순환신경망

최 석 재

lingua@naver.com

순환신경망

RNN, LSTM, GRU

RNN *Recurrent Neural Network*

by John Hopfield (1982)

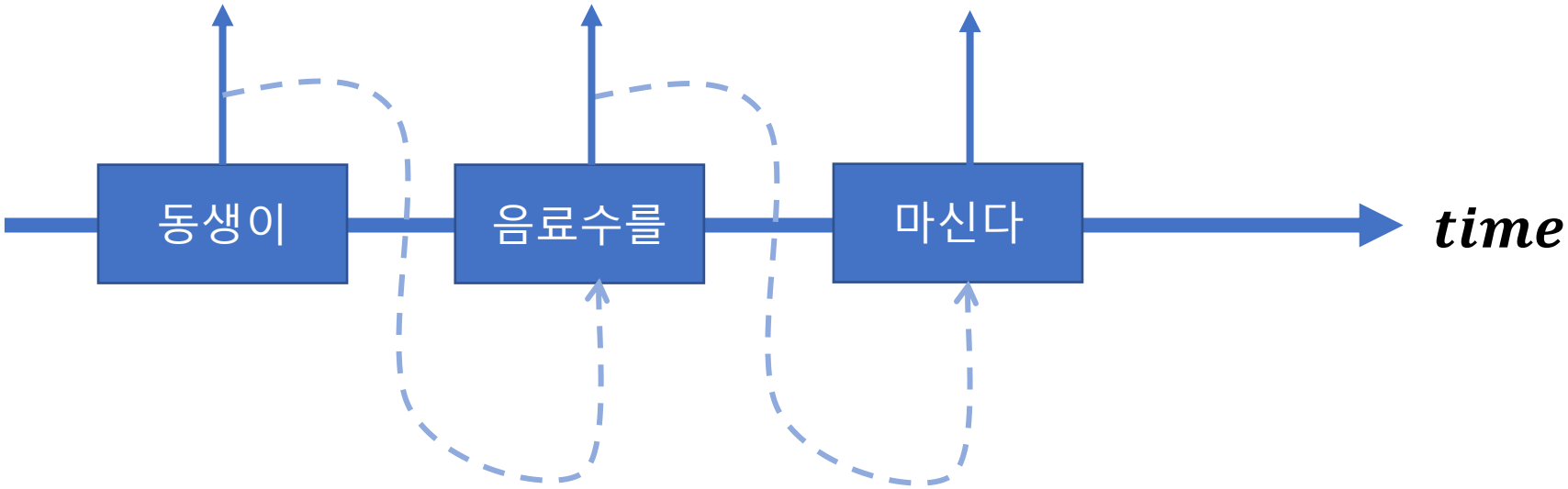
- RNN은 시퀀스 데이터(sequence data)를 다루기에 적절한 알고리즘
- 시퀀스 데이터는 특정 순서가 가정될 수 있는 데이터
- 대표적인 것이 텍스트 데이터로서 각 단어는 예상되는 순서가 있다

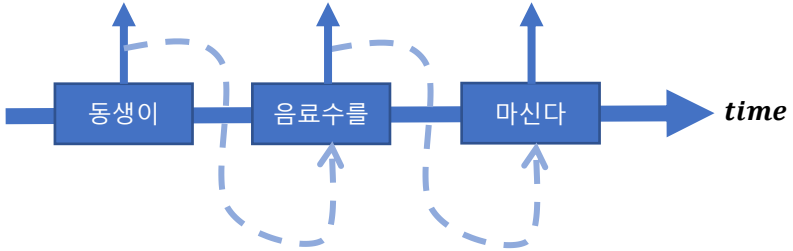
  - 동생이 음료수를 마신다 → 자연스러움
  - 음료수를 동생이 마신다 → 가능하나, 덜 자연스러움
  - 마신다 음료수를 동생이 → 어색함

- '동생이, 음료수를, 마신다' 3개 어절은 확률상 상호 예상 순서가 있다

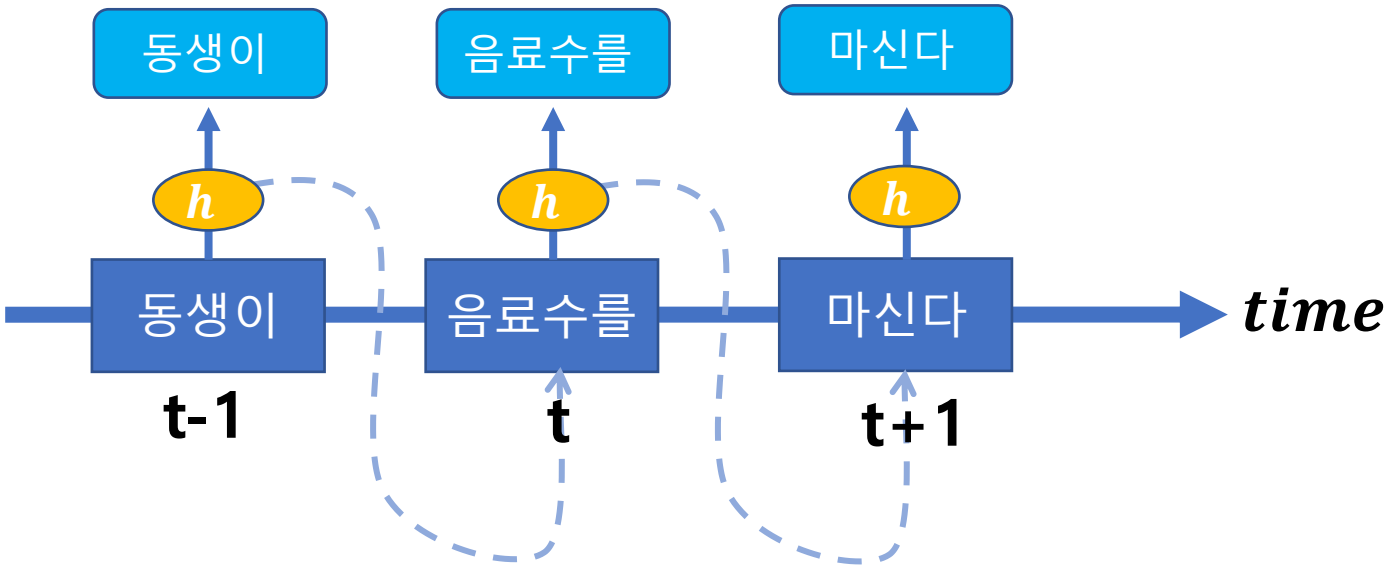
RNN 작동 원리

- 즉, 텍스트 데이터에서는 앞 단어가 뒷 단어의 발생에 영향을 미친다

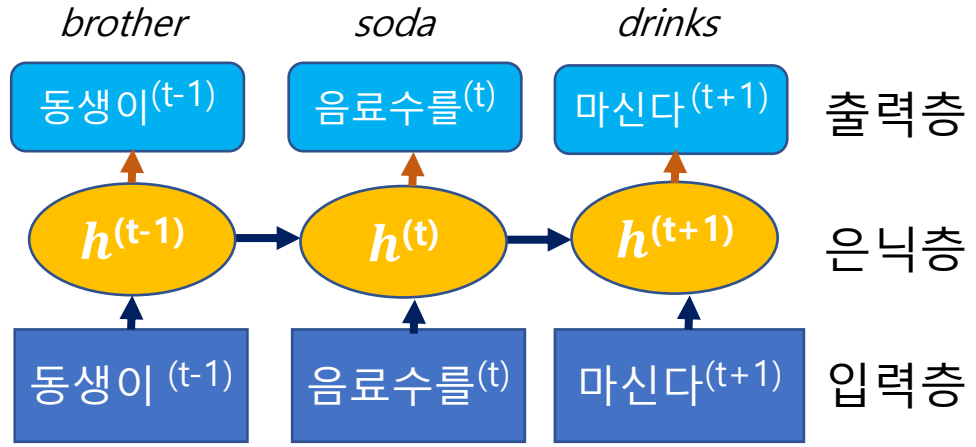




- 이러한 시계열적 특성을 반영한 모델이 순환신경망(RNN)이다



또는

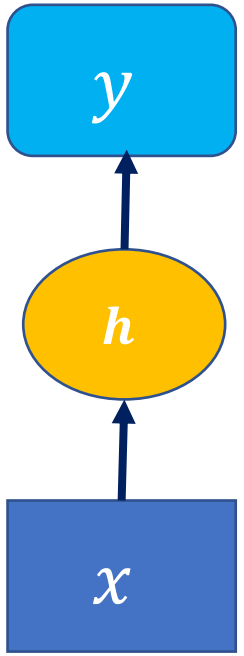


- 은닉유닛 $h^{(t)}$는 두 개의 입력을 받는다

① 입력층의 현재 타임스텝(t)의 입력
② 은닉층의 이전 타임스텝(t-1)의 입력

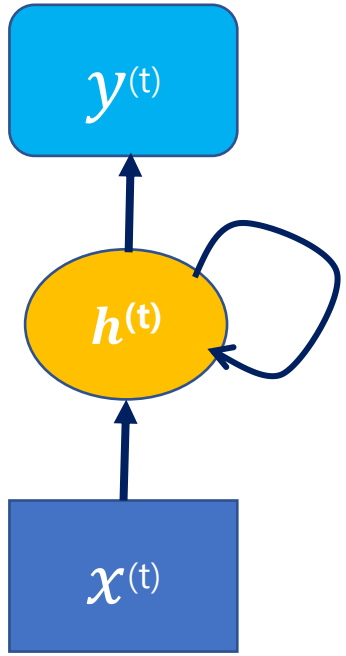
- 이와 같이 순환신경망은 기존 타임스텝에 대한 은닉층을 참고하기 때문에 아래 오른쪽과 같이 표현한다

기본 신경망



동생이 음료수를 마신다

순환 신경망

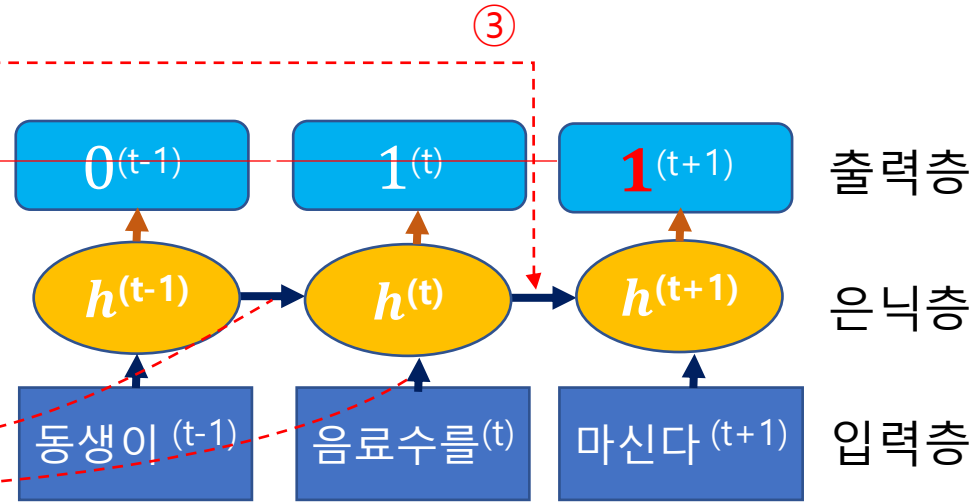


t1. 동생이
t2. 음료수를
t3. 마신다

일반 순환신경망

- 일반 분류 상황에서는 각 타임스텝의 결과는 다음 타임스텝의 상태 업데이트에만 사용
- 마지막 타임스텝에서만 최종 출력

```
state_t = 0
for input_t = input_sequence:
    output_t = fun(input_t, state_t)
    stat_t = output_t
```



일반 순환신경망의 사용

- 순환신경망은 input이 3D(batch_size, time_steps, feature_len)이기 때문에 보통 embedding 층과 연결하여 사용한다
- Embedding 층은 Flatten()하지 않으면 3D로 출력된다

```
model = models.Sequential()

model.add(layers.Embedding(input_dim=10000, output_dim=50, input_length=20))
model.add(layers.SimpleRNN(units=50))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.summary()
```

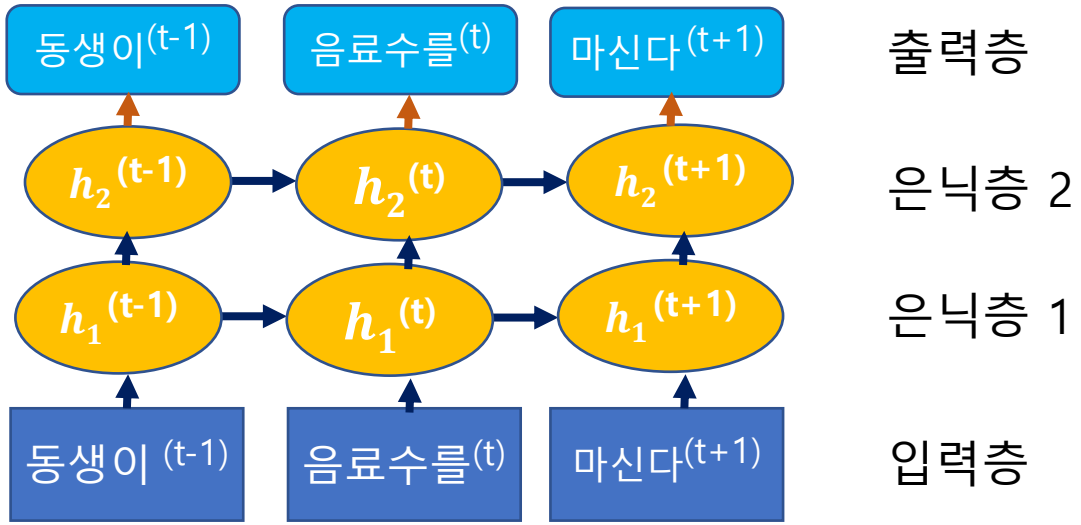
Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 20, 50)	500000
simple_rnn (SimpleRNN)	(None, 50)	5050
dense (Dense)	(None, 32)	1632
dense_1 (Dense)	(None, 1)	33

임베딩의 출력층은 batch_size, input_length(maxlen), output_dim(embedding_dim)로서
RNN의 입력층 3D shape (batch_size=batch size, time_steps=input length, feature_len=output dim)와 같다.
임베딩 결과값의 shape가 RNN이 요구하는 입력문의 shape와 일치한다
① batch_size는 뒷쪽의 model.fit() 단계에서 결정된다
② input_length는 사용하는 단어의 수로 padding 단계에서 결정되었다. 이것이 time_steps로 단어 수만큼 순환된다
③ output_dim은 임베딩 차원의 수로, 결과 노드의 수이다. 이것이 feature length가 된다

가장 중요한 것은 임베딩의 결과로 RNN이 요구하는 3D 포맷이 만들어지게 설계되었다는 점이다.

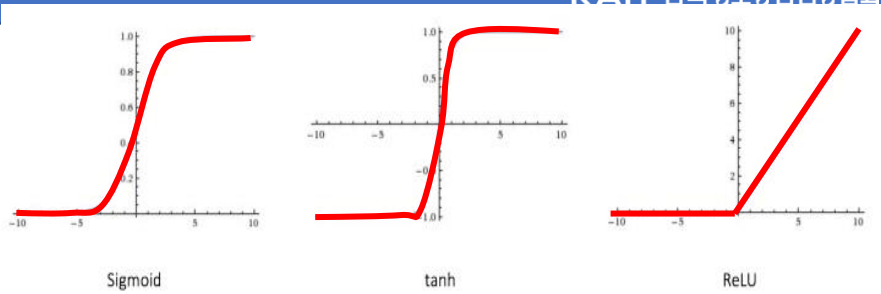
다층 RNN

- RNN 층은 복수로 쌓을 수 있다

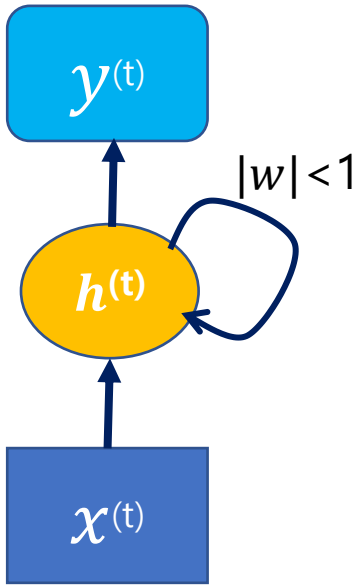


※ LSTM(units=n, return_sequences=True)

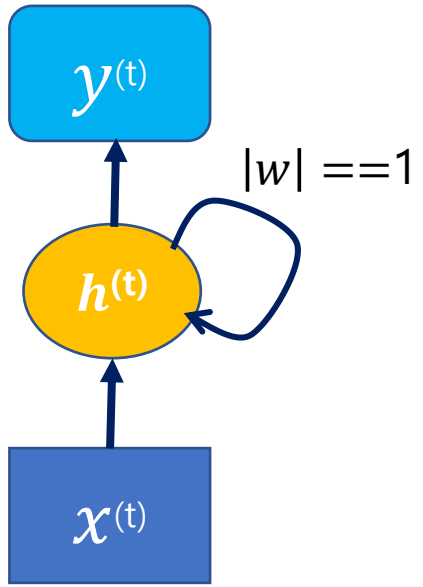
RNN의 문제점



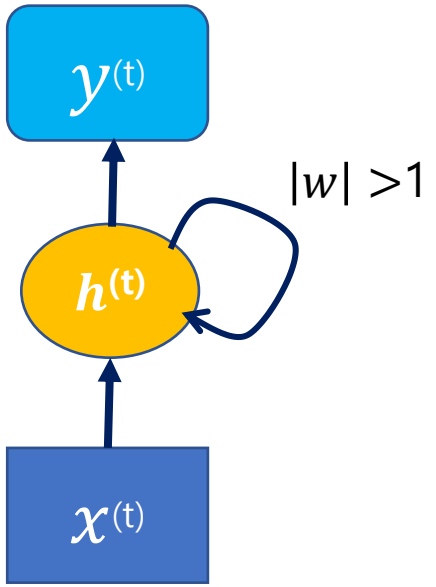
- RNN은 은닉층의 가중치가 지속적으로 곱해지는 것이므로
  가중치 절대값이 1보다 작으면 결국 매우 작은 값이 도출되고 (그래디언트 소실),
  가중치 절대값이 1보다 크면 결국 매우 큰 값이 도출된다 (그래디언트 폭주)
- 특히 가중치는 대부분 1보다 작기 때문에 그래디언트 소실이 일어난다.
  가중치가 활성화 함수 tanh를 지나며 매우 작아져 정보 전달이 안된다.
- 문장이 길어질수록 이 현상이 심해지며, 이를 앞의 정보가 뒤로 잘 전달되지 못한다고 표현한다



그래디언트 소실



그래디언트 유지



그래디언트 폭주

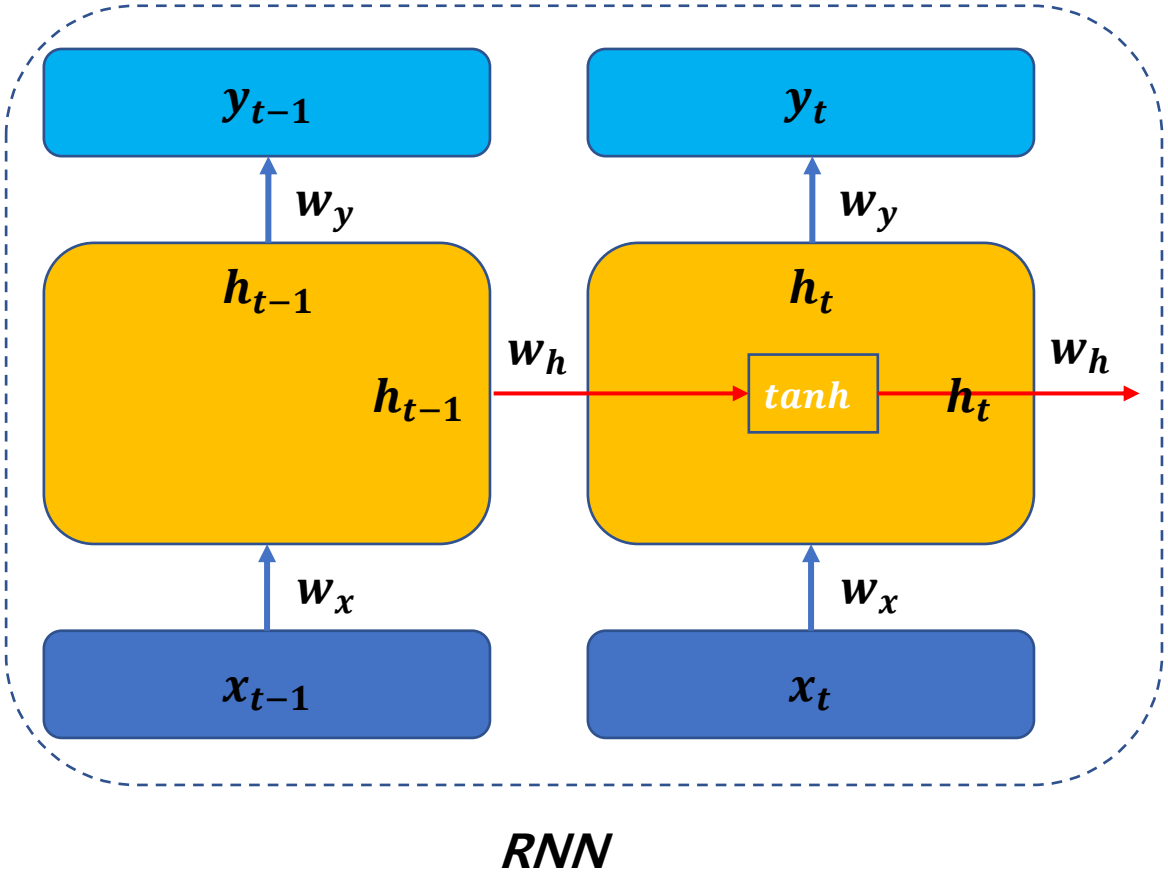
LSTM *Long Short-Term Memory*

by Sepp Hochreiter and Jürgen Schmidhuber (1997)

- LSTM은 과거 정보의 소실이 일어나지 않도록 한다
- 이를 위해 셀 상태(장기 상태), 입력 게이트, 삭제 게이트, 출력 게이트의 네 가지 장치를 만들었다
- 네 가지 장치를 위한 연산이 각각 별도로 존재하기 때문에 RNN에 비하여 계산량이 많아 속도가 느린 점이 단점이다
- 그러나 긴 문장에서도 그래디언트 소실이 잘 일어나지 않는 것이 장점이다. 즉, 정보의 소실이 많이 발생하지 않는다 (오래된 과거의 정보를 더 잘 기억한다)
- 최근 GPU 등의 사용으로 연산속도가 빨라져 연산 시간에 대한 부담이 많이 줄어들었다

기본 구조

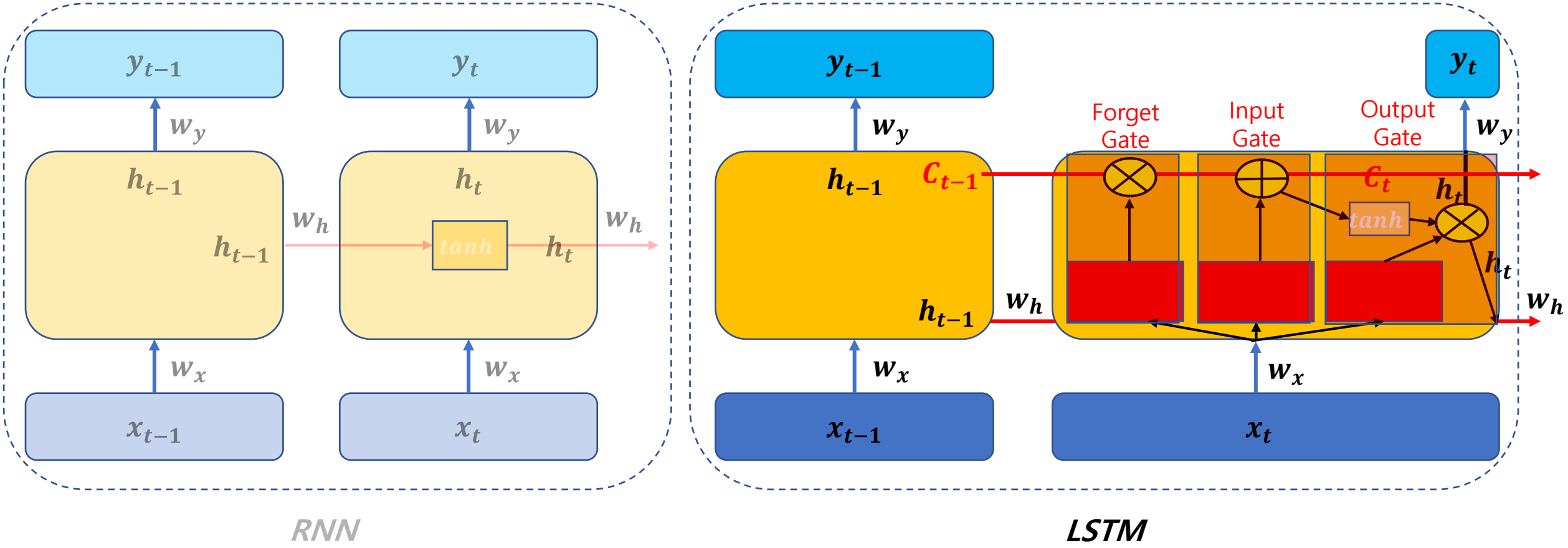
- 먼저 RNN의 구조를 아래와 같이도 표현할 수 있다



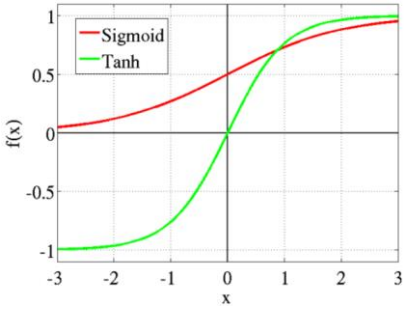
RNN

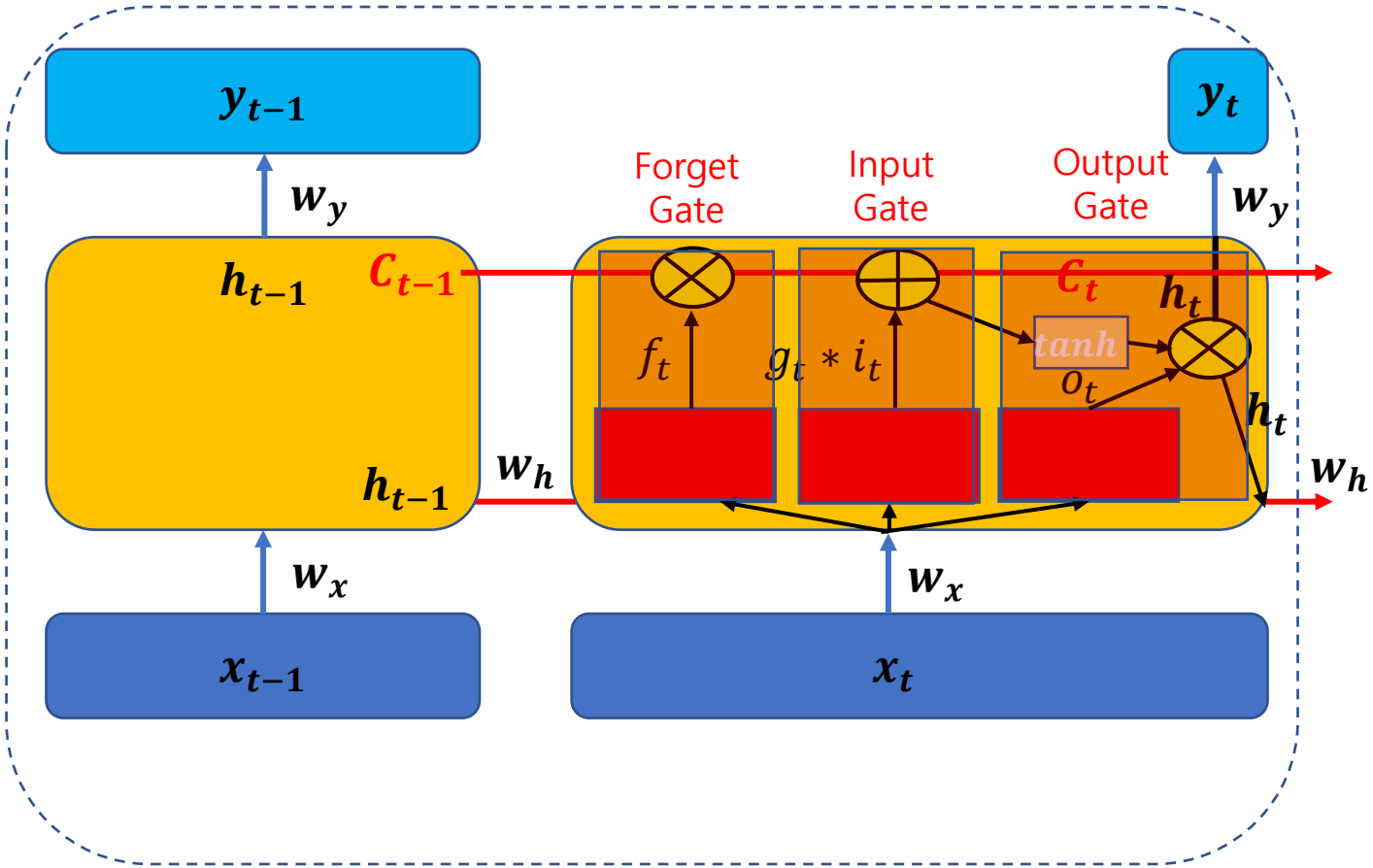
구조 비교

- 이러한 방식으로 RNN과 LSTM의 기본 구조를 비교하면 다음과 같다



구성 요소

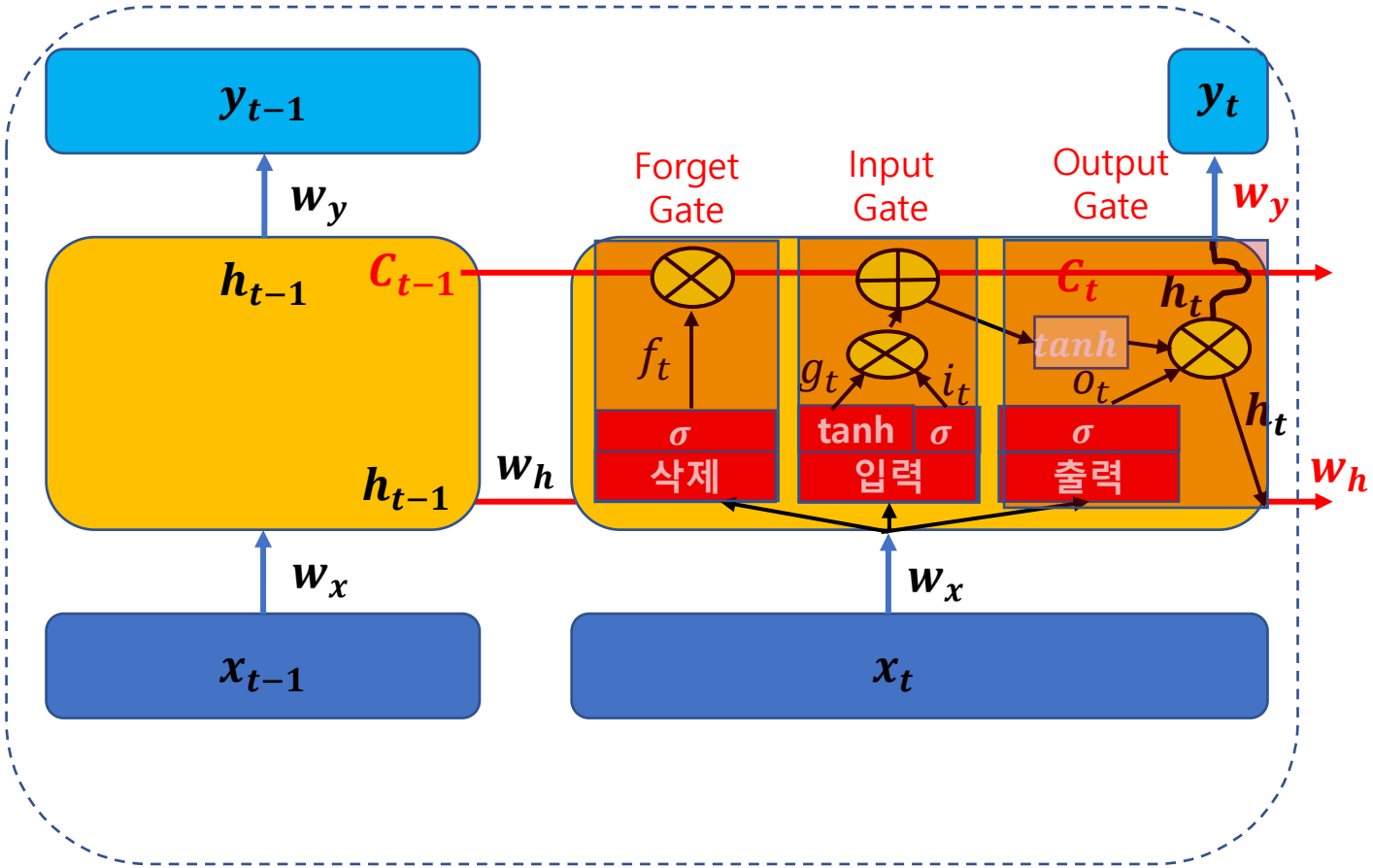




LSTM

- 셀상태(Cell State)는 과거의 정보를 오래 유지하는 것이 목적. 현재 타임스텝의 삭제 게이트의 값을 곱하고, 입력 게이트의 값을 더해가며 계속 다음 타임스텝으로 넘긴다
- 삭제 게이트는 셀상태의 값을 줄이는 것이 목적. f_t 값(Sig 출력)을 도출하여 Cell State와 곱셈 연산을 한다
- 입력 게이트는 현재 타임스텝의 정보를 반영하는 것이 목적. g_t와 i_t 두 값을 도출하여 곱한 뒤, 셀상태에 덧셈 연산을 한다(g_t는 tanh 출력, i_t 는 Sig 출력)
- 출력 게이트는 현재 타임스텝의 출력층과 다음 타임스텝으로 넘길 값을 구하는 것이 목적. 현재 타임스텝의 정보에 Sig 출력한 O_t 를 구한 뒤, 삭제 및 입력 게이트를 거친 Cell State의 tanh 출력한 결과를 곱셈 연산한다
- 셀 상태가 이전 타임스텝의 정보와 현재 타임스텝의 정보를 덧셈 연산하므로 그래디언트 소실이 잘 일어나지 않는다

세부 구조



LSTM

- 앞의 표현을 보다 정확하게 표현하면 다음과 같다
- GRU는 이 과정을 단순화시켜 같은 목적을 이룬다(속도 상승)

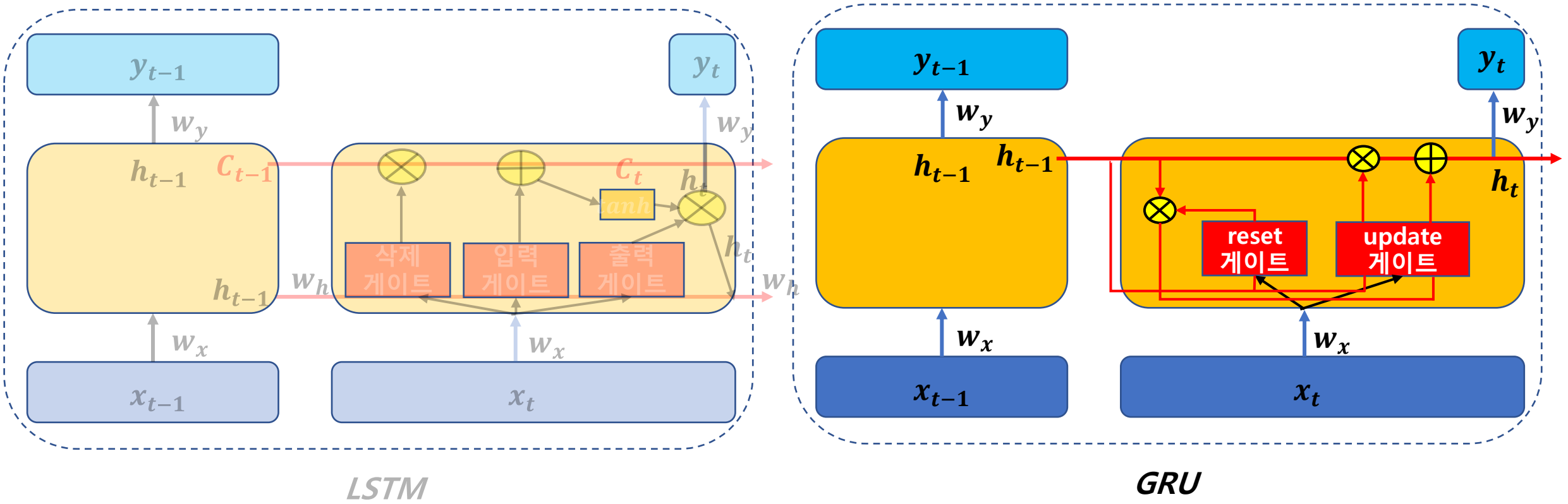
GRU *Gated Recurrent Unit*

by 조경현 (2014)

- LSTM과 같이 다양한 게이트를 두지만 보다 단순화하여 계산량을 줄인다
- 비슷한 성능에 연산시간이 줄어 각광을 받았다
- 최근 GPU의 보급과 성능이 올라가면서 다시 LSTM으로 돌아가는 추세이다

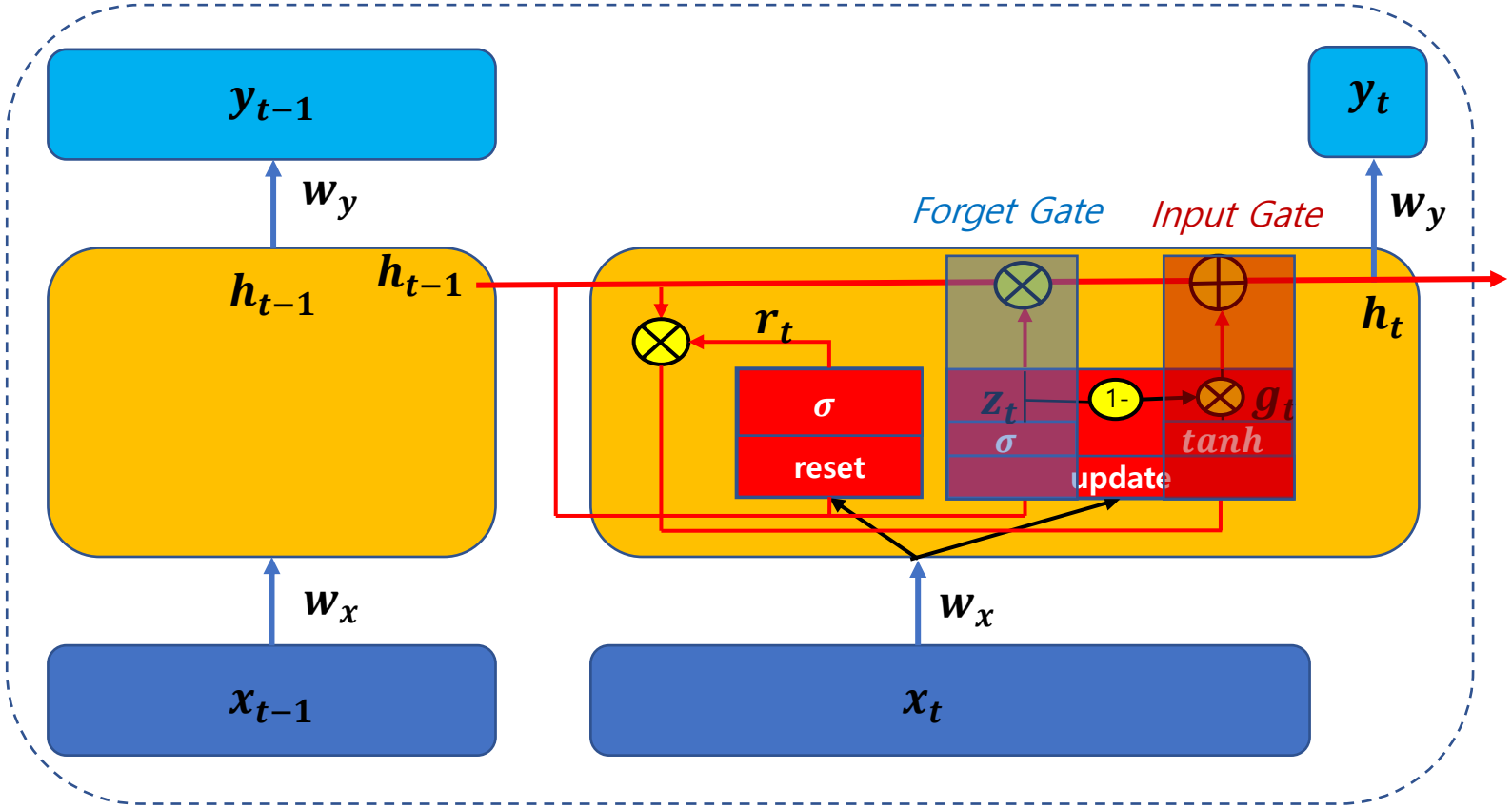
구조 비교

- Cell State가 따로 없이 하나의 상태 벡터를 갖는다
- 게이트의 수가 하나 줄어들었다
- 이렇게 하여도 과거의 정보를 덧셈 연산자로 업데이트하므로 계속 유지할 수 있다



세부 구조

- Sig 결과를 거친 z_t값이 1에 가까운 값이 되면 Input Gate에 해당하는 부분은 0이 곱해져 닫힌다
- z_t값이 0에 가까운 값이 되면 Forget Gate에 해당하는 부분은 0이 곱해져 닫힌다



- Reset Gate는 삭제 게이트에 해당
- Update Gate는 실제로는 입력 게이트와 출력 게이트를 다 가지고 있음
- 다만, 하나의 게이트가 '1-'라는 간단한 연산으로 입력과 출력 게이트의 역할을 다 하는 것임

- 즉, GRU는 상태 게이트의 값이 덧셈 연산으로 소실되지 않게만 해주면 구조를 단순화해도 된다는 것을 보여준다

순환신경망을 이용한 이진분류

RNN, LSTM, GRU

순환신경망 모델

- 순환신경망을 이용한 이진분류 코드는 앞에서 본 워드 임베딩을 이용한 방법과 모델 설계 부분만 제외하고 동일함
- 워드 임베딩을 하게 되면 embedding dimension을 값으로 갖는 층이 하나 더 늘게 되어 출력 데이터의 차원이 2D → 3D로 바뀌게 됨
- RNN 계열 층은 Embedding 이후의 사용을 염두에 두고 있어 입력값이 3D로 되어 있으며, 특별한 조절을 해주지 않아도 잘 맞도록 되어 있음

임베딩의 출력층은 batch_size, input_length(maxlen), output_dim(embedding_dim)로서 RNN의 입력 (batch_size=batch size, time_steps=input length, feature_len=output dim)과 같다

모델 설계

- from keras import models
- from keras import layers

- model = models.Sequential()

RNN 층의 결과는 2D로서, Dense 층이 연결될 수 있다
RNN 층을 복수로 쌓으려면 *model.add(layers.SimpleRNN(units=embedding_dim, return_sequences=True))*

- model.add(layers.Embedding(input_dim=max_words, output_dim=embedding_dim, input_length=maxlen))

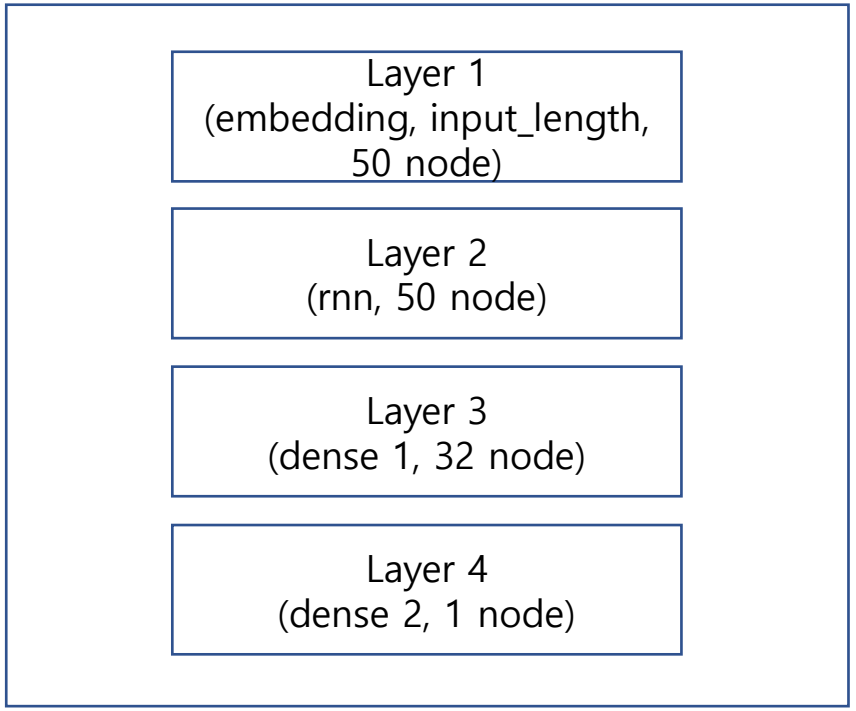
순환신경망을 쌓는 경우에는 return_sequences=True로 한다

- model.add(layers.SimpleRNN(units=embedding_dim, return_sequences=False)) # SimpleRNN, LSTM, GRU
- model.add(layers.Dense(units=32, activation='relu')) # 은닉층
- model.add(layers.Dense(units=class_number, activation='sigmoid')) # 출력층

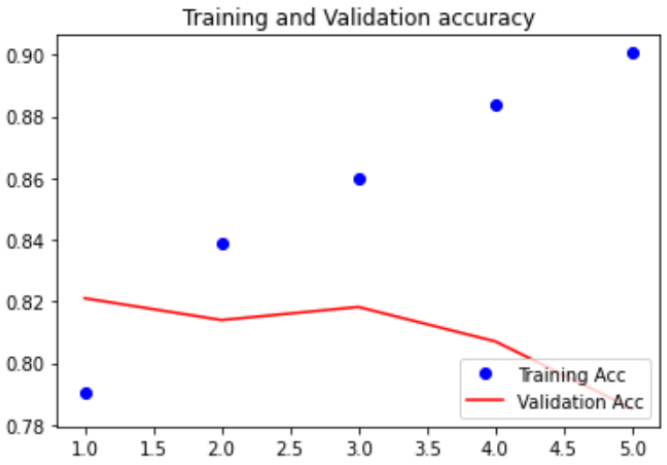
모델 요약 출력

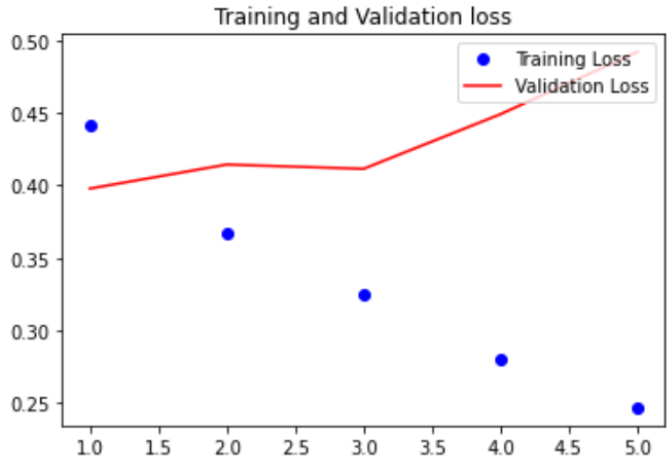
- model.summary()

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 50)          500000

 simple_rnn (SimpleRNN)      (None, 50)                5050

 dense (Dense)               (None, 32)                1632

 dense_1 (Dense)             (None, 1)                 33

=================================================================
Total params: 506,715
Trainable params: 506,715
Non-trainable params: 0
_________________________________________________________________
```



RNN 결과





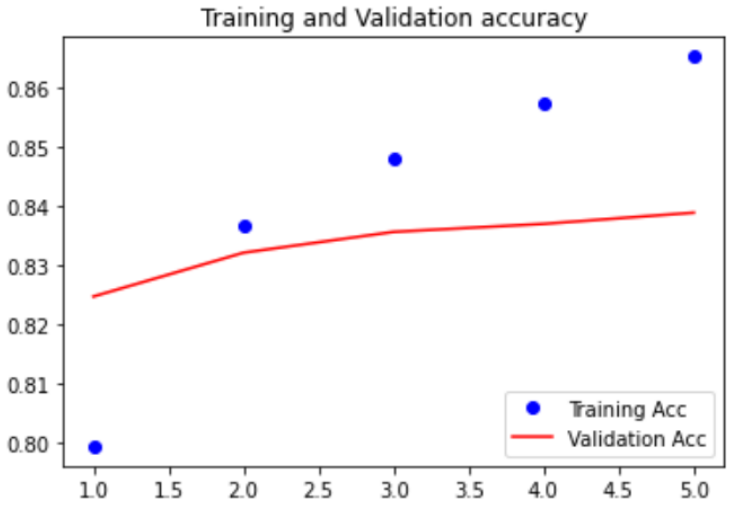
	검증데이터 최고수치
ANN	0.832
Pre-embedding	0.762
Train data Embedding	0.823
SimpleRNN	0.821

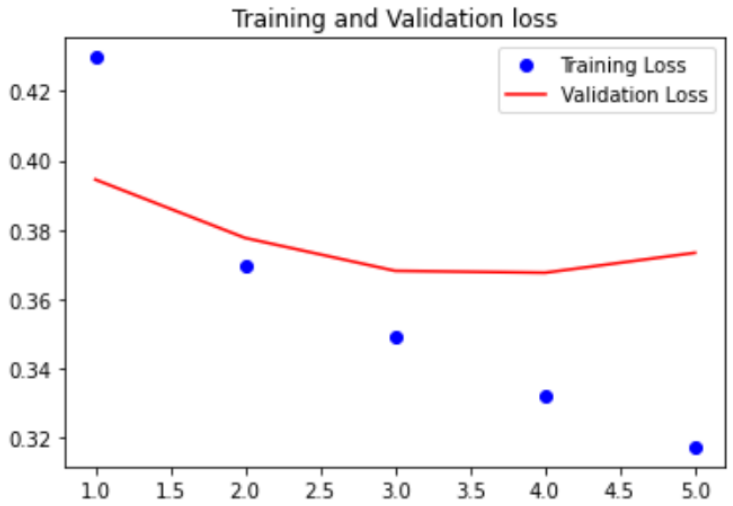
```
Epoch 1/5
1621/1621 [==============================] - 25s 14ms/step - loss: 0.4418 - acc: 0.7903 - val_loss: 0.3978 - val_acc: 0.8210
Epoch 2/5
1621/1621 [==============================] - 18s 11ms/step - loss: 0.3672 - acc: 0.8389 - val_loss: 0.4144 - val_acc: 0.8139
Epoch 3/5
1621/1621 [==============================] - 17s 10ms/step - loss: 0.3253 - acc: 0.8599 - val_loss: 0.4115 - val_acc: 0.8182
Epoch 4/5
1621/1621 [==============================] - 18s 11ms/step - loss: 0.2803 - acc: 0.8837 - val_loss: 0.4492 - val_acc: 0.8070
Epoch 5/5
1621/1621 [==============================] - 17s 10ms/step - loss: 0.2465 - acc: 0.9006 - val_loss: 0.4923 - val_acc: 0.7852
```

```
Train accuracy of each epoch: [0.79  0.839 0.86  0.884 0.901]
Validation accuracy of each epoch: [0.821 0.814 0.818 0.807 0.785]
```

```
772/772 [==============================] - 3s 3ms/step - loss: 0.4878 - acc: 0.7877
prediction model loss & acc: [0.4877738058567047, 0.7876898050308228]
```

LSTM 결과





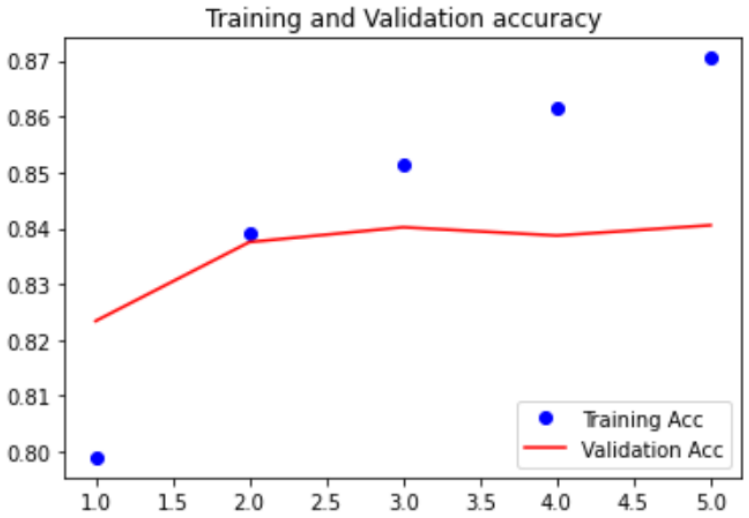
	검증데이터 최고수치
ANN	0.832
Pre-embedding	0.762
Train data Embedding	0.823
SimpleRNN	0.821
LSTM	0.839

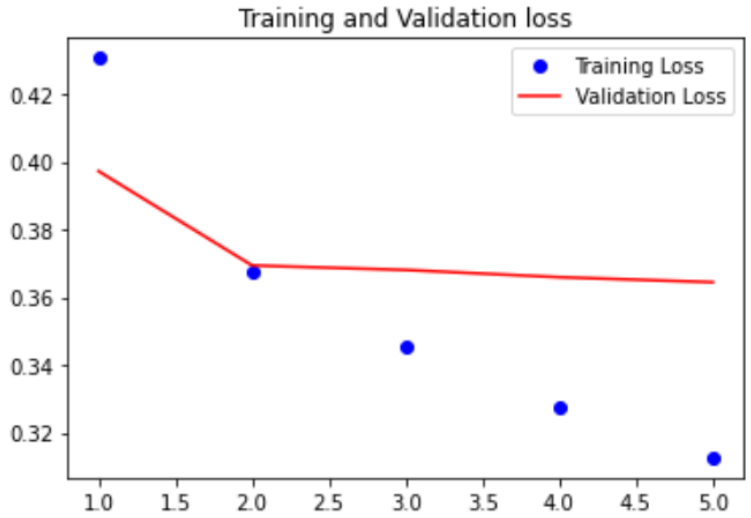
```
Epoch 1/5
1621/1621 [==============================] - 35s 20ms/step - loss: 0.4298 - acc: 0.7995 - val_loss: 0.3945 - val_acc: 0.8247
Epoch 2/5
1621/1621 [==============================] - 35s 21ms/step - loss: 0.3697 - acc: 0.8368 - val_loss: 0.3776 - val_acc: 0.8321
Epoch 3/5
1621/1621 [==============================] - 34s 21ms/step - loss: 0.3494 - acc: 0.8480 - val_loss: 0.3682 - val_acc: 0.8356
Epoch 4/5
1621/1621 [==============================] - 31s 19ms/step - loss: 0.3319 - acc: 0.8572 - val_loss: 0.3676 - val_acc: 0.8370
Epoch 5/5
1621/1621 [==============================] - 31s 19ms/step - loss: 0.3173 - acc: 0.8652 - val_loss: 0.3734 - val_acc: 0.8389
Train accuracy of each epoch: [0.8   0.837 0.848 0.857 0.865]
Validation accuracy of each epoch: [0.825 0.832 0.836 0.837 0.839]
```

※ 더 높아질 수도 있었다

```
772/772 [==============================] - 5s 5ms/step - loss: 0.3823 - acc: 0.8352
prediction model loss & acc: [0.38228243589401245, 0.8351892828941345]
```

GRU 결과



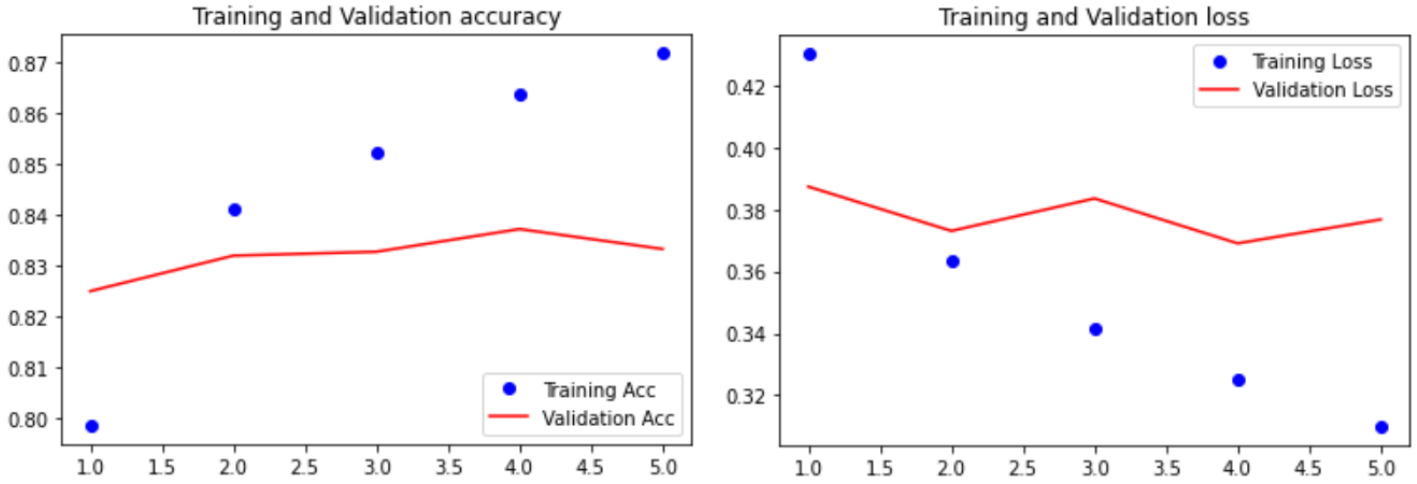


	검증데이터 최고수치
ANN	0.832
Pre-embedding	0.762
Train data Embedding	0.823
SimpleRNN	0.821
LSTM	0.839
GRU	0.841

```
Epoch 1/5
1621/1621 [==============================] - 38s 21ms/step - loss: 0.4307 - acc: 0.7990 - val_loss: 0.3972 - val_acc: 0.8234
Epoch 2/5
1621/1621 [==============================] - 31s 19ms/step - loss: 0.3673 - acc: 0.8392 - val_loss: 0.3694 - val_acc: 0.8375
Epoch 3/5
1621/1621 [==============================] - 31s 19ms/step - loss: 0.3452 - acc: 0.8514 - val_loss: 0.3681 - val_acc: 0.8402
Epoch 4/5
1621/1621 [==============================] - 31s 19ms/step - loss: 0.3276 - acc: 0.8614 - val_loss: 0.3659 - val_acc: 0.8387
Epoch 5/5
1621/1621 [==============================] - 31s 19ms/step - loss: 0.3126 - acc: 0.8705 - val_loss: 0.3644 - val_acc: 0.8405

Train accuracy of each epoch: [0.799 0.839 0.851 0.861 0.87 ]
Validation accuracy of each epoch: [0.823 0.838 0.84  0.839 0.841]

772/772 [==============================] - 4s 5ms/step - loss: 0.3740 - acc: 0.8359
prediction model loss & acc: [0.3739853501319885, 0.8358777165412903]
```

※ 더 높아질 수도 있었다

GRU 2층 결과



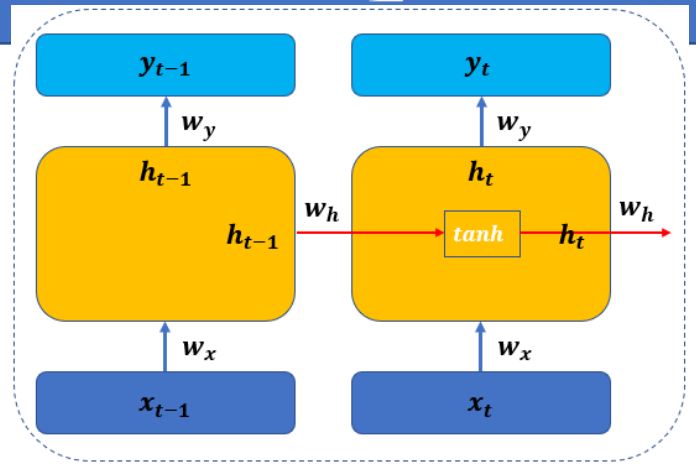
	검증데이터 최고수치
ANN	0.832
Pre-embedding	0.762
Train data Embedding	0.823
SimpleRNN	0.821
LSTM	0.839
GRU	0.841
GRU 2층	0.837

```
Epoch 1/5
1621/1621 [==============================] - 63s 34ms/step - loss: 0.4302 - acc: 0.7986 - val_loss: 0.3873 - val_acc: 0.8250
Epoch 2/5
1621/1621 [==============================] - 53s 33ms/step - loss: 0.3632 - acc: 0.8411 - val_loss: 0.3731 - val_acc: 0.8319
Epoch 3/5
1621/1621 [==============================] - 52s 32ms/step - loss: 0.3414 - acc: 0.8523 - val_loss: 0.3835 - val_acc: 0.8327
Epoch 4/5
1621/1621 [==============================] - 52s 32ms/step - loss: 0.3249 - acc: 0.8636 - val_loss: 0.3690 - val_acc: 0.8372
Epoch 5/5
1621/1621 [==============================] - 50s 31ms/step - loss: 0.3100 - acc: 0.8717 - val_loss: 0.3767 - val_acc: 0.8333

Train accuracy of each epoch: [0.799 0.841 0.852 0.864 0.872]
Validation accuracy of each epoch: [0.825 0.832 0.833 0.837 0.833]

 772/772 [==============================] - 7s 8ms/step - loss: 0.3762 - acc: 0.8314
 prediction model loss & acc: [0.37621983885765076, 0.8313828706741333]
```

연습문제

1. 임베딩 차원을 다르게 하여 성능을 평가해 보세요 (easy)
2. RNN, LSTM, GRU를 적용해 보세요 (easy)
3. 챗봇 데이터를 이용하여 다중분류에 적용해 보세요

참고: Parameter

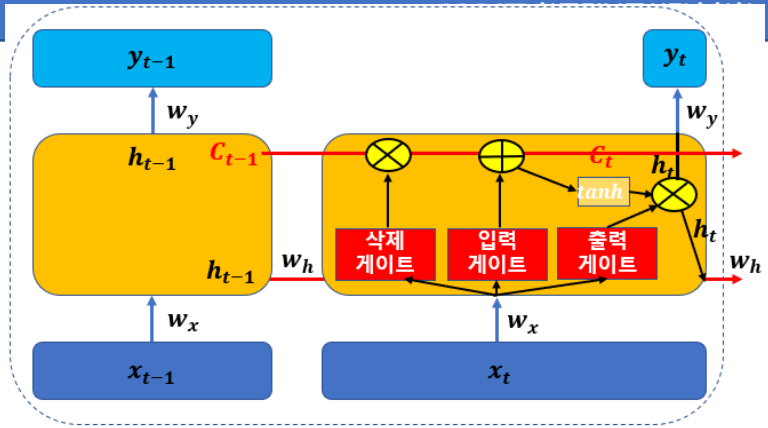


- Parameter 수는 자동 계산되므로 확인할 필요는 없다 (계산시간추정)
- 다만 RNN과 LSTM를 Parameter 수로 비교하여 본다
- 순환층은 각 타임스텝과 곱하는 가중치가 있다는 점이 Dense층과 다르다
- 먼저 SimpleRNN의 경우,

```
model = models.Sequential()

model.add(layers.Embedding(input_dim=10000, output_dim=50))
model.add(layers.SimpleRNN(50))
model.add(layers.Dense(32))
model.add(layers.Dense(1, activation='sigmoid'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 20, 50)            500000

simple_rnn (SimpleRNN)       (None, 50)                5050

dense (Dense)                (None, 32)                1632

dense_1 (Dense)              (None, 1)                 33

=================================================================
Total params: 506,715
Trainable params: 506,715
Non-trainable params: 0
```

- Embedding 층은 input이 10,000 x output 50 이므로 모든 Parameter의 종류는 500,000
- RNN 층은 input이 50이고 output도 50 이므로 우선 (50 x 50)
  + 이전 타임스텝의 셀(미니배치) 50개 각각에 대한 가중치 x 현재 타임스텝의 50 셀에 대한 가중치
  + 50개 셀에 대한 편향(y절편) = 5,050
- Dense 층은 input 50 x output 32 + 편향 32 = 1,632
- 출력층은 input 32 x output 1 + 편향 1 = 33



- 다음은 LSTM의 경우이다
- LSTM은 RNN과는 다르게 게이트 3개가 추가로 있다
- 이 3개의 게이트 각각에도 가중치가 각각 배당된다는 점이 RNN과 다르다

```
model = models.Sequential()

model.add(layers.Embedding(input_dim=1000, output_dim=50))
model.add(layers.LSTM(50))
model.add(layers.Dense(32))
model.add(layers.Dense(1, activation='sigmoid'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 50)          500000

lstm (LSTM)                  (None, 50)                20200

dense (Dense)                (None, 32)                1632

dense_1 (Dense)              (None, 1)                 33

=================================================================
Total params: 521,865
Trainable params: 521,865
Non-trainable params: 0
_________________________________________________________________
```

- LSTM 층은 RNN과 같이 다음을 기본으로 한다
  (input 50 x output 50) + (previous 50 x current 50) + 50
- 이것이 3개의 게이트와 Cell State 총 4개에 모두 적용된다
- 따라서 4 x ((input 50 x output 50) + (previous 50 x previous 50) + 50) = 20,200

※ RNN의 5,050 개 파라미터에 비교하여 4배가 많다