8강 순환신경망

박지훈

목차

1. FeedForwardNetwork vs RecurrentNetwork
2. 순환신경망
3. 케라스의 순환층
4. IMDB데이터 적용
5. 시각화, 결과
6. LSTM
7. GRU
8. Reuters데이터
9. 모델 비교

1. FeedForwardNetwork vs RecurrentNetwork

- FeedForwardNetwork란?

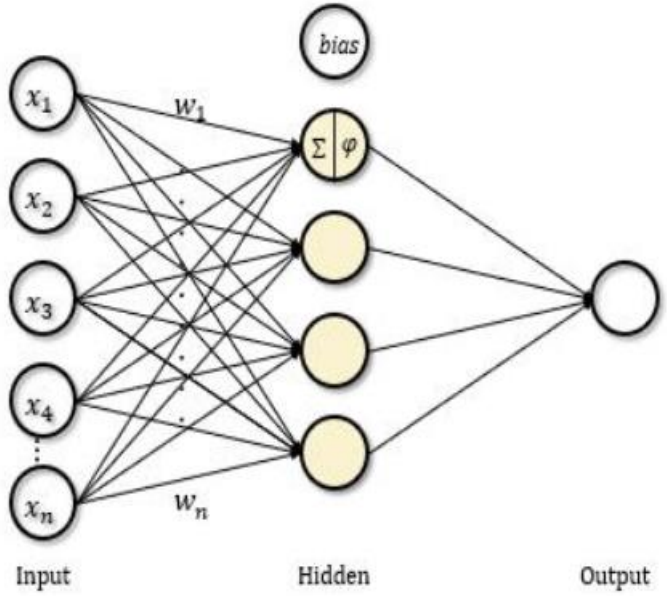
→ 단위 간의 연결을 형성하지 않은 인공신경망

재귀신경네트워크를 갖춘 RNN과 다르다.

고안된 인공신경망의 최초이자 가장 단순한 형태.

이 네트워크에서 정보는 입력노드에서 hiddenlayer를 통해 **한방향으로만** 앞으로 이동한다. 출력노드, 네트워크에 순환이나 루프가 없다.

일반적으로 FFNN은 입력층-은닉층-출력층을 거치는 모델 아키텍쳐를 가짐



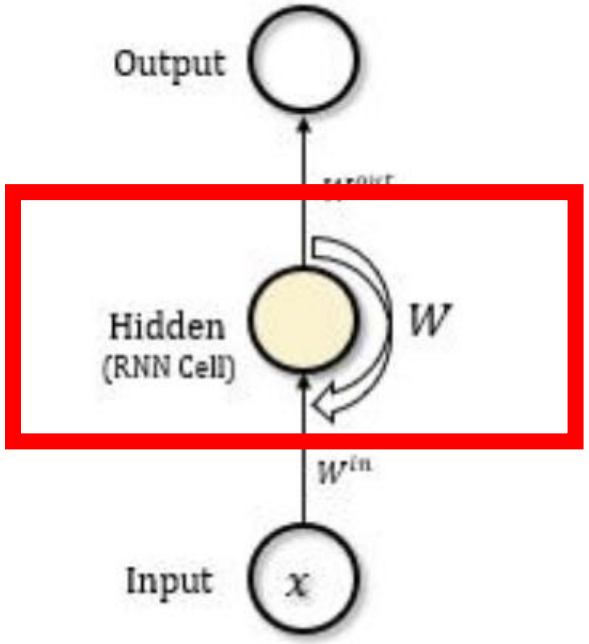
일반적으로 우리가 아는 ANN모형이라고 생각하면 됨.

1. FeedForwardNetwork vs RecurrentNetwork

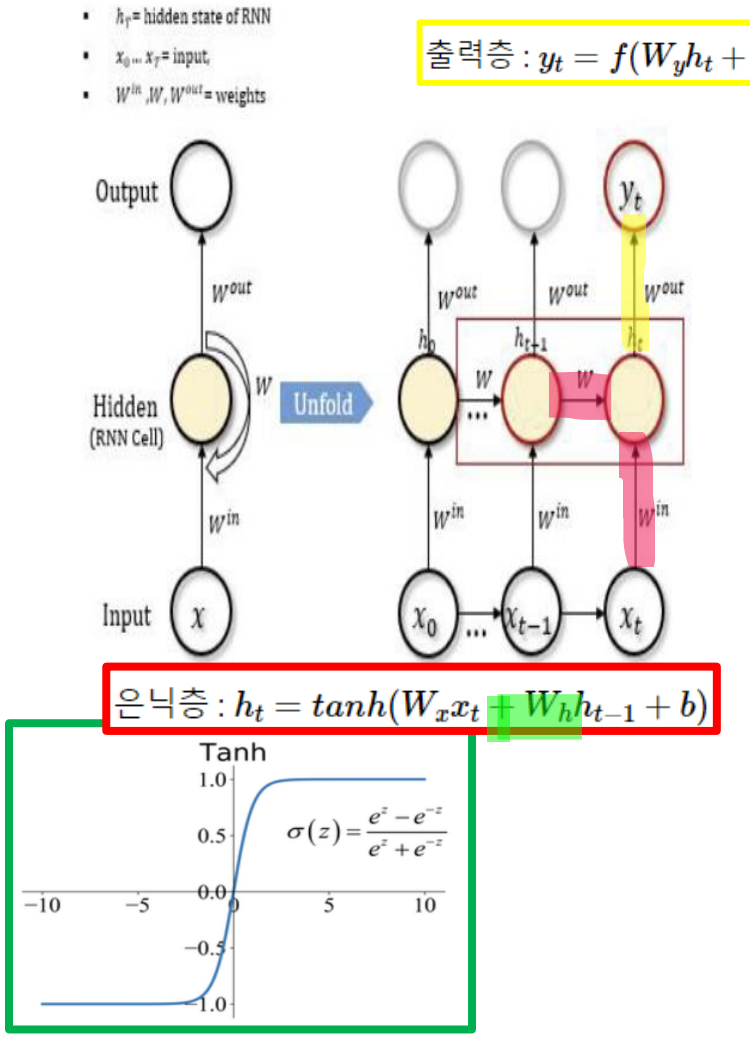
- RNN이란?

재귀신경망(RNN)은 연결이 시퀀스를 따라 방향성 그래프를 형성하는 인공신경망의 클래스이다.

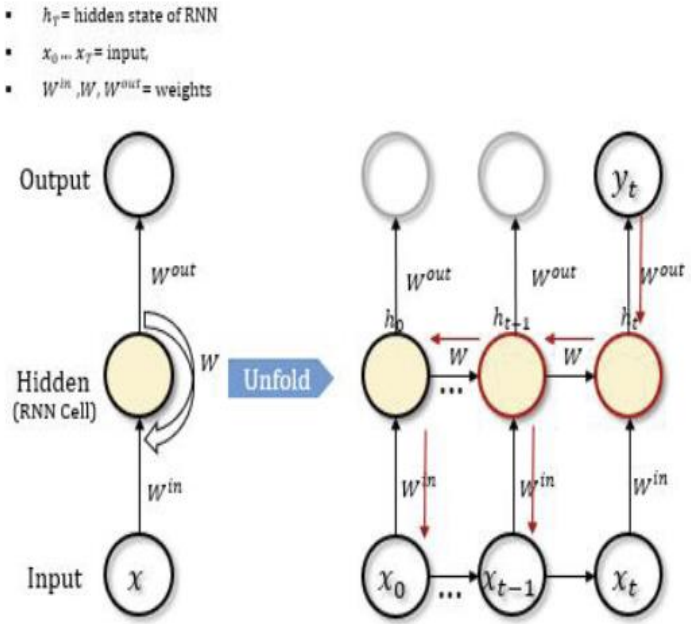
- 시간에 대한 동적 시간 동작을 나타내기 용이함.
- 기존 FFNN같이 입력층-은닉층-출력층을 가지고 있으나, **은닉층이 순환구조를 가지며, 동일한 가중치를 공유한다는 점에서 차이가 있다.**
- 일반적은 ANN과 달리, **순서, 시간 등을 가중치로 줄 수 있다.(Sequential data에 잘 쓰인다.)**



RNN을 더 알아보자.

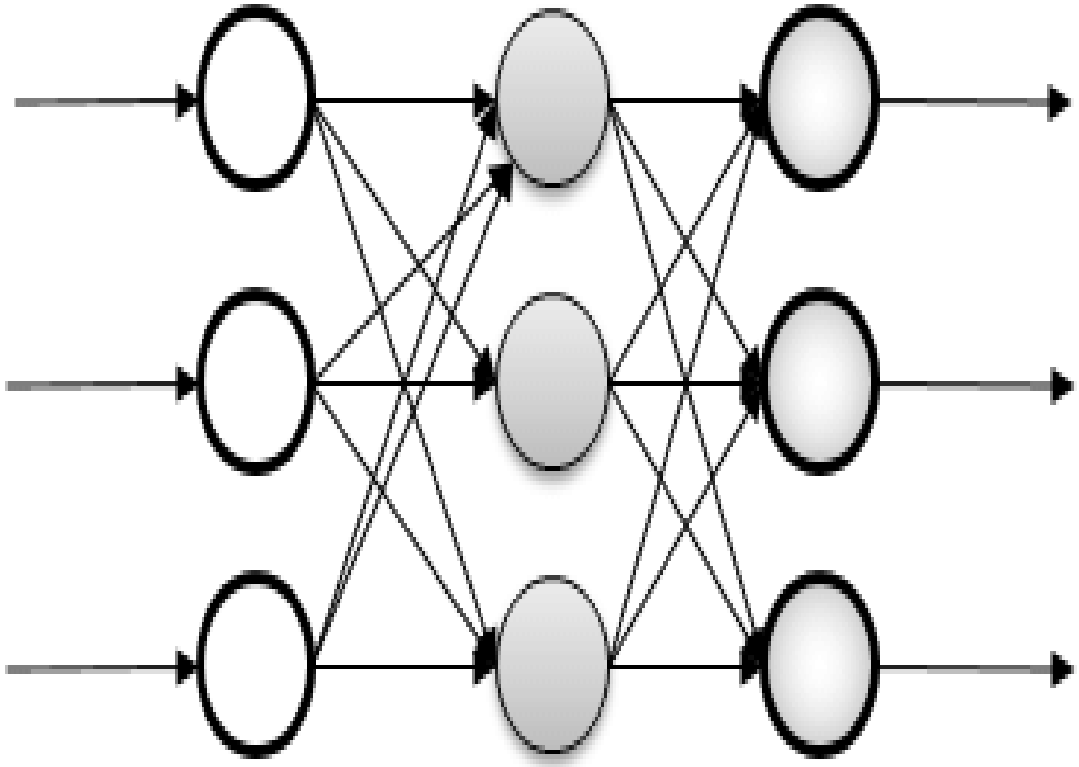


- 좌우 그림은 RNN을 시간방향으로 전개한 모습이다.
- Hidden층은 이전 hidden층의 가중치와 입력층을 함께 받아서 결과를 도출한다. (이를 반복)
- 각 시점 x(t)마다 층이 별개인것으로 간주하고 FFNN과 동일하게 역전파를 수행한다.



역전파법

Feedforward neural network

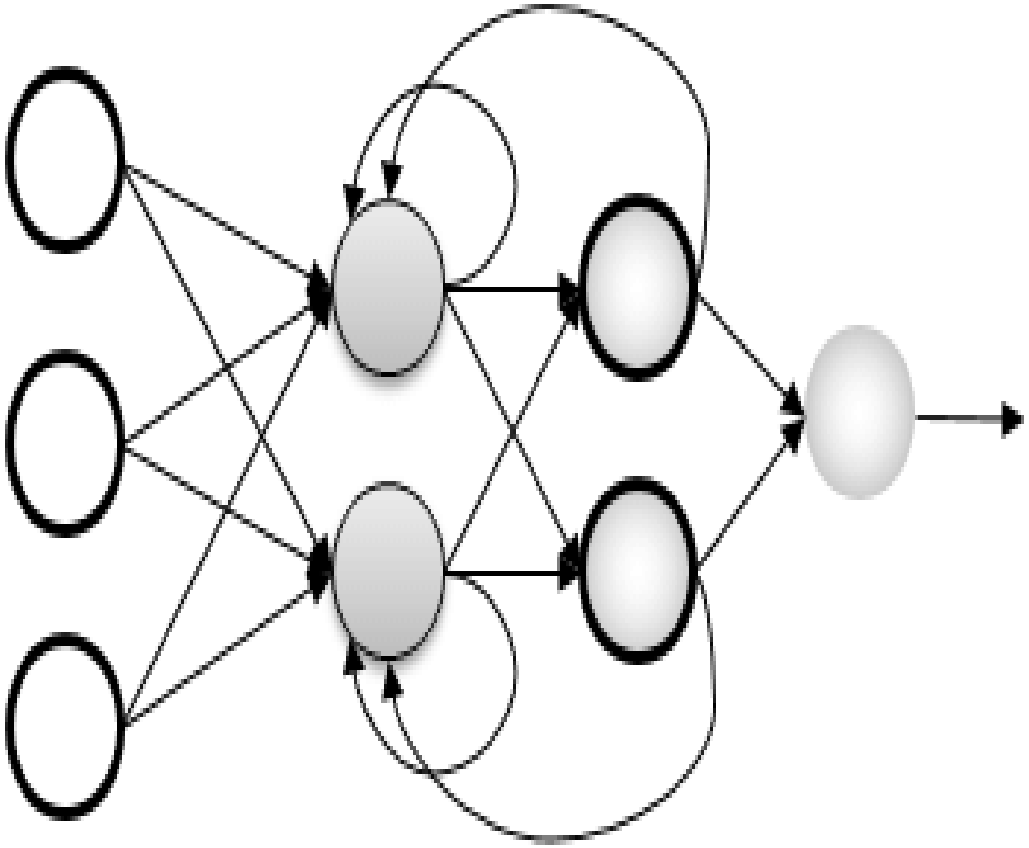


Input Layer

Hidden Layer

Output Layer

Recurrent neural network



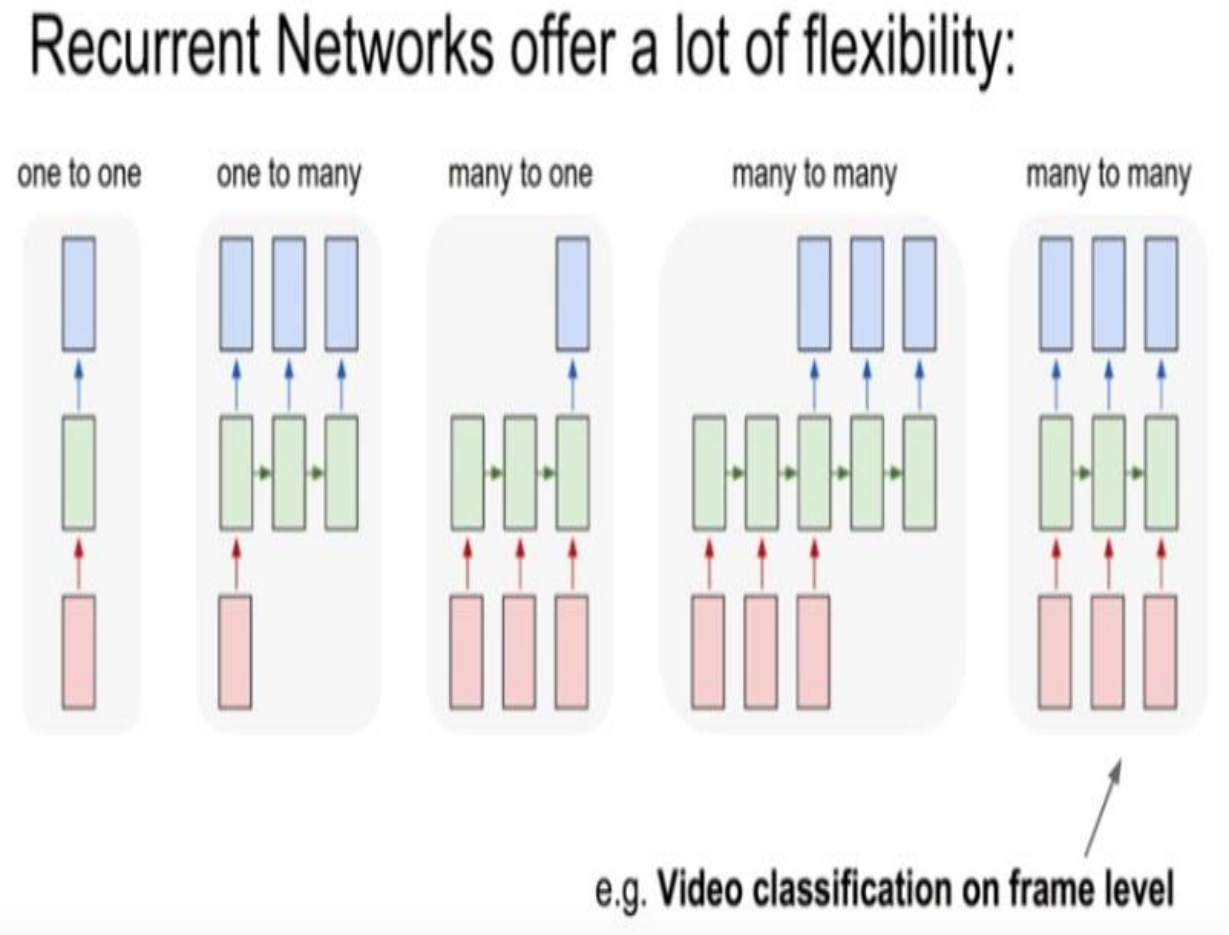
Input Layer

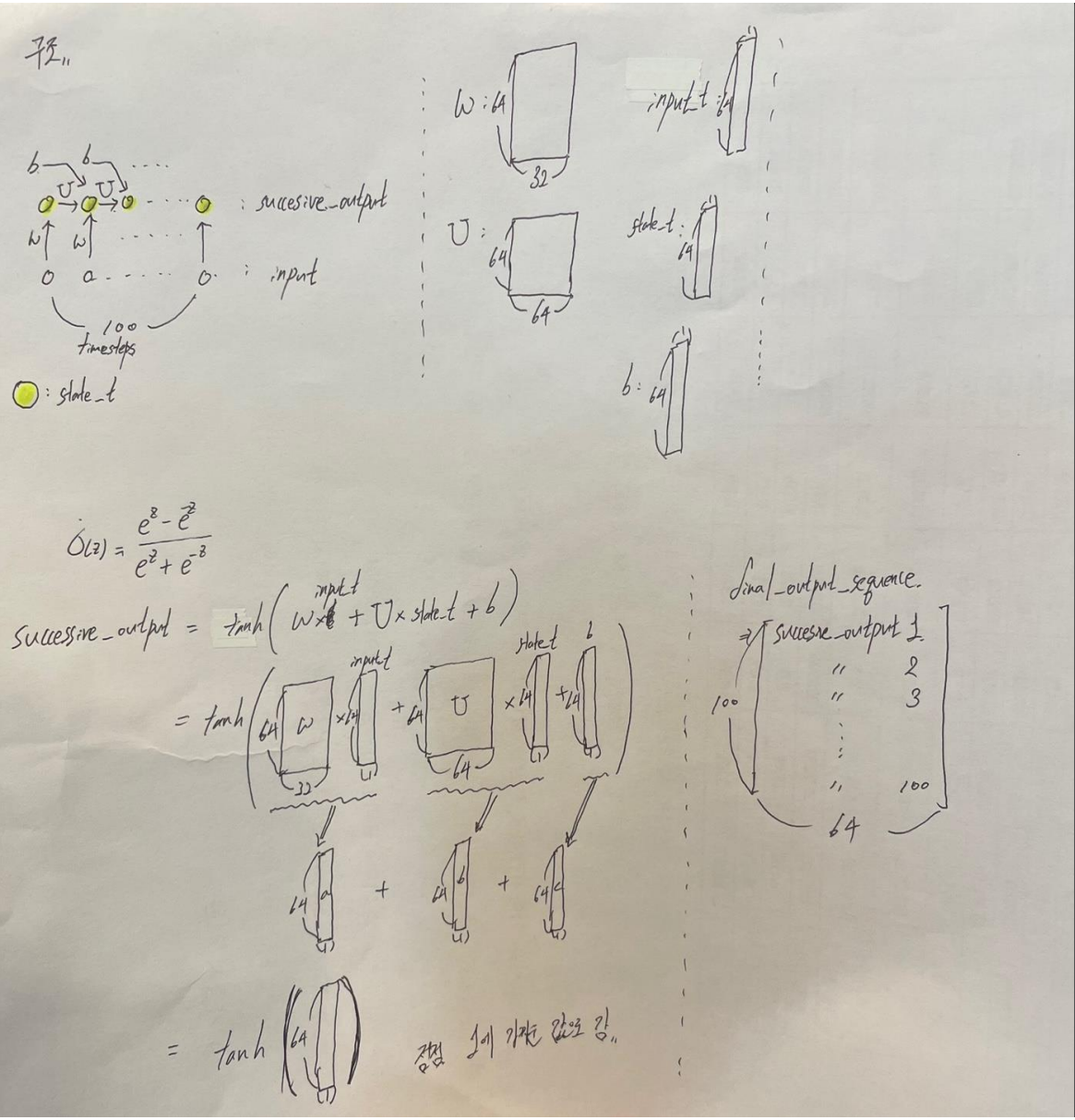
Hidden Layer

Output Layer

순환신경망의 다양한 구조

- RNN의 강력한 기능 중 하나는 입출력의 길이를 다르게 설계할 수 있다.
- One-to-many : 주로image-captioning(하나의 이미지에 대해서 이미지에 대한 설명을 만듬.)
- Many-to-one : 주로 감성분류, 스팸 메일 분류 등에 사용
- Many-to-many : 주로 챗봇, 번역기 등에 사용됨.






```python
timesteps = 100
input_features = 32
output_features = 64

inputs = np.random.random((timesteps, input_features))
print('inputs shape : ', inputs.shape)

state_t = np.zeros((output_features, )) ##상태값 초기화
print('state_t shape : ', state_t.shape)

W = np.random.random((output_features, input_features)) ##input이랑 만나는 가중치치
print('W shape : ', W.shape)
U = np.random.random((output_features, output_features)) ##전 hidden에서 현재 hidden이랑 만나는 가중치
print('U shape : ', U.shape)
b = np.random.random((output_features, ))

succesive_outputs = []

for input_t in inputs : ##input_t가 32개씩 100번 시행됨.
  output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
  succesive_outputs.append(output_t)
  state_t = output_t

final_output_sequence = np.stack(succesive_outputs, axis=0)
print(final_output_sequence[:2])
print('len of final_output_sequence :', final_output_sequence.shape)
```

Tensorflow keras를 통한 구현

IMDB데이터

- IMDB데이터는 영화리뷰에 대한 데이터 50000개로 구성
- 이 중 25,000개의 훈련데이터와 25,000개의 테스트 데이터로 나뉨
- 각각 50%씩 긍정리뷰(1)와 부정리뷰(0)로 구성(label화)

- 이미 데이터는 전처리가 되어 있으며, 정수인코딩이 되어있음.

(사전에 있는 단어와 숫자를 매핑시켜 단어대신 숫자로 표현)

```
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence
```

```
(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=num_words)
print(input_train[0])
print(y_train[0])
## 부정은 0, 긍정은 1
```

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

IMDB데이터를 모델에 넣기 위한 pad_to_sequence

- 각각의 리스트의 길이가 천차만별이다.
- 이것을 모델에 넣기 위해선 같은 크기의 리스트로 만들어주어야한다.(pad_to_sequence)

★Imdb.load_data(**numwords**)

- 훈련데이터에서 가장 자주 사용하는 단어 1만개만 사용하겠다는 의미
- 사용 빈도수가 10000등 안에 들지 못하는 데이터는 training을 시키지 않음
- 이런 과정을 통해 적절한 크기의 벡터 데이터를 얻을 수 있음

우리는 빠른 결과를 위해 500개의 단어만 사용하자. (pad_to_sequence=500)

즉, 각 리스트의 길이는 500.

```
num_words = 10000
max_len = 500
batch_size = 32

(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=num_words)
# 훈련데이터에서 가장 자주 사용하는 단어 1만개만 사용하겠다는 의미
# 사용 빈도수가 10000등 안에 들지 못하는 데이터는 training을 시키지 않음
# 이런 과정을 통해 적절한 크기의 벡터 데이터를 얻을 수 있음
print(len(input_train))
print(len(input_test))
print(input_train[0])

input_train = sequence.pad_sequences(input_train, maxlen=max_len)
input_test = sequence.pad_sequences(input_test, maxlen=max_len)
print(input_train.shape) #패딩해서 500개 더 생김
print(input_test.shape) #패딩해서 500개 더 생김
```

```
25000
25000
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256,
(25000, 500)
(25000, 500)
```

모델구성(1)

```
from tensorflow.keras.layers import SimpleRNN, Embedding
from tensorflow.keras.models import Sequential
```

- 우측사진 예시
- 케라스에서 제공하는 SimpleRNN을 사용

- Return_sequence
  - False : 층을 더 쌓지 못함
  - True : 다음 층으로 넘겨줌; 모델을 쌓는게 가능

```
model = Sequential()
model.add(Embedding(10000, 32)) #input:10000, output32
model.add(SimpleRNN(32, return_sequences=True)) #input:32, output:32
model.add(SimpleRNN(32, return_sequences=True)) #input:32, output:32
model.add(SimpleRNN(32, return_sequences=True)) #input:32, output:32
model.add(SimpleRNN(32, return_sequences=True)) #input:32, output:32
model.add(SimpleRNN(32, return_sequences=True)) ##값을 sequence로 넘겨줄거냐 말거냐
model.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_9 (Embedding)     (None, None, 32)          320000

 simple_rnn_15 (SimpleRNN)   (None, None, 32)          2080

 simple_rnn_16 (SimpleRNN)   (None, None, 32)          2080

 simple_rnn_17 (SimpleRNN)   (None, None, 32)          2080

 simple_rnn_18 (SimpleRNN)   (None, None, 32)          2080

 simple_rnn_19 (SimpleRNN)   (None, None, 32)          2080

=================================================================
Total params: 330,400
Trainable params: 330,400
Non-trainable params: 0
_________________________________________________________________
```

모델구성(2)

- 목적 : test데이터가 긍정리뷰인지 부정리뷰인지를 분류

- optimier : rmsprop
- Activation func : sigmoid(0/1)
- Loss : binary_crossentropy
- Metric : acc
- Epoch : 10
- Batch_size : 128
- Validation_split : 0.2

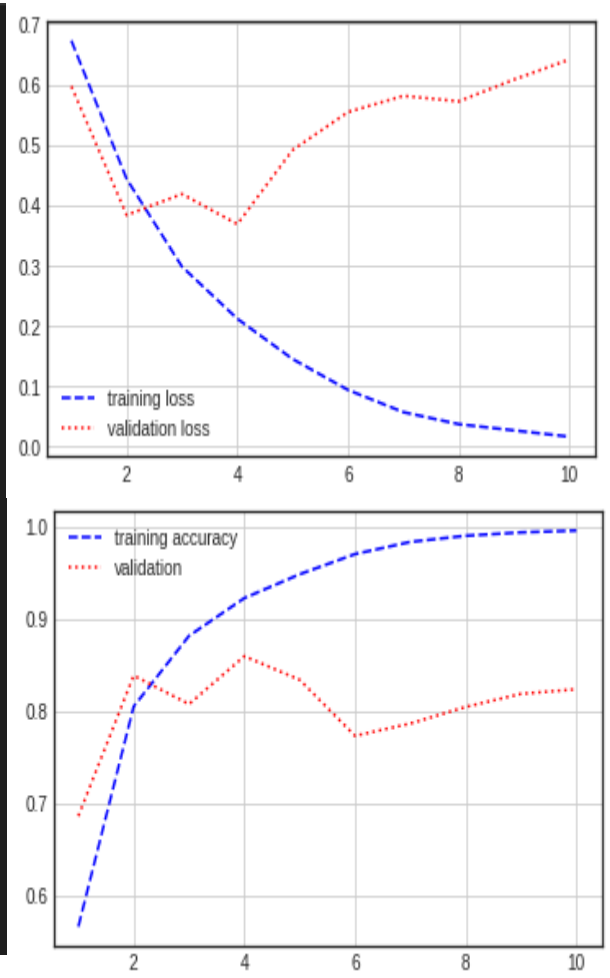
```
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

```
from tensorflow.keras.layers import Dense

model = Sequential()

model.add(Embedding(num_words, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss = 'binary_crossentropy',
              metrics=['acc'])
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, None, 32)          320000

 simple_rnn_7 (SimpleRNN)    (None, 32)                2080

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 322,113
Trainable params: 322,113
Non-trainable params: 0
_________________________________________________________________
```

결과 시각화, 평가

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')

loss = history.history['loss']
val_loss = history.history['val_loss']
acc = history.history['acc']
val_acc = history.history['val_acc']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'b--', label='training loss')
plt.plot(epochs, val_loss, 'r:', label='validation loss')
plt.grid()
plt.legend()

plt.figure()
plt.plot(epochs, acc, 'b--', label='training accuracy')
plt.plot(epochs, val_acc, 'r:', label='validation')
plt.grid()
plt.legend()
```




```
model.evaluate(input_test, y_test)
```

```
782/782 [==============================] - 29s 37ms/step - loss: 0.6556 - acc: 0.8215
[0.655566394329071, 0.8215199708938599]
```

Loss : 0.6556
Acc : 0.8215

- 우리는 500개의 단어만을 이용했기때문에, 성능이 좋지 않음.
- SimpleRNN은 실전에서 사용하기엔 너무 단순하다
- 또한, 긴 시퀀스를 처리하는데 적합하지 않다.
- SimpleRNN은 모든 타임스템의 정보를 유지할 수 있지만, 실제로 긴 시간에 걸친 의존성은 학습할 수 없음.(Vanishing Gradient Probelm)

RNN의 단점

1. 짧은 시퀀스에 대해서만 효과를 보임
(**Long Term Dependencies**)

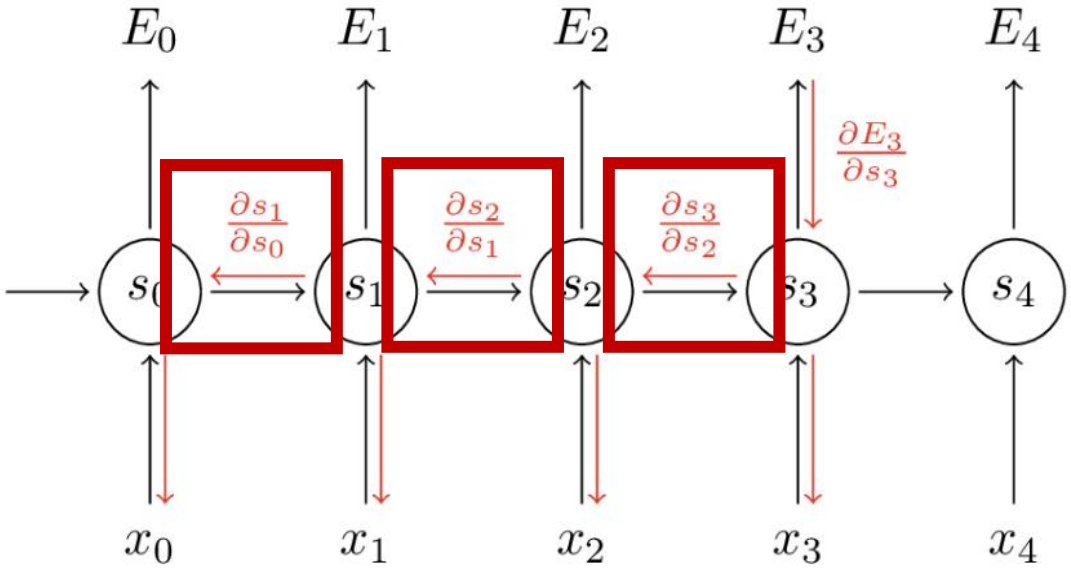
- 시점이 길어질수록 앞의 정보가 뒤로 충분히 전달되지 못함
- 가장 중요한 단어가 앞쪽에 위치한 경우, RNN이 충분한 기억력을 갖지 못함.

Ex)I grew up in France and want to be a plumber who the best is in the world and I speak fluent **French**

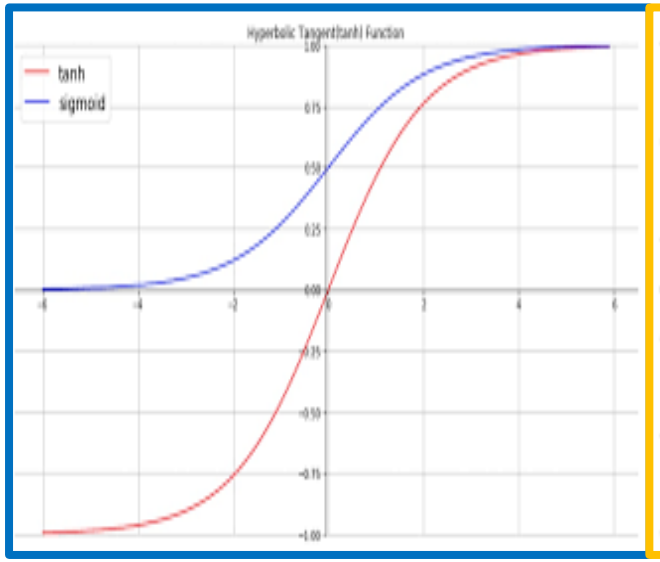
안타깝게도 (Bengio et al..,1994)논문에 따르면, RNN에서는 이 문제를 해결하지 못함

- (Vanishing Gradient Probelm)

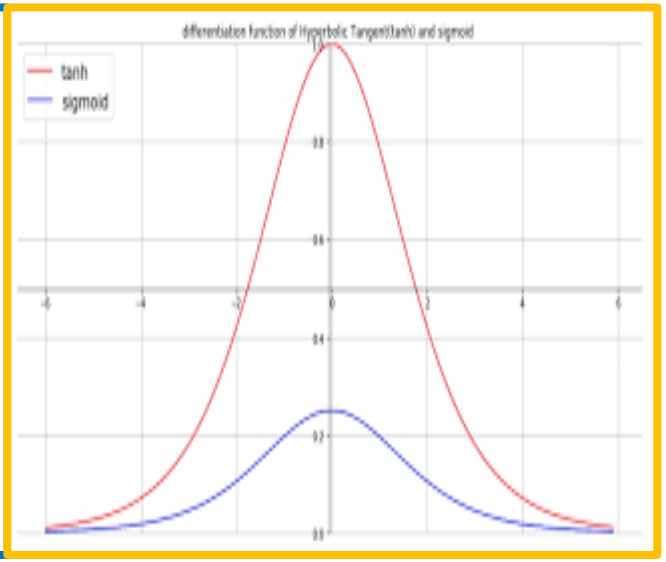
해결을 위해 LSTM 등장



Sigmoid가 역전파 과정에서 기울기소실되는 문제를 딥러닝1에서 배웠을것이다.



Sigmoid와 tanh

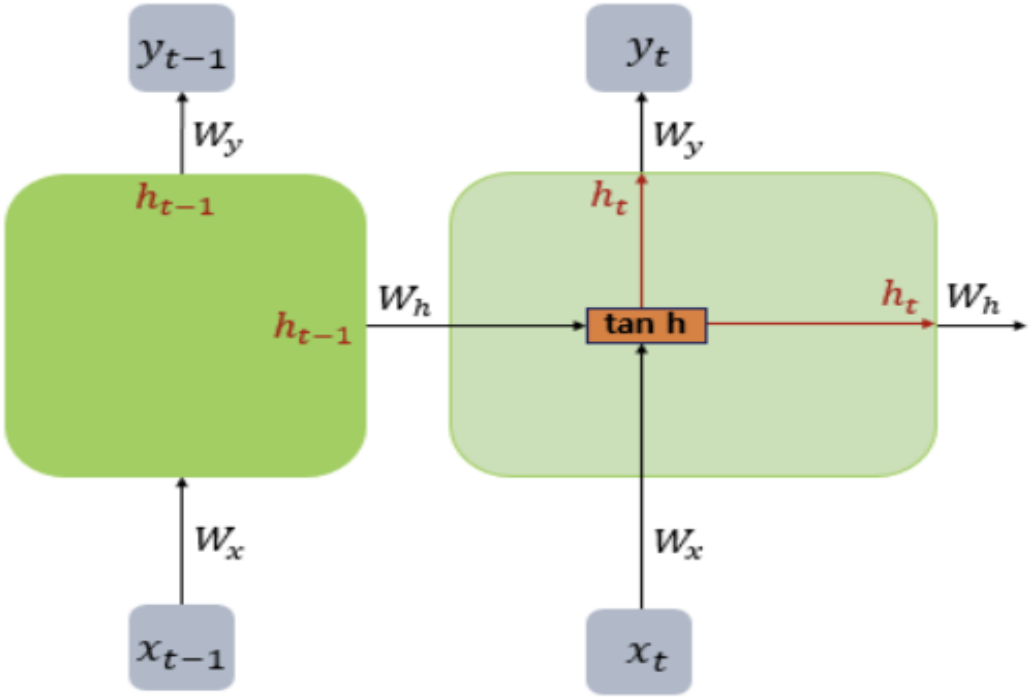


Sigmoid와 tanh기울기

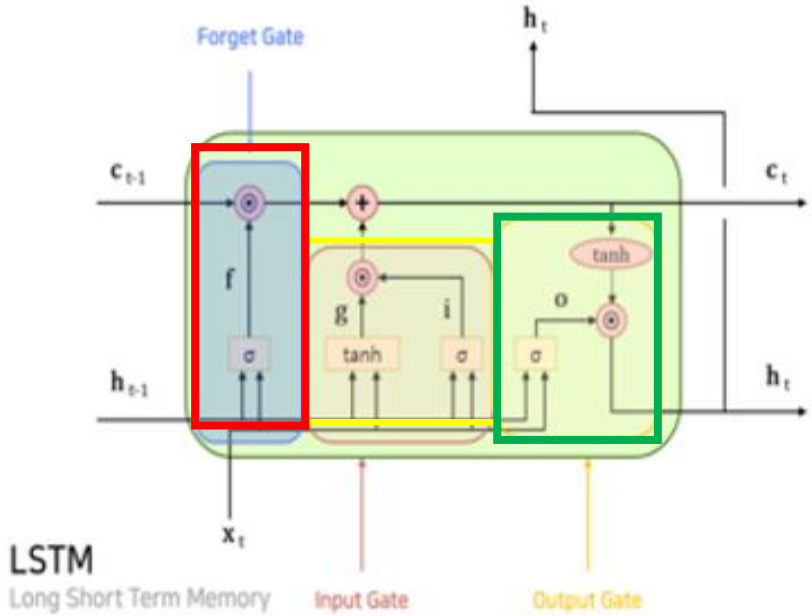
LSTM
(long short-term memory)

LSTM(1)

RNN의 내부 모습
- 단순, 심심



Lstm의 내부 모습
- 입력, 망각, 출력게이트 추가
(불필요한 기억을 지우고, 기억해야 할 것을 정함)



LSTM(2-입력게이트)

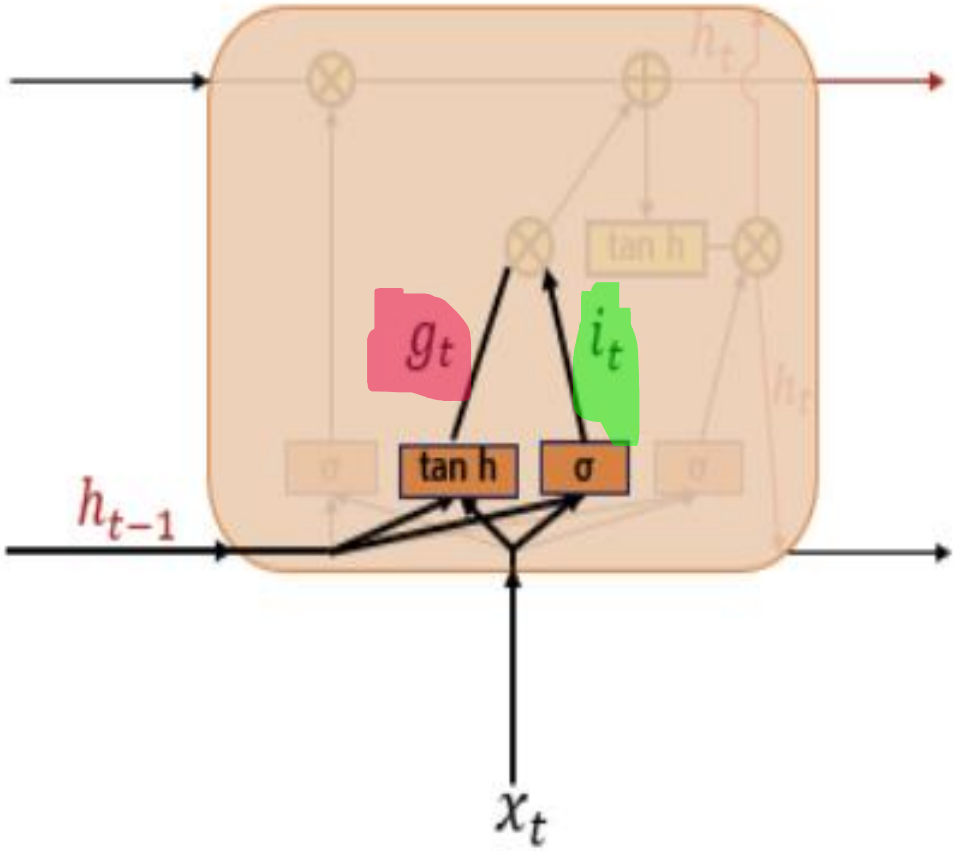
- 현재정보를 기억하기 위한 게이트

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

$$g_t = tanh(W_{xg}x_t + W_{hg}h_{t-1} + b_g)$$

- it : 0~1
- gt : -1~1

이 두 값을 종합하여 현재정보를 기억한다.

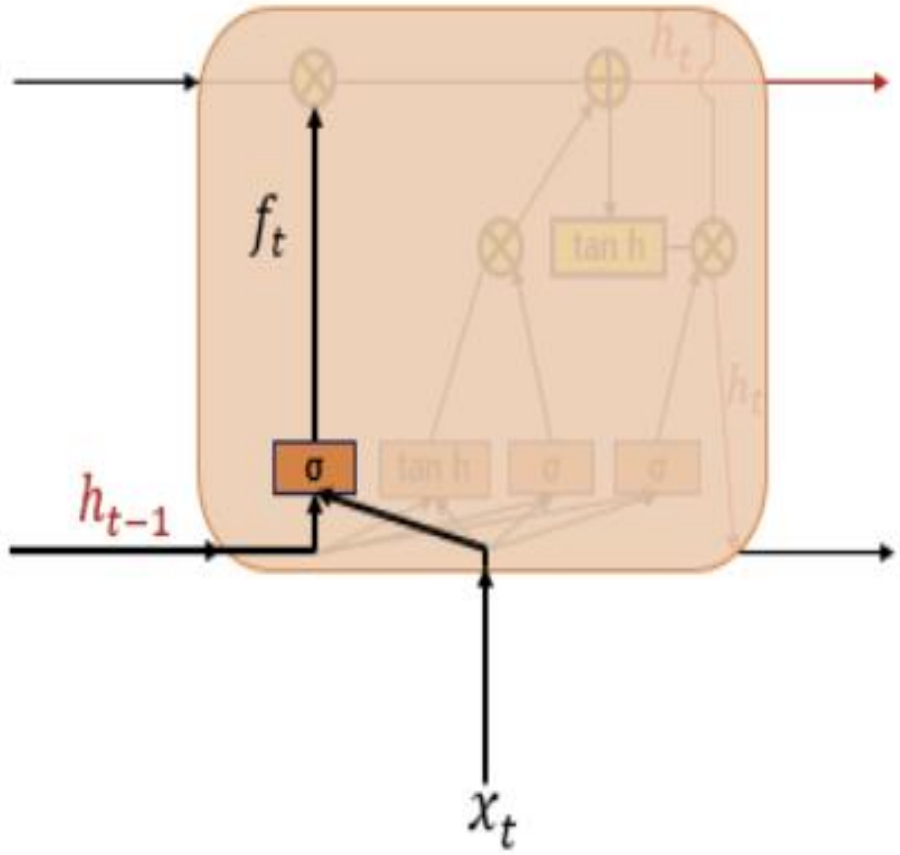


LSTM(3-망각게이트)

- 기억을 삭제하기 위한 게이트

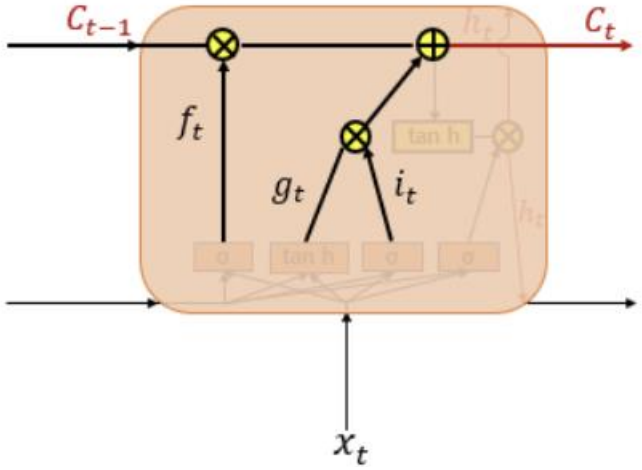
$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

- 0에 가까울수록 정보가 많이 삭제된 것이고
  1에 가까울수록 정보를 온전히 기억한 것



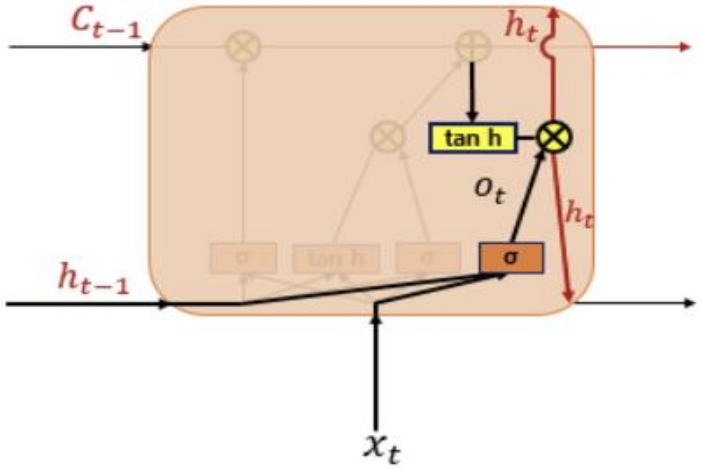
LSTM(4-출력게이트)

- 현재 셀상태



$$C_t = f_t \circ C_{t-1} + i_t \circ g_t$$

- 출력게이트

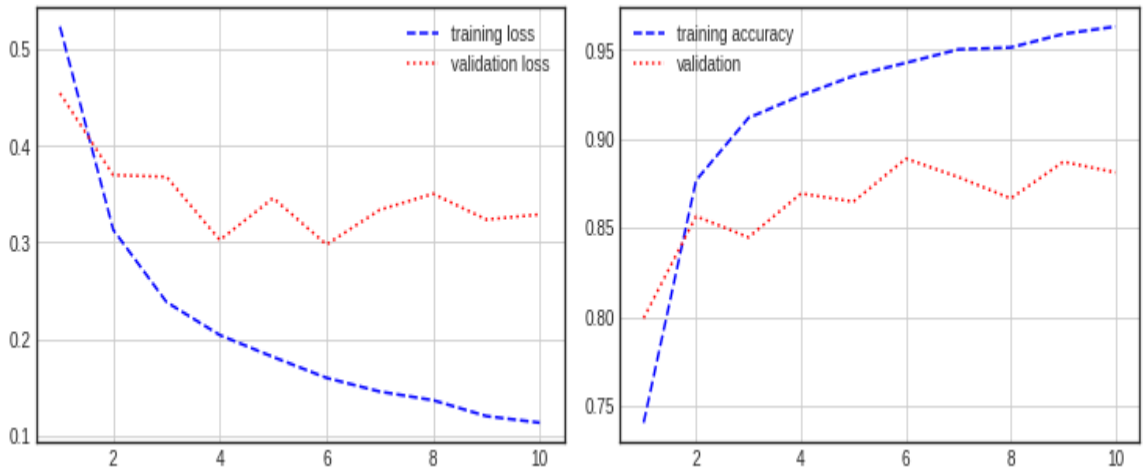


$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$
$$h_t = o_t \circ tanh(c_t)$$

LSTM구현

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, GRU, Embedding

##RNN대신에 LSTM으로 바꾸면 어떤 차이가 있을까
## 모델 구성
model = Sequential()

model.add(Embedding(num_words, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
model.summary()
```

```
Model: "sequential_12"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 embedding_12 (Embedding)     (None, None, 32)          320000

 lstm (LSTM)                  (None, 32)                8320

 dense_1 (Dense)              (None, 1)                 33

=================================================================
Total params: 328,353
Trainable params: 328,353
Non-trainable params: 0
_________________________________________________________________
```



loss

acc

```
model.evaluate(pad_x_test, y_test)
```

```
782/782 [==============================] - 7s 8ms/step - loss: 0.4527 - acc: 0.8628
[0.45266687870025635, 0.8628000020980835]
```

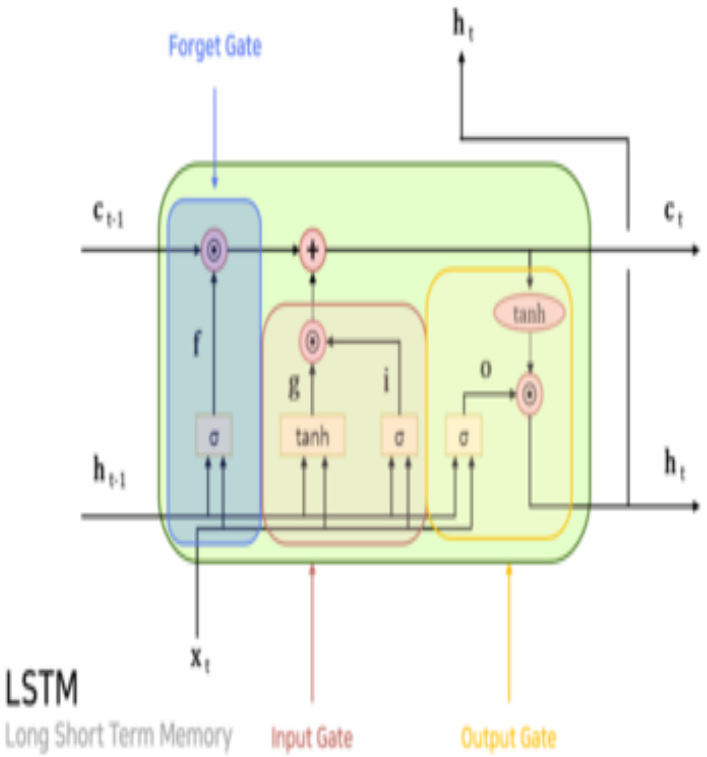
- RNN(82%)보다 성능좋음.(LSTM-86%)
- RNN보다 시간이 빠름

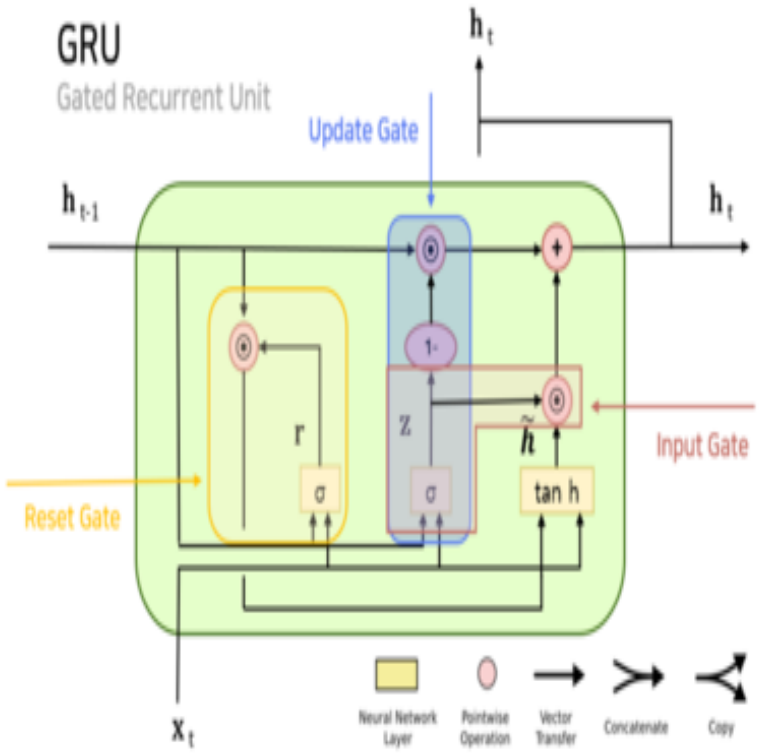
단점
- 너무 복잡(**파라미터의 수가 RNN보다 6000개 가량 늘어났음. RNN은 322,113개**)
- 매개변수가 많아서 너무 오래걸림
- 너무 어려움. 에바
- 해결하기 위해 **GRU**사용

GRU

Gated Recurent Unit

GRU LSTM비교



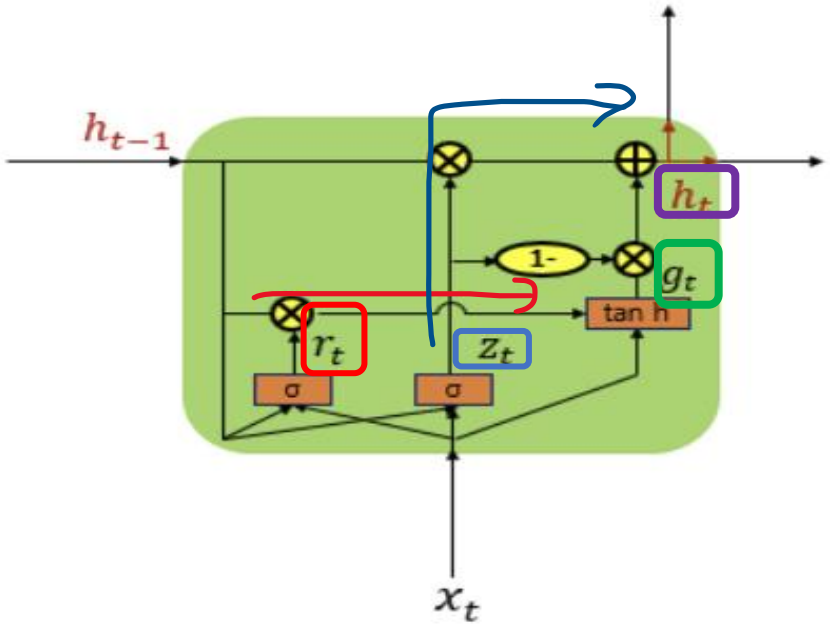


- LSTM에 비해 비교적 GRU가 덜 복잡하다.

- GRU는 LSTM의 게이트를 사용한다는 개념을 유지한채, 매개변수를 줄여 계산과 시간을 줄여준다.

- 여담으로 GRU는 대한민국 '조경현' 박사님이 제안한 모델이다.

GRU(1)



- LSTM과 다르게 GRU는 reset gate, update gate 총 2가지로 이루어져 있다.
- LSTM에 비해 학습속도가 2배정도 빠르고, 비슷한 성능을 보인다고 알려져있다.
- 기존의 사례들로 미루어보아, 매개변수의 양이 적을때, GRU가 조금 더 낫고, 데이터양이 많으면 LSTM이 낫다고 알려져 있다.

- Reset gate : $r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$ 이전 hidden state의 값을 얼마나 활용할 것인지에 대한 정보

- Update gate : $z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$ LSTM의 입력게이트와 망각게이트를 합침

- 재귀함수 : $g_t = tanh(W_{hg}(r_t \circ h_{t-1}) + W_{xg}x_t + b_g)$

- 출력함수 : $h_t = (1 - z_t) \circ g_t + z_t \circ h_{t-1}$

GRU구현 with Reuters데이터(1)

- 데이터 설명

Reuters데이터

-> IMDB와 같이 뉴스데이터지만, 클래스가 46개나 있다.

-> 정수인코딩이 이루어져 있다.

-> 클래스가 많으니, IMDB보다 많은 데이터를 사용할 필요가 있다.

```
from tensorflow.keras.datasets import reuters

num_words=10000
(x_train, y_train), (x_test, y_test) = reuters.load_data(num_words=num_words)

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

```
(8982,)
(2246,)
(8982,)
(2246,)
```

모델구성

- 앞에서 했던 LSTM과 유사하지만, 클래스가 많으므로 층을 한층 더 쌓고, 옵티마이저도 adam.

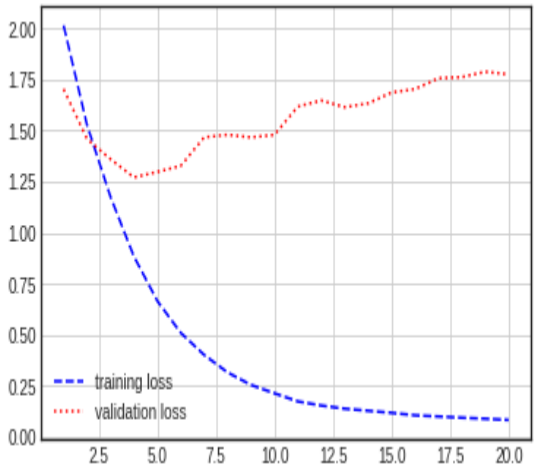
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dense, Embedding

model = Sequential()
model.add(Embedding(input_dim = num_words, output_dim = 256))
model.add(GRU(256, return_sequences=True))
model.add(GRU(128))
model.add(Dense(46, activation='softmax'))

model.compile(optimizer='adam',
              loss = 'sparse_categorical_crossentropy',
             metrics=['acc'])
model.summary()
```

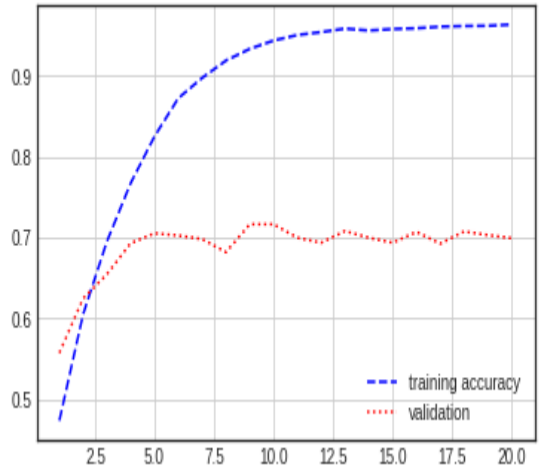
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, None, 256)         2560000

 gru_2 (GRU)                 (None, None, 256)         394752

 gru_3 (GRU)                 (None, 128)               148224

 dense_1 (Dense)             (None, 46)                5934

=================================================================
Total params: 3,108,910
Trainable params: 3,108,910
Non-trainable params: 0
_________________________________________________________________
```

학습 및 시각화

- Batch_size : 32
- Epochs : 20
- Validation_split : 0.2

```
history = model.fit(pad_x_train, y_train,
          batch_size=32,
          epochs=20,
          validation_split=0.2)
```





사실 GRU를 학습할땐, 다른 데이터셋을 사용했으므로, 앞의 LSTM,RNN과 비교하긴 힘들다.

그럼에도 불구하고 앞서 3번의 학습들을 정리해보자.

Reuters뉴스 데이터

- GRU : 68.12%

IMDB데이터

- RNN : 82%

- LSTM : 86.28%

정리

- RNN, LSTM, GRU의 배경, 이론을 배움

- 성능 RNN < LSTM, GRU
  - LSTM : 데이터양 많고, 매개변수 많을 때 사용
  - GRU : 상대적으로 데이터양이 더 적을 때, 성능이 좋음