness: 29.772727272727273 %
```

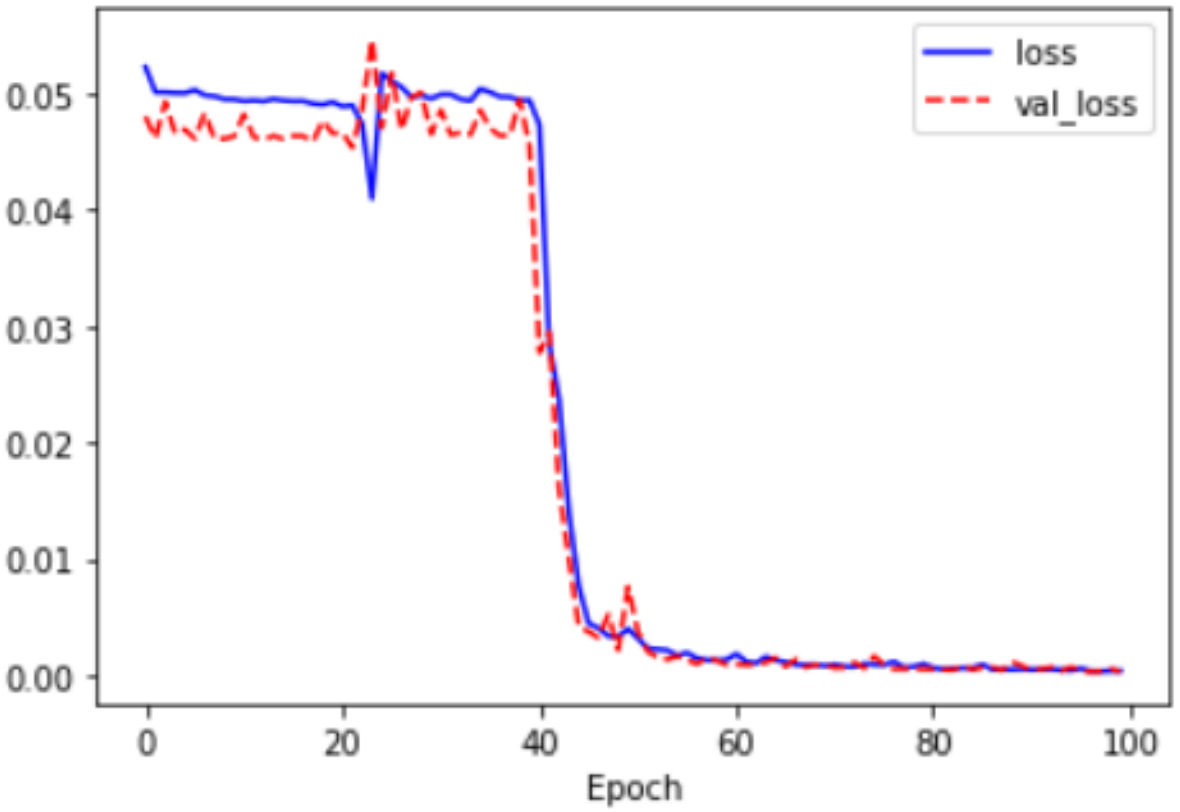
LSTM 모델

```
[ ]   1 model = tf.keras.Sequential([
      2   tf.keras.layers.LSTM(units=30, return_sequences=True, input_shape=[100,2]),
      3   tf.keras.layers.LSTM(units=30),
      4   tf.keras.layers.Dense(1)
      5 ])
      6
      7 model.compile(optimizer='adam', loss='mse')
      8 model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 100, 30)           3960
_________________________________________________________________
lstm_1 (LSTM)                (None, 30)                7320
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 31
=================================================================
Total params: 11,311
Trainable params: 11,311
Non-trainable params: 0
_________________________________________________________________
```

학습과 시각화

- **loss와 val_loss는 40 에포크를 넘어가면서 매우 가파르게 줌**
  - val_loss는 변동폭이 loss보다 크지만 전체적으로는 계속 감소하는 경향



예측

- **테스트 데이터에 대한 loss는 0에 가까운 값**
  - 다섯 개의 샘플에 대한 오차도 0.04를 넘는 값이 없으며
  - 정확도 역시 95.9%로 거의 96%에 가까운 것을 확인

```
[ ]  1 model.evaluate(X[2560:], Y[2560:])
     2 prediction=model.predict(X[2560:2560+5])
     3
     4 # 5개 테스트 데이터에 대한 예측을 표시합니다.
     5 for i in range(5):
     6   print(Y[2560+i], '₩t', prediction[i][0], '₩tdiff:', abs(prediction[i][0] - Y[2560+i]))
     7
     8 prediction = model.predict(X[2560:])
     9 fail = 0
    10 for i in range(len(prediction)):
    11   # 오차가 0.04 이상이면 오답입니다.
    12   if abs(prediction[i][0] - Y[2560+i]) > 0.04:
    13     fail +=1
    14
    15 print('correctness:', (440-fail)/440*100, '%')
```

```
14/14 [==============================] - 0s 13ms/step - loss: 3.9453e-04
0.009316712705671063    0.02192213    diff: 0.012605417432010898
0.1595728709352136      0.15154022    diff: 0.008032651151430648
0.343633615267837       0.33932722    diff: 0.004306399119460513
0.047836290850227836    0.03347582    diff: 0.014360470875090126
0.07471709800989841     0.06377047    diff: 0.01094662500651096
correctness: 95.9090909090909 %
```

7장 순환 신경망

3 GRU layers

.5h

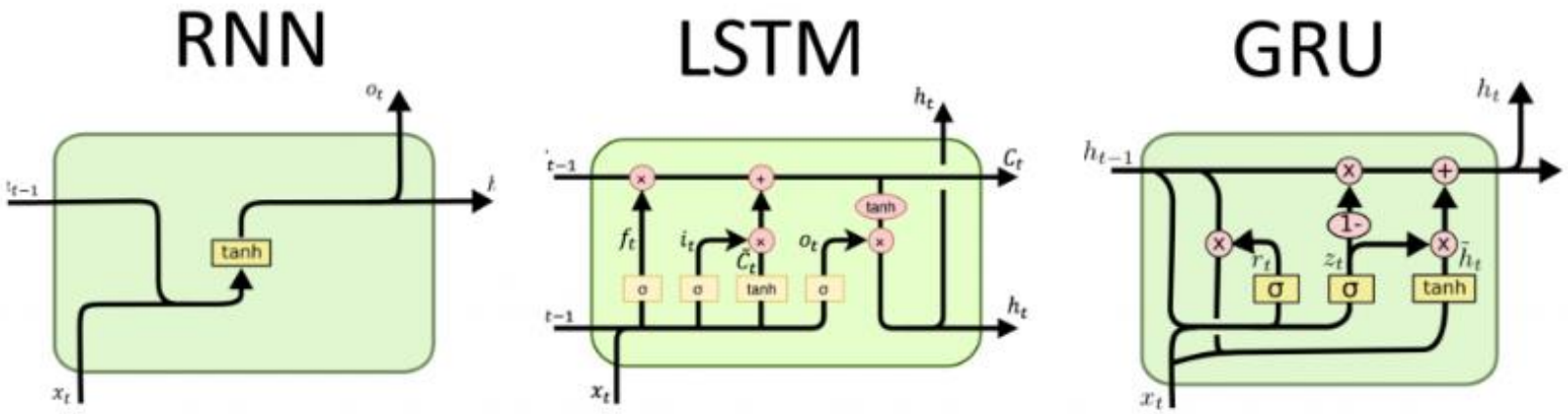
GRU(Gated Recurrent Unit) 레이어

- **LSTM과 비슷한 역할, 더 간단한 구조**
  - 특정 문제에서는 LSTM보다 더 적합한 레이어
  - 2014, 뉴욕 대학교의 조 경현 교수가 참여한 팀
    - **벤지오 교수의 제자**

https://www.sisain.co.kr/news/articleView.html?idxno=32570

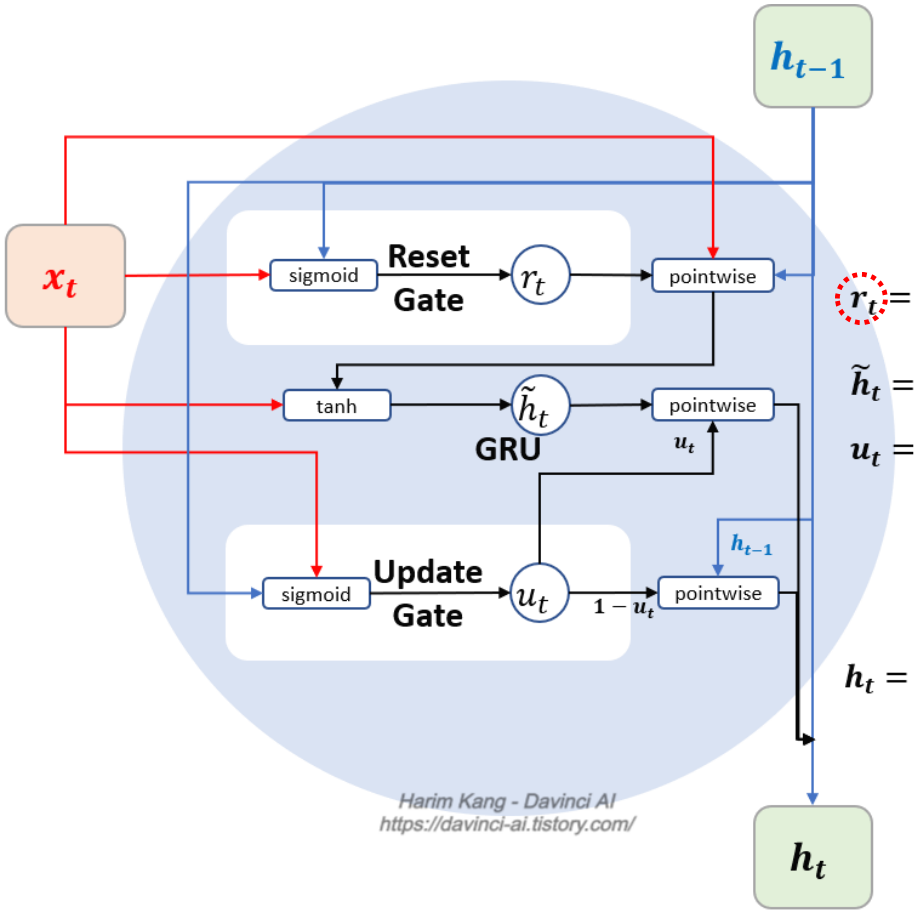
GRU 구조

- **GRU는 LSTM에서의 셀 상태(cell state) 역할을 하는 c가 없음**
  - cell state의 역할을 다음의 출력 h에서 그 역할을 함께 함
- **GRU에서는 Update Gate, Reset Gate 두 가지만 존재**
  - 활성화 함수는 sigmoid 2번과 tanh 1번 사용



GRU 장점

- **구조가 간단하고 문제에 따라 성능이 좋음**
  - 계산 상의 이점
  - LSTM와 비슷하거나 곱셈 문제 등의 일부 분야에서는 성능이 좋게 나타남



시그모이드 2번
tanh 1번

$$r_t = sigmoid(x_t U^r + h_{t-1} W^r)$$

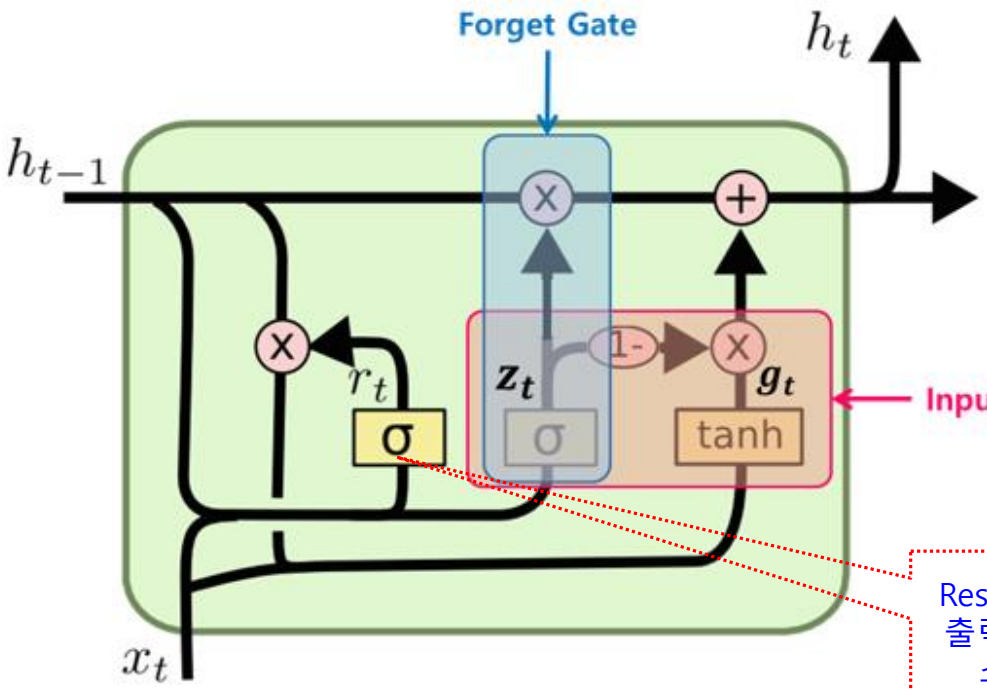
$$\tilde{h}_t = tanh(x_t U^{\tilde{h}} + r_t * h_{t-1} W^{\tilde{h}})$$

$$u_t = sigmoid(x_t U^u + h_{t-1} W^u)$$

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

GRU의 reset gate Rt, update gate Zt

- LSTM Cell에서의 두 상태 벡터 $\mathbf{c}_t$와 $\mathbf{h}_t$가 하나의 벡터 $\mathbf{h}_t$로 합쳐졌다.
- 하나의 gate controller인 $\mathbf{z}_t$가 **forget**과 **input** 게이트(gate)를 모두 제어한다. $\mathbf{z}_t$가 `1`을 출력하면 forget 게이트가 열리고 input 게이트가 닫히며, $\mathbf{z}_t$가 `0`일 경우 반대로 forget 게이트가 닫히고 input 게이트가 열린다. 즉, 이전$(t-1)$의 기억이 저장 될때 마다 타임 스텝 t의 입력은 삭제된다.
- GRU 셀은 output 게이트가 없어 전체 상태 벡터 $\mathbf{h}_t$가 타임 스텝마다 출력되며, 이전 상태 $\mathbf{h}_{t-1}$의 어느 부분이 출력될지 제어하는 새로운 gate controller인 $\mathbf{r}_t$가 있다.



Reset gate

$$\mathbf{r}_t = \sigma\left(\mathbf{W}_{xr}^T \cdot \mathbf{x}_t + \mathbf{W}_{hr}^T \cdot \mathbf{h}_{t-1} + \mathbf{b}_r\right)$$

$$\mathbf{z}_t = \sigma\left(\mathbf{W}_{xz}^T \cdot \mathbf{x}_t + \mathbf{W}_{hz}^T \cdot \mathbf{h}_{t-1} + \mathbf{b}_z\right)$$ update gate

$$\mathbf{g}_t = \tanh\left(\mathbf{W}_{xg}^T \cdot \mathbf{x}_t + \mathbf{W}_{hg}^T \cdot (\mathbf{r}_t \otimes \mathbf{h}_{t-1}) + \mathbf{b}_g\right)$$

$$\mathbf{h}_t = \mathbf{z}_t \otimes \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \otimes \mathbf{g}_t$$

tanh 를 통과한 gt와 이전 스텝의 출력인 ht-1은 zt 값에 따라 최종 출력에 어느 정도 비율을 점유될 지 결정

Reset gate를 통과한 rt는 이전 타임스텝의 출력인 ht-1에 곱해지기 때문에 이전 타임스텝의 정보를 얼마나 남길지를 결정

파일

- **ch7_2_RNN_study.ipynb**

마킹 곱셈 문제

- **데이터 생성**

```
X = []
Y = []
for i in range(3000):
  # 0 ~ 1 범위의 랜덤한 숫자 100개를 만듭니다.
  lst = np.random.rand(100)

  # 마킹할 숫자 2개의 인덱스를 뽑습니다.
  idx = np.random.choice(100, 2, replace=False)

  # 마킹 인덱스가 저장된 원-핫 인코딩 벡터를 만듭니다.
  zeros=np.zeros(100)
  zeros[idx]=1

  # 마킹 인덱스와 랜덤한 숫자를 합쳐서 X에 저장합니다.
  X.append(np.array(list(zip(zeros, lst))))
  # 마킹 인덱스가 1인 값만 서로 곱해서 Y에 저장합니다.
  Y.append(np.prod(lst[idx]))
```

모델 생성

- **GRU 층**
  - 패러미터 수가 LSTM보다 23.3% 감소

```
1 model = tf.keras.Sequential([
2   tf.keras.layers.GRU(units=30, return_sequences=True, input_shape=[100,2]),
3   tf.keras.layers.GRU(units=30),
4   tf.keras.layers.Dense(1)
5 ])
6
7 model.compile(optimizer='adam', loss='mse')
8 model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru (GRU)                    (None, 100, 30)           3060
_________________________________________________________________
gru_1 (GRU)                  (None, 30)                5580
_________________________________________________________________
dense (Dense)                (None, 1)                 31
=================================================================
Total params: 8,671
Trainable params: 8,671
Non-trainable params: 0
_________________________________________________________________
```

SimpleRNN	LSTM	GRU
2,851	11,311	8,671

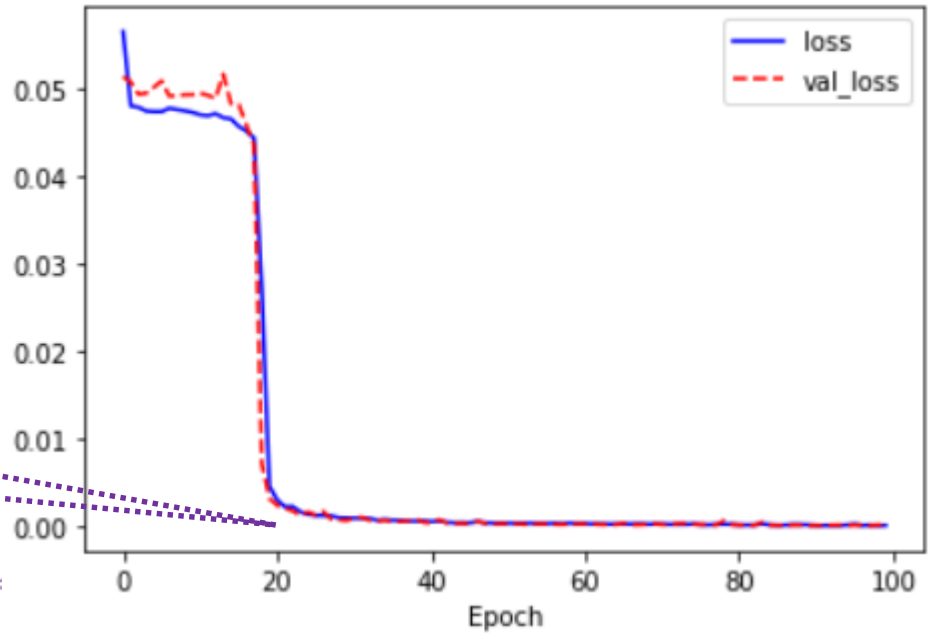
학습과 시각화

```
X = np.array(X)
Y = np.array(Y)

# 모형 학습
history = model.fit(X[:2560], Y[:2560], epochs=100, validation_split=0
.2)

# 모형 학습 시각화
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```

20 에폭에서 값이 줄어들고 안정적, LSTM보다 결과가 매우 좋음



예측

- **LSTM 레이어보다 GRU레이어로 더 잘 풀리는 문제**
  - 정확도는 99.3%로 거의 99%에 가까운 값

```
# 모형 테스트 및 결과
model.evaluate(X[2560:], Y[2560:])
prediction=model.predict(X[2560:2560+5])

# 5개 테스트 데이터에 대한 예측을 표시합니다.
for i in range(5):
  print(Y[2560+i], '₩t', prediction[i][0], '₩tdiff:', abs(prediction[i][0] - Y[2560+i]))

prediction = model.predict(X[2560:])
fail = 0
for i in range(len(prediction)):
  # 오차가 0.04 이상이면 오답입니다.
  if abs(prediction[i][0] - Y[2560+i]) > 0.04:
    fail +=1

print('correctness:', (440-fail)/440*100, '%')
```

```
14/14 [==============================] - 0s 13ms/step - loss: 1.3958e-04
0.025460655274426407    0.024673868    diff: 0.000786787303721375
0.3193748845131831      0.3263419      diff: 0.006967012736038725
0.12277371727313707     0.124509096    diff: 0.0017353788724928099
0.3530137775509947      0.34941977     diff: 0.0035940049260252405
0.08399048857061454     0.08783957     diff: 0.003849085051135334
correctness: 99.31818181818181 %
```