8. 순환신경망

8.1. 순차적 데이터

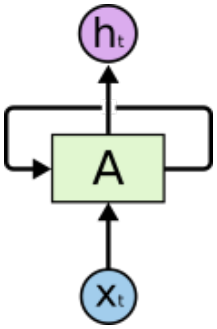
주가나 날씨처럼 시간에 따라 변화하거나, 텍스트나 음악처럼 글자나 음이 순서대로 나타난 정보들이 있다. 이러한 데이터를 **순차적 데이터**(sequential data)라고 한다.

순차적 데이터에서는 대개 순서상 앞이나 뒤에 있는 정보가 서로 영향을 주기도 하고, 주기성이나 경향성을 띄기도 한다. 날씨의 경우 1년을 주기로 추웠다 더워지기를 반복하고, 텍스트의 경우 앞에 나온 말을 보면 뒤에 나올 말을 짐작할 수 있다.

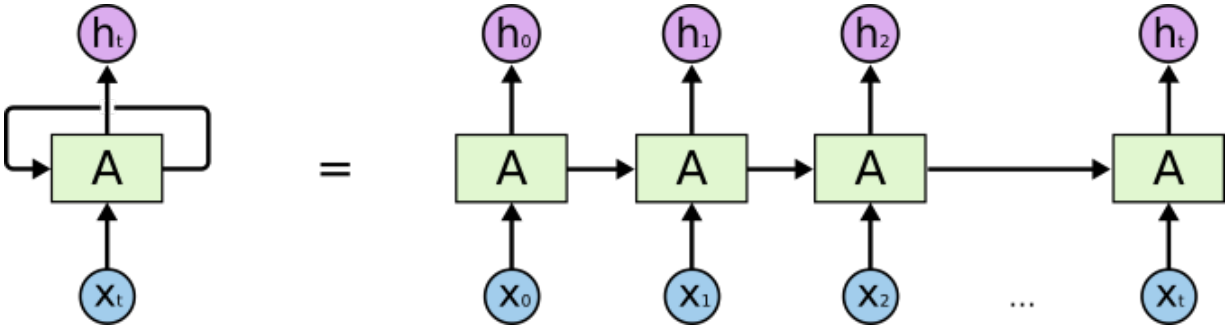
8.2. 순환신경망

순차적 데이터를 다루는 전통적인 방법으로는 자기회귀(autoregression)나 마코브 연쇄(Markov chain)와 같은 방법들이 있다. 딥러닝에서 이러한 특성을 분석할 수 있는 방법으로는 순환신경망(Recurrent Neural Network, 이하 RNN)가 있다.

RNN에는 여러가지 형태가 있으나 가장 대표적인 것은 아래 그림과 같은 형태이다.



입력층 X에서 은닉층 A를 거쳐 출력층 h로 신호가 전달된다는 점에서 RNN은 앞먹임 신경망과 동일한 구조를 가진다. 한 가지 차이는 은닉층에서 자기 자신으로 돌아오는 고리가 있다는 것이다. 즉 첫번째 입력 X_1이 한 번 신경망을 거쳐 나가고 나면, 두번째 입력 X_2가 처리될 때는 X_1의 은닉층 상태가 X_2의 입력과 함께 은닉층으로 들어오게 된다. 이러한 과정을 통해서 앞의 입력이 뒤의 입력에 미치는 영향을 파악할 수 있다. 따라서 RNN을 펼치면 아래와 같은 형태의 네트워크가 된다.



8.3. 장기 의존성의 문제

RNN에서 은닉층에서 은닉층으로 가는 연결은 같은 신호를 반복해서 전달한다. 즉, 다음과 같은 형태의 식으로 표현된다.

$$A_t = f(A_{t-1}, X_t) = \sigma(wA_{t-1} + vX_t + b)$$

논의를 간단히 하기 위해 입력층과 절편은 제외하고 생각해보자.

$$A_t = \sigma(wA_{t-1})$$

그러면 은닉층의 이전 상태에 대한 다음 상태의 미분은 아래와 같다.

$$\frac{\partial A_t}{\partial A_{t-1}} = w\sigma'(wA_{t-1})$$

만약 은닉층을 여러 층 거칠 경우에 그 미분은 아래와 같은 식이 된다.

$$\begin{aligned}\frac{\partial A_{t+n}}{\partial A_t} &= \frac{\partial A_{t+n}}{\partial A_{t+n-1}} \cdots \frac{A_{t+1}}{A_t} \\ &= \prod_{k=0}^{n-1} \frac{A_{t+k+1}}{A_{t+k}} \\ &= \prod_{k=0}^{n-1} w\sigma'(wA_{t+k}) \\ &= w^n \prod_{k=0}^{n-1} \sigma'(wA_{t+k})\end{aligned}$$

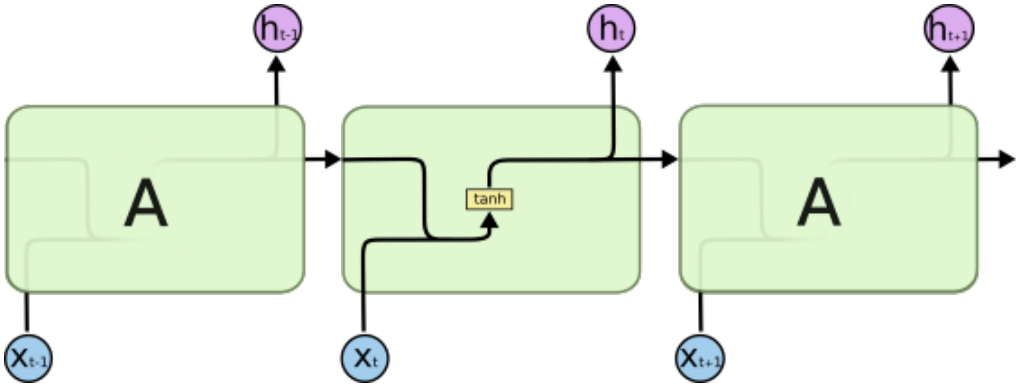
여기서 보면 w^n 때문에 n이 커지면 $w < 1$인 경우 미분이 0으로 수렴하고, $w > 1$인 경우 발산하게 된다. 전자는 사라지는 경사, 후자는 폭발하는 경사(exploding gradient)라고 한다.

사라지는/폭발하는 경사 자체는 앞먹임 신경망에서도 동일한 문제이나 RNN에서는 더 심각한 문제가 된다. 앞먹임 신경망에서는 모형의 깊이가 깊어지면서 생기는 문제지만, RNN에서는 데이터가 길어지기만해도 생기는 문제이기 때문이다.

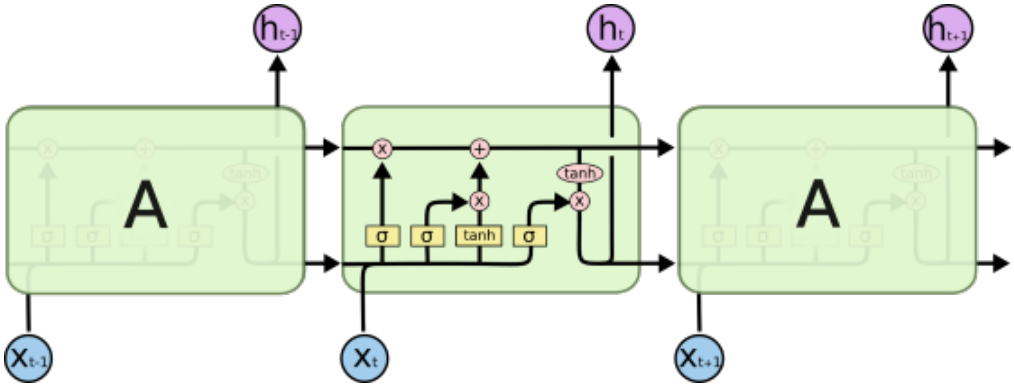
두 입력 사이에 거리가 멀리 떨어진 경우($n \gg 0$)에 존재하는 영향을 장기 의존성(long-term dependency)이라고 한다. 한국어의 텍스트의 경우 주어는 문장의 맨 처음에, 동사는 문장의 맨 끝에 나오므로 문장이 길어지면 주어와 동사 사이에 장기 의존성이 생기게 된다. 그런데 RNN은 두 입력 사이의 거리가 멀면 경사하강법이 잘 작동하지 않아, 장기의존성을 학습하기가 어렵다.

8.4. LSTM

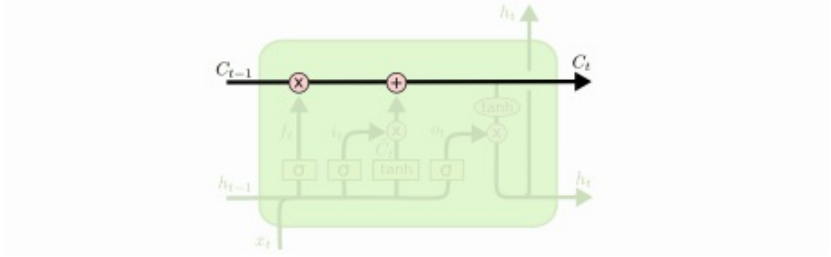
단순한 RNN을 좀 더 자세히 보면 아래와 같은 구조를 가지고 있다. 입력 X_t와 이전 은닉층의 상태 A_{t-1}이 합쳐져 활성화 함수로 들어가고 그 출력 h_t가 출력층과 다음 은닉층으로 넘어가는 것이다.



RNN이 장기 의존성을 학습하지 못하는 이유는 사라지는/폭발하는 경사 때문이고, 사라지는/폭발하는 경사는 활성화 함수를 여러 번 반복해서 거치기 때문이다. 그렇다면 은닉층에서 은닉층으로 바로 신호가 전달할 수 있게 하면 어떨까? 이 아이디어에 바탕을 둔 것이 LSTM(Long Short-Term Memory)이다.



매우 복잡하게 보이지만 위의 그림에서 핵심 아이디어는 아래 부분이다.



일단 LSTM에서는 은닉층에서 은닉층으로 전달되는 신호 C와 은닉층에서 출력층으로 전달되는 신호 h가 분리되었다. 그리고 C는 별다른 활성화 함수를 거치지 않고 바로 다음 은닉층으로 전달된다. 따라서 사라지는/폭발하는 경사 문제에서 자유롭게 된다. 대신 망각 게이트(forgetting gate, 위의 그림에서 분홍색 원 안의 X)를 두어 신호를 차단하거나, 입력 게이트(input gate, 분홍색 원 안의 +)를 통해 새로운 신호를 추가한다.

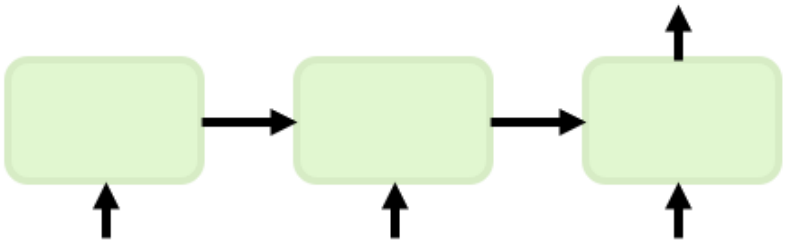
LSTM은 최근에 특히 텍스트, 음성 분석 등에서 각광을 받고 있다. (LSTM 자체는 1997년에 발표된 모형이다.) LSTM은 다음과 같은 문제들에 탁월한 성과를 보여주고 있다.

- 손글씨 인식
- 음성 인식
- 기계 번역
- 이미지 설명 생성
- 문법 분석

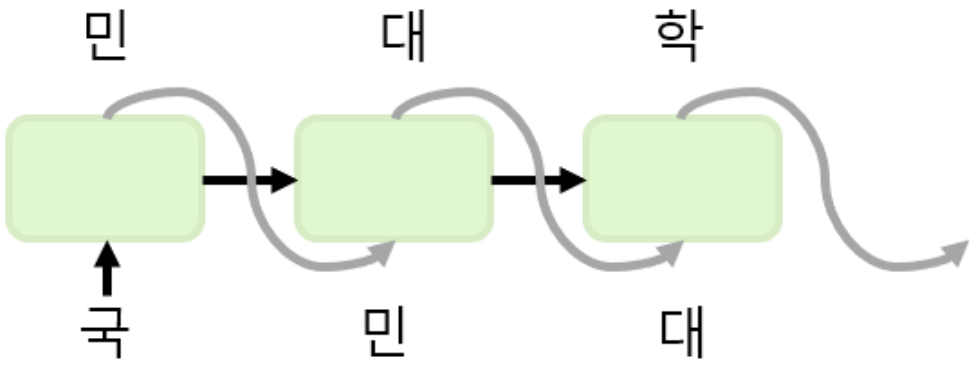
8.5. 순환신경망의 여러 구조

순환신경망의 세부적인 구조는 문제에 따라 달라진다. 우선은 입력과 출력이 다른 경우를 보자. 예를 들어 입력에는 거시경제 변수를, 출력에는 코스피 지수를 넣을 수 있다. 이럴 때는 입력과 출력의 갯수가 똑같아야 한다.

그런데 입력은 여러 개지만 출력은 1개만 있을 때도 있다. 예를 들면 텍스트를 분석해서 하나의 점수를 주는 경우가 있다. 이때는 아래 그림과 같이 마지막 노드만 출력을 하도록 한다.



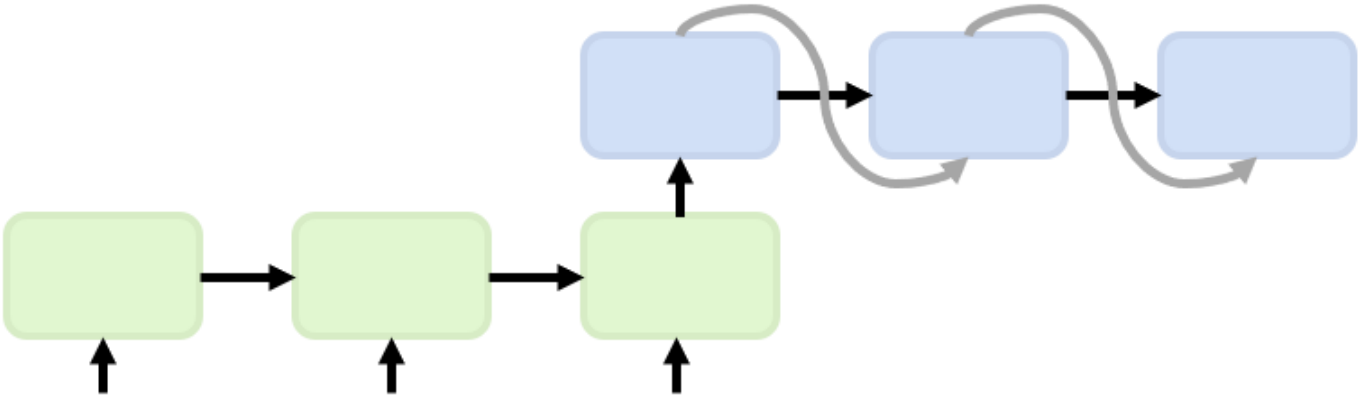
입력과 출력이 같은 경우도 있다. 스마트폰의 키보드에는 다음에 입력할 글자나 단어를 추천해주는 경우가 있다. 이럴 경우 입력을 앞 글자, 출력을 뒷 글자로 하고 앞의 출력을 다시 뒤의 입력으로 넣어주는 구조로 만든다. 아래 그림은 "국민대학"이라는 4글자를 이런 구조의 순환신경망으로 학습/예측시키는 예이다.



이렇게 출력을 다시 입력으로 넣는 구조에서는 학습시킬 때와 예측할 때가 달라진다. 예측을 할 때는 위의 그림처럼 바로 출력을 입력으로 넣으면 되지만, 학습을 시킬 때는 뒤로 갈 수록 실제 데이터와 입력이 달라져버리는 경우가 생긴다. 예를 들어 2번째 노드가 "민" 다음에 "의"를 예측하면 3번째 노드는 입력이 "의"가 되고 출력이 "학"이 되어 버린다. 이런 문제를 막기 위해 학습 시킬 때는 실제 데이터를 입력에 넣어주는 방법을 쓴다. 이런 기법을 teacher forcing이라고 한다.

8.5.1. Seq2Seq

번역의 경우에는 입력된 단어의 갯수와 출력된 단어의 갯수가 서로 다르다. 이럴 때는 앞에서 배운 구조들을 합쳐서 Seq2Seq라는 구조를 만든다.



seq2seq는 2개의 네트워크로 이뤄지는데 번역의 경우 원래 언어를 처리하는 인코더(encoder)와 번역될 언어를 생성하는 디코더(decoder)로 나뉘어진다. 인코더는 원래의 언어의 의미를 하나의 벡터로 디코더에 전달하고, 디코더는 이를 다시 번역될 언어의 순서로 생성하는 것이다.

seq2seq는 번역만이 아니라 문서 요약 등에도 사용할 수 있다. 다만 seq2seq는 긴 문장의 의미를 하나의 벡터로 압축해야한다는 어려움이 있는데, 최근에는 원 문장의 특정 부분에만 주의(attention)를 주는 방법이 해결책으로 제안되고 있다.

8.6. 실습

`keras`를 불러들인다.

```
library(keras)
```

8.6.1. imdb 데이터

`imdb` 영화평 데이터를 불러온다. 해당 데이터는 케라스에 내장되어 있다.

```
NUM.WORDS = 10000
```

```
imdb <- dataset_imdb(num_words = NUM.WORDS)
word_index <- dataset_imdb_word_index()
```

훈련용 데이터와 테스트용 데이터를 불러온다. x는 영화평, y는 영화평의 긍/부정 여부다.

```
x_train <- imdb$train$x
y_train <- imdb$train$y
x_test <- imdb$test$x
y_test <- imdb$test$y
```

y는 0과 1로 이뤄져 있다.

```
y_train[1:10]
```

```
[1] 1 0 0 1 0 0 1 0 1 0
```

x는 단어 번호로 이뤄져 있다. 가장 많이 사용된 단어 50종은 제외되어 있으며, 그 외에는 빈도 순으로 NUM.WORDS에 지정된 갯수의 단어만을 포함한다.

```
x_train[1]
```

```
[[1]]
  [1]    1   14   22   16   43  530  973 1622 1385   65  458 4468   66 3941    4
 [16]  173   36  256    5   25  100   43  838  112   50  670    2    9   35  480
 [31]  284    5  150    4  172  112  167    2  336  385   39    4  172 4536 1111
 [46]   17  546   38   13  447    4  192   50   16    6  147 2025   19   14   22
 [61]    4 1920 4613  469    4   22   71   87   12   16   43  530   38   76   15
 [76]   13 1247    4   22   17  515   17   12   16  626   18    2    5   62  386
 [91]   12    8  316    8  106    5    4 2223 5244   16  480   66 3785   33    4
[106]  130   12   16   38  619    5   25  124   51   36  135   48   25 1415   33
[121]    6   22   12  215   28   77   52    5   14  407   16   82    2    8    4
[136]  107  117 5952   15  256    4    2    7 3766    5  723   36   71   43  530
[151]  476   26  400  317   46    7    4    2 1029   13  104   88    4  381   15
[166]  297   98   32 2071   56   26  141    6  194 7486   18    4  226   22   21
[181]  134  476   26  480    5  144   30 5535   18   51   36   28  224   92   25
[196]  104    4  226   65   16   38 1334   88   12   16  283    5   16 4472  113
[211]  103   32   15   16 5345   19  178   32
```

keras의 RNN은 항상 펼쳐진 상태로만 사용가능하다. 따라서 최대 길이를 미리 지정해줘야 한다.

```
MAXLEN = 20
```

pad_sequence는 최대 길이보다 짧으면 0으로 채우고, 길면 잘라낸다.

```
x_train = pad_sequences(x_train, MAXLEN)
```

```
x_test = pad_sequences(x_test, MAXLEN)
```

8.6.2. 앞먹임 신경망

```
ff = keras_model_sequential()
```

layer_embedding은 각 단어를 8차원 공간상의 한 점으로 투사한다. 이후 데이터를 평평하게 만들어 앞먹임 신경망에 넣는다.

```
ff %>%
    layer_embedding(input_dim = NUM.WORDS, output_dim = 8, input_length = MAXLEN) %>%
    layer_flatten() %>%
    layer_dense(1, activation='sigmoid')
```

요약

```
summary(ff)
```

```
_____________________________________________________________________________
Layer (type)                     Output Shape                  Param #
=============================================================================
embedding_1 (Embedding)          (None, 20, 8)                 80000
_____________________________________________________________________________
flatten_1 (Flatten)              (None, 160)                   0
_____________________________________________________________________________
dense_1 (Dense)                  (None, 1)                     161
=============================================================================
Total params: 80,161
Trainable params: 80,161
Non-trainable params: 0
_____________________________________________________________________________
```

모형 설정은 크게 다르지 않다.

```
ff %>% compile(optimizer = optimizer_rmsprop(), loss = 'binary_crossentropy', metrics = 'accuracy')
```

학습을 시킨다.

```
history = ff %>%  fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2,
  callbacks = c(callback_early_stopping(monitor = "val_loss"))
)
```

학습 기록을 확인한다.

```
history
```

```
Trained on 20,000 samples, validated on 5,000 samples (batch_size=128, epochs=10)
Final epoch (plot to see history):
val_loss: 0.496
 val_acc: 0.7536
    loss: 0.327
     acc: 0.8703
```

테스트 데이터로 긍부정을 예측한다.

```
y_pred = predict_classes(ff, x_test)
```

예측과 실제를 바탕으로 혼돈행렬을 만든다.

```
library(caret)
```

```
Loading required package: lattice
Loading required package: ggplot2
```

```
confusionMatrix(y_pred, y_test)
```

```
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 9498 2936
         1 3002 9564

               Accuracy : 0.7625
                 95% CI : (0.7572, 0.7677)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.525
 Mcnemar's Test P-Value : 0.3989

            Sensitivity : 0.7598
            Specificity : 0.7651
         Pos Pred Value : 0.7639
         Neg Pred Value : 0.7611
             Prevalence : 0.5000
         Detection Rate : 0.3799
   Detection Prevalence : 0.4974
      Balanced Accuracy : 0.7625

       'Positive' Class : 0
```

8.6.3. 순환신경망

순환신경망은 각각의 입력을 받는 LSTM 노드가 있기 때문에 `layer_flatten`이 필요가 없다.
`return_sequences`는 `TRUE`면 입력과 같은 수의 출력을 내어놓는다. `FALSE`면 하나의 출력만 내놓는다.

```
rnn = keras_model_sequential()
```

```
rnn %>%
    layer_embedding(input_dim = NUM.WORDS, output_dim = 8, input_length = MAXLEN) %>%
    layer_lstm(1, activation='sigmoid', return_sequences = F)
```

요약

```
summary(rnn)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 20, 8)             80000
_________________________________________________________________
lstm_1 (LSTM)                (None, 1)                 40
=================================================================
Total params: 80,040
Trainable params: 80,040
Non-trainable params: 0
_________________________________________________________________
```

설정

```
rnn %>% compile(optimizer = optimizer_rmsprop(), loss = 'binary_crossentropy', metrics = 'accuracy')
```

학습

```
history = rnn %>%  fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2,
  callbacks = c(callback_early_stopping(monitor = "val_loss"))
)
```

기록

```
history
```

```
Trained on 20,000 samples, validated on 5,000 samples (batch_size=128, epochs=8)
Final epoch (plot to see history):
val_loss: 0.5935
 val_acc: 0.6988
    loss: 0.5365
     acc: 0.755
```

예측

```
y_rnn = predict_classes(rnn, x_test)
```

혼돈행렬

```
confusionMatrix(y_rnn, y_test)
```

```
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 7717 2808
         1 4783 9692

               Accuracy : 0.6964
                 95% CI : (0.6906, 0.7021)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.3927
 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.6174
            Specificity : 0.7754
         Pos Pred Value : 0.7332
         Neg Pred Value : 0.6696
             Prevalence : 0.5000
         Detection Rate : 0.3087
   Detection Prevalence : 0.4210
      Balanced Accuracy : 0.6964

       'Positive' Class : 0
```

8.6.4. 순환 신경망의 변형

이번에는 LSTM에서 마지막에 32개의 아웃풋을 내보내고 이를 dense 레이어가 받아 1개의 최종예측을 내보내도록 만들어보자.

```
rnn2 = keras_model_sequential()
rnn2 %>%
    layer_embedding(input_dim = NUM.WORDS, output_dim = 8, input_length = MAXLEN) %>%
    layer_lstm(32, activation='tanh', return_sequences = F) %>%
    layer_dense(1, activation='sigmoid')

summary(rnn2)
```

```
_________________________________________________________________________________
Layer (type)                         Output Shape                      Param #
=================================================================================
embedding_4 (Embedding)              (None, 20, 8)                     80000
_________________________________________________________________________________
lstm_3 (LSTM)                        (None, 32)                        5248
_________________________________________________________________________________
dense_3 (Dense)                      (None, 1)                         33
=================================================================================
Total params: 85,281
Trainable params: 85,281
Non-trainable params: 0
_________________________________________________________________________________
```

```
rnn2 %>% compile(optimizer = optimizer_rmsprop(), loss = 'binary_crossentropy', metrics = 'accuracy')

history = rnn2 %>%  fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2,
  callbacks = c(callback_early_stopping(monitor = "val_loss"))
)

history
```

```
Trained on 20,000 samples, validated on 5,000 samples (batch_size=128, epochs=4)
Final epoch (plot to see history):
val_loss: 0.5143
 val_acc: 0.748
    loss: 0.3909
     acc: 0.8238
```

8.6.5. 시퀀스 출력

이번에는 마지막 노드만 출력을 하는 대신 모든 노드가 출력을 하게 하자. 이렇게 하면 출력도 순차적인 형태가 된다. 아래 코드에서는 20개의 노드가 4개씩 출력을 하므로 이를 평평하게 만들어(`layer_flatten`) 80개의 입력으로 바꾸어 마지막 `layer_dense`로 보낸다.

```
rnn3 = keras_model_sequential()
rnn3 %>%
    layer_embedding(input_dim = NUM.WORDS, output_dim = 8, input_length = MAXLEN) %>%
    layer_lstm(4, activation='tanh', return_sequences = T) %>%
    layer_flatten() %>%
    layer_dense(units=1, activation='sigmoid')

summary(rnn3)
```

```
_________________________________________________________________________________
Layer (type)                         Output Shape                      Param #
=================================================================================
embedding_11 (Embedding)             (None, 20, 8)                     80000
_________________________________________________________________________________
lstm_10 (LSTM)                       (None, 20, 4)                     208
_________________________________________________________________________________
flatten_2 (Flatten)                  (None, 80)                        0
_________________________________________________________________________________
dense_7 (Dense)                      (None, 1)                         81
=================================================================================
Total params: 80,289
Trainable params: 80,289
Non-trainable params: 0
_________________________________________________________________________________
```

```
rnn3 %>% compile(optimizer = optimizer_rmsprop(), loss = 'binary_crossentropy', metrics = 'accuracy')

history = rnn3 %>%  fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2,
  callbacks = c(callback_early_stopping(monitor = "val_loss"))
)

history

Trained on 20,000 samples, validated on 5,000 samples (batch_size=128, epochs=5)
Final epoch (plot to see history):
val_loss: 0.5259
 val_acc: 0.7402
    loss: 0.4016
```