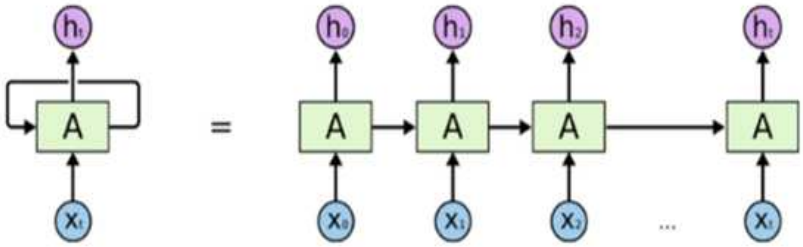
11. RNN(Recurrent Neural Networks)

RNN은 순차적인 데이터(sequential data)를 입력으로 하는 신경망으로 1982년 John Hopfield에 의해 발명된 Hopfield networks가 그 시초라고 할 수 있다[LeCun, 2015]. 그림 11.1은 RNN의 기본 구조를 이해하기 위한 것으로 오른쪽의 그림은 왼쪽의 네트워크를 시간 축에서 펼쳐 놓은 것이다.



An unrolled recurrent neural network.

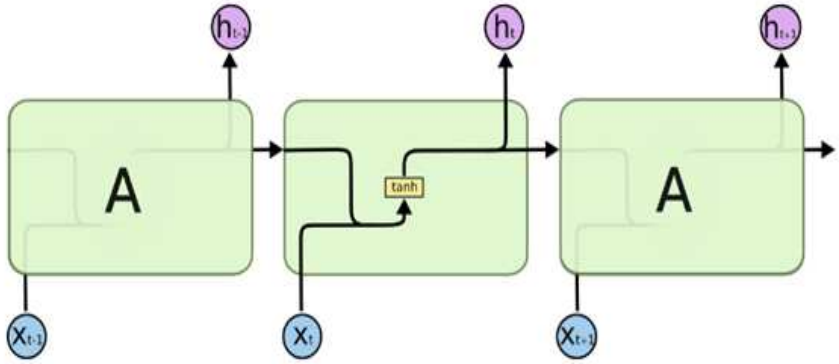
그림 11.1 RNN의 기본적인 구조

오른 쪽의 그림에서 보면 입력 은 이전의 입력인 에 의해서 결정된 Hidden layer의 값과 함께 네트워크에 입력된다. 그리고 여기에서의 Hidden layer의 값은 다음 시간의 네트워크의 입력으로 사용된다. 시간적으로 펼쳤을 때 오른쪽의 그림과 같지만 본래는 왼쪽의 그림처럼 순환 구조의 네트워크일 뿐이므로 weight값이나 bias값 같은 네트워크의 파라미터는 시간에 관계없이 동일하다.

길이가 긴 시계열 데이터의 경우 장기 의존성(Long-Term Dependencies)이 있다고 말하는데 이런 경우 네트워크의 길이가 길어지게 되면서 vanishing gradient 문제(back propagation 훈련과정에서 곱 연산이 계속되어 업데이트할 가 0에 가까워져 훈련이 진행되지 않는 문제)가 발생한다. 이 문제를 해결하기 위한 여러 가지 방법들이 연구되었고, 그 중 LSTM(Long Short Term Memory)은 현재 가장 효과가 좋은 알고리즘으로 널리 쓰이고 있다.

가. LSTM(Long Short Term Memory)

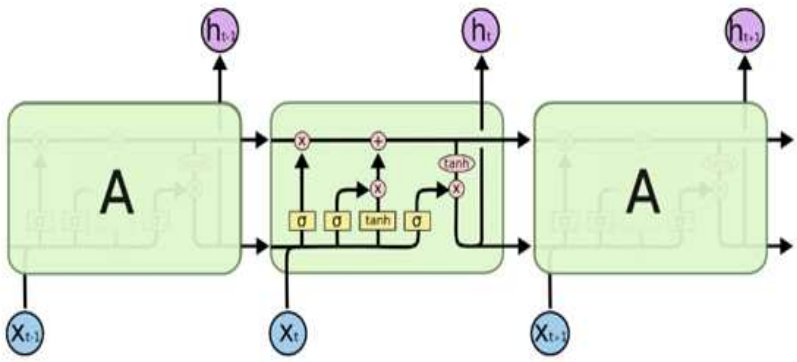
LSTM은 [Hochreiter & Schmidhuber, 1997]이 제안한 알고리즘으로 개선과 대중화가 꾸준히 이루어지고 있다. 아래 그림 11.2는 tanh를 활성함수로 하는 표준 RNN의 Layer를 보여준다. 모든 RNN은 신경망의 연쇄 구조 형태를 갖는다. 즉, 지금까지 살펴본 전형적인 RNN은 특정 시간 t에서의 입력 x_t와 이전 시점에서의 상태변수를 입력으로 하는 하나의 tanh층(layer)으로 구성되어 있었다.



The repeating module in a standard RNN contains a single layer.

그림 11.2 RNN 셀 내부의 기본적인 구조

아래 그림은 LSTM의 구조를 가진 Layer를 보여준다. LSTM도 표준 RNN과 마찬가지로 체인구조를 가진다. 하지만 아주 특별한 방식으로 상호작용하는 4가지 요소가 있다. LSTM 신경망은 연쇄 구조를 갖는다는 점은 RNN과 동일하지만 그림 11.3과 같이 좀 더 복잡한 구조를 갖도록 설계되어 있다. 반복되는 모듈이 특별한 상호작용을 하는 4개의 층을 갖는다.



The repeating module in an LSTM contains four interacting layers.

그림 11.3 LSTM 셀 내부의 기본적인 구조

LSTM의 핵심아이디어는 셀 상태(the cell state)라는 것인데 아래 그림 11.4에서 보이는 수평으로 가로지르는 굵은 선이 바로 그것이다. 일종의 컨베이어 벨트와 같은 모양새이다. 사소한 선형연산을 거쳐서 체인전체를 지나간다. 이로 인해 정보가 큰 변화 없이 다음 체인으로 전달된다.

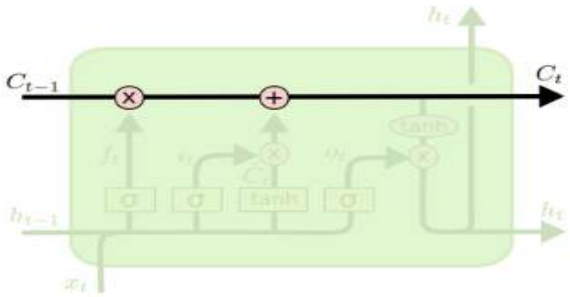


그림 11.4 LSTM의 셀 상태(cell state) 및 흐름

LSTM에는 게이트(gates)라고 하는 셀 상태에 정보를 제거하거나 추가할 수 있는 기능이 있다. 게이트는 선택적으로 정보를 전달할 수 있는 방법이다. 이들은 아래의 그림 11.5와 같이 sigmoid 신경망 층과 점 단위 곱하기 연산으로 이루어져있다.

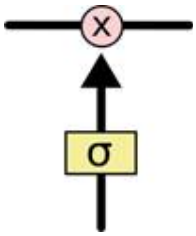
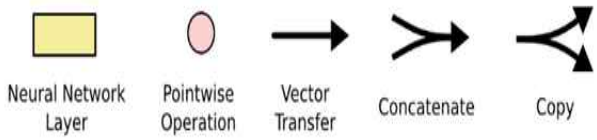


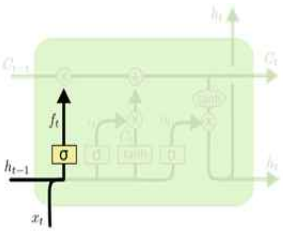
그림 11.5 LSTM 게이트

sigmoid 레이어는 0과 1 사이의 숫자를 출력하여 각 구성 요소의 얼마만큼을 통과시켜야 하는지 결정한다. 값 0은 "아무 것도 통과시키지 말 것"을 의미하지만 값 1은 "모든 것을 통과 시키자"는 의미이다. LSTM에는 셀 상태를 보호하고 제어하기 위한 세 개의 게이트가 있다.

이 신경망의 세부 구조는 그림 11.6의 (b),(c),(d),(e)와 같이 4가지 요소로 구분하여 살펴볼 수 있다.

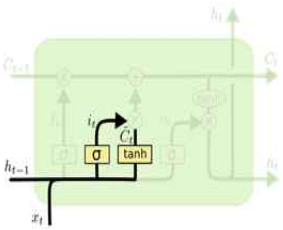


(a) LSTM 모델에 사용된 기호



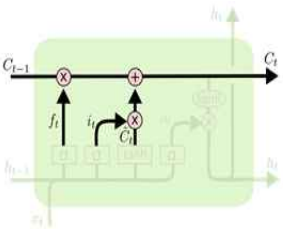
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

(b) 망각 게이트(forget gate)의 구조



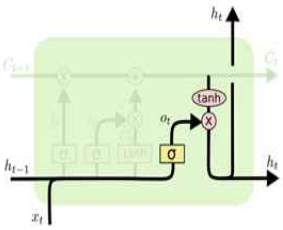
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

(c) 입력 게이트(input gate)와 tanh 층의 구조



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

(d) 상태 변수의 갱신



$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

(e) 출력 게이트(output gate)의 구조

그림 11.6 LSTM 모델의 4가지 요소

LSTM의 첫 번째 단계는 그림 11.6 (b)와 같이, 셀 상태에서 버릴 것을 결정하는 단계이다. 이 결정은 "forget gate layer"라고 불리는 시그모이드 층에 의해서 이루어진다. 이전 상태 h_{t-1}와 현재의 입력 x_t가 sigmoid 층에 의해서 0과 1사이의 값을 출력하고, 0이면 완전히 이전의 셀 상태를 버리게 되고 1이면 셀 상태를 그대로 유지하게 된다.
다음 단계는 셀 상태에 새로운 정보를 추가하는 과정이다. 그림 11.6 (c)와 같이 두 가지 과정이 필요하다. 하나는 업데이트되는 정보를 얼마나 반영할 것인가를 결정하는 sigmoid 층의 값을 결정하는 것이다. 다른 하나는 업데이트되는 정보를 결정하는 tanh 층의 값을 결정하는 것이다.

앞의 두 단계에서 다음 체인으로 넘길 셀 상태는 결정이 나게 된다. 두 단계를 같이 그려보면 그림 11.6 (d)와 같다. 이전 셀 상태 C_{t-1}에 f_t를 곱한 것은 이전 셀 상태를 새로운 셀 상태에 얼마나 반영할 것인가에 대한 결정이다. 현재 입력에 의한 값 $\widetilde{C}_t$에 추가될 값의 반영 정도를 결정하는 값 i_t를 곱한 것은 새롭게 셀 상태에 추가될 값을 결정하는 것이다.

마지막 세 번째 게이트는 출력값을 결정하는 게이트이다. RNN은 시간마다 출력 값이 존재하는데 그것을 결정한다. 그림 11.6 (e)는 출력게이트를 보여준다. 현재 셀 상태 값이 tanh 함수를 거치도록 만들고, 이전 상태 h_{t-1}와 현재의 입력 x_t가 sigmoid 층에 의해서 출력한 0과 1사이의 값을 곱한 결과가 현재 상태의 출력 h_t가 된다.

기존의 RNN은 Backpropagation 알고리즘을 사용하였을 때 네트워크의 구조가 깊어지면(RNN에서는 한 번에 처리할 시계열 데이터가 길어지면 네트워크의 구조가 깊어진다.) 곱 연산의 반복으로 파라미터 업데이트가 힘들어지는 vanishing gradient문제가 발생하는데 LSTM구조를 사용하면 그것을 극복할 수 있다. 시계열 데이터를 다루는 응용에서는 LSTM 신경망이 주로 사용되어 왔는데, 2014년에 이 모델의 복잡한 구조를 좀 더 단순화하되 LSTM 신경망의 장점은 유지할 수 있는 새로운 모델로 GRU(Gated Recurrent Unit) 신경망이 제안되면서 최근 많은 주목을 받고 있다[Cho et el., 2014]. 그림 11.7은 GRU의 구조를 보여주고 있다.

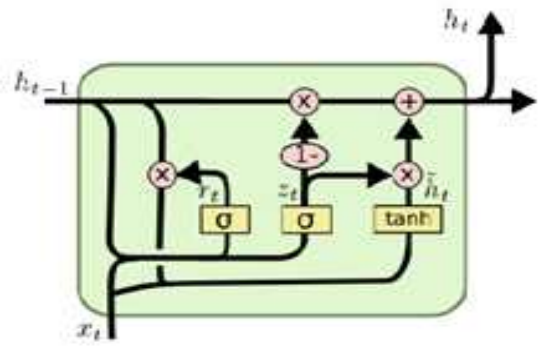


그림 11.7 GRU 모델의 구조

나. Multi-layer RNN

최근 들어 RNN의 은닉층을 여러 층 쌓은 다층 순환신경망(Multi-layer RNN)모델은 은닉 층의 표현력을 확장하고 모델의 성능을 높이는 것으로 알려져 있다. 다층 순환신경망은 가장 기본적인 RNN, LSTM 및 GRU 등의 모델로 다양하게 구성할 수 있다. 레이어를 늘리는 방법은 층을 여러 층으로 쌓아 깊은 구조를 표현할 수도 있으며, 시간을 길게 하여 길이를 길게 표현할 수도 있다. 그림 11.8은 여러 층으로 쌓아서 구현한 다층 RNN 구조를 보여주고 있다.

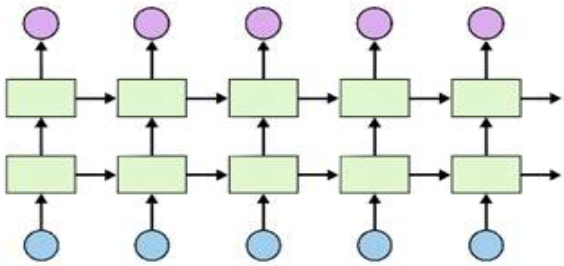


그림 11.8 다층 순환신경망 구조

하지만, 이러한 다층 RNN 구조에서 층이 여러개 쌓인다고 해서 반드시 성능이 좋아지는 것은 아니다. 다층으로 층이 쌓임에 따라, 학습속도가 느려지게 되는 부작용이 발생한다. 즉, 오류역전파과정에서 미분 값의 급격한 증가 또는 급격한 감소 문제가 발생하게 된다. 이러한 문제는 잔차 연결(residual connection)을 추가함으로서 해결될 수 있다.

다. Bidirectional RNN(BRNN)

기존의 RNN은 일반 Neural Networks에 비교해서, 데이터의 이전 상태 정보를 “메모리(Memory)” 형태로 저장할 수 있다는 장점이 있었다. 하지만 시계열 데이터의 현재 시간 이전 정보뿐만 아니라, 이후 정보까지 저장해서 활용할 수 있다면 더 좋은 성능을 기대할 수 있게 된다. 예를 들어 “기온차이가 심한 환절기에 **를 예방하기 위해 체온을 보호하자”문장에서 **에 들어갈 단어를 예측하고자 한다면, 우리는 앞의 정보인 “기온차이”, “환절기”이라는 정보를 가지고 **에 들어갈 단어가 “감기”라고 예측할 수도 있지만, 뒤의 정보인 “예방”과 “체온”라는 단어까지 함께 고려하면 더 높은 확률로 정답이 “감기”라는 것을 예측할 수 있을 것이다.

Bidirectional Recurrent Neural Networks(BRNNs)을 이용하면, 이렇게 이전 정보와 이후 정보를 모두 저장할 수 있게 되는 장점을 가지게 된다. BRNNs 구체적인 architecture는 아래 그림 11.9와 같다. BRNN 역시 가장 기본적인 RNN, LSTM 및 GRU 등의 모델로 다양하게 구성할 수 있다.

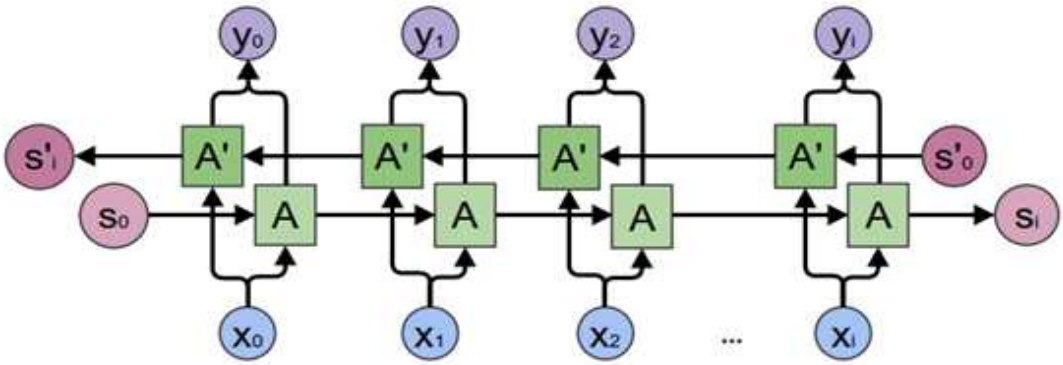


그림 11.9 Bidirectional RNN의 구조

위의 그림에서 알 수 있듯이, BRNN은 2개의 Hidden Layer를 가지고 있다. 전방향 상태(Forward States) 정보를 가지고 있는 Hidden Layer와 후방향 상태(Backward States) 정보를 가지고 있는 Hidden layer가 있고, 이 둘은 서로 연결되어 있지 않다. 하지만, 입력값은 이 2가지 Hidden Layer에 모두 전달되고, Output Layer도 이 2가지 Hidden Layer로 모두 값을 받아서 최종 Output을 계산하게 된다.

라. RNN을 이용한 MNIST 숫자 식별

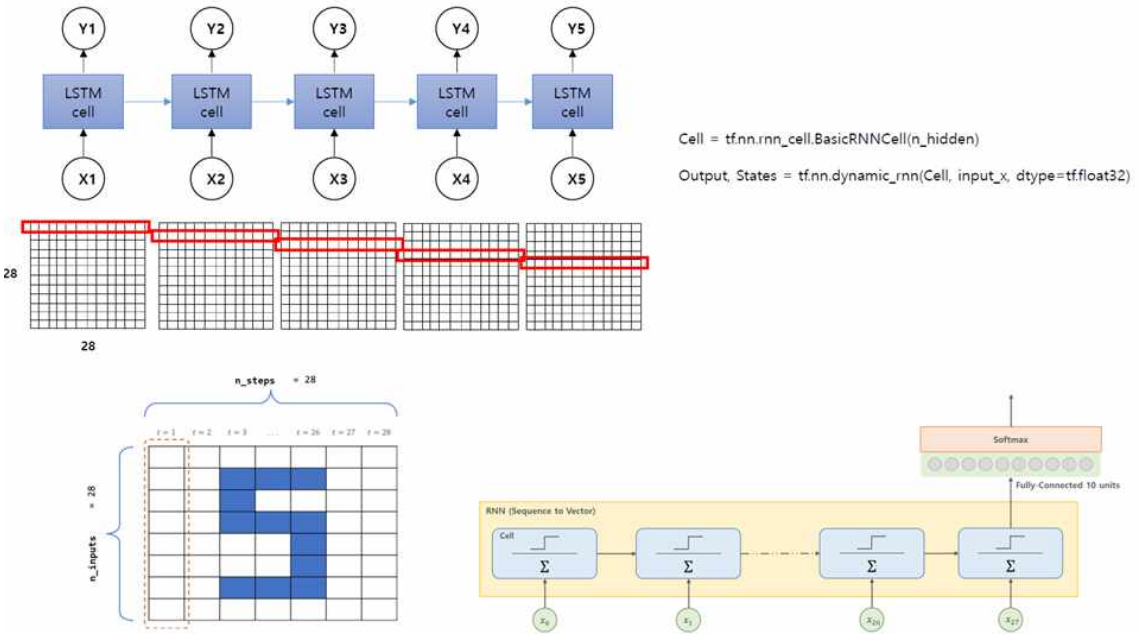


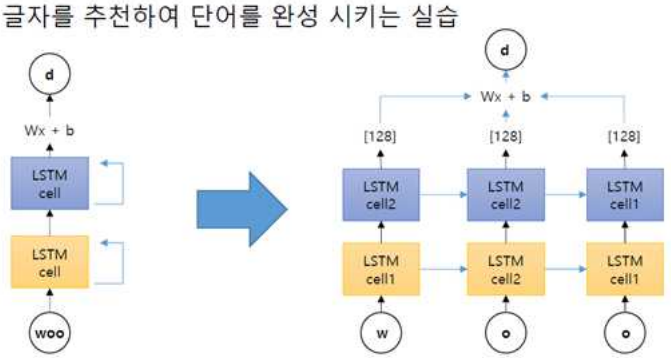
그림 11.10 RNN을 이용한 MNIST 숫자 식별

< Example 11-1 > RNN을 이용한 MNIST 숫자 식별(Basic RNN)

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("./mnist/data/", one_hot=True)

#옵션 설정
learning_rate = 0.001
total_epoch = 30
batch_size = 128

# RNN 은 순서가 있는 자료를 다루므로,
# 한 번에 입력받는 갯수와, 총 몇 단계로 이루어져있는 데이터를 받을지를 설정.
# 이를 위해 가로 픽셀수를 n_input 으로, 세로 픽셀수를 입력 단계인 n_step 으로 설정
n_input = 28
n_step = 28
n_hidden = 128
n_class = 10

#신경망 모델 구성
X = tf.placeholder(tf.float32, [None, n_step, n_input])
Y = tf.placeholder(tf.float32, [None, n_class])
W = tf.Variable(tf.random_normal([n_hidden, n_class]))
b = tf.Variable(tf.random_normal([n_class]))
cell = tf.nn.rnn_cell.BasicRNNCell(n_hidden)
outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)

outputs = tf.transpose(outputs, [1, 0, 2])
outputs = outputs[-1]

model = tf.layers.dense(outputs, 10)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=model, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)

#신경망 모델 학습
sess = tf.Session()
sess.run(tf.global_variables_initializer())
total_batch = int(mnist.train.num_examples/batch_size)

for epoch in range(total_epoch):
    total_cost = 0
    for i in range(total_batch):
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        # X 데이터를 RNN 입력 데이터에 맞게 [batch_size, n_step, n_input] 형태로 변환합니다.
        batch_xs = batch_xs.reshape((batch_size, n_step, n_input))
        _, cost_val = sess.run([optimizer, cost], feed_dict={X: batch_xs, Y: batch_ys})
        total_cost += cost_val

    print('Epoch:', '%04d' % (epoch + 1),
          'Avg. cost =', '{:.3f}'.format(total_cost / total_batch))
print('최적화 완료!')

#결과 확인
is_correct = tf.equal(tf.argmax(model, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
test_batch_size = len(mnist.test.images)
test_xs = mnist.test.images.reshape(test_batch_size, n_step, n_input)
test_ys = mnist.test.labels
print('정확도:', sess.run(accuracy, feed_dict={X: test_xs, Y: test_ys}))
```

마. RNN을 이용한 단어 자동 완성

- 영문자 4개로 구성된 단어를 학습시켜, 3글자만 주어지면
- 나머지 한 글자를 추천하여 단어를 완성 시키는 실습




```
Cell1 = tf.nn.rnn_cell.BasicLSTMCell(n_hidden)
Cell1 = tf.nn.rnn_cell.DropoutWrapper(Cell1, output_keep_prob=0.5)
Cell2 = tf.nn.rnn_cell.BasicLSTMCell(n_hidden)
Multi_cell = tf.nn.rnn_cell.MultiRNNCell([Cell1, Cell2])
Outputs, States = tf.nn.dynamic_rnn(Multi_cell, input_x, dtype = tf.float32)
```

그림 11.11 RNN을 이용한 단어 자동 완성

바. Sequence to sequence를 이용한 번역 테스트

- Sequence to Sequence는 구글이 사용하는 신경망 모델
- 순차적인 정보를 입력받는 신경망(RNN)과 출력하는 신경망을 조합한 모델
- 번역, 챗봇 등 문장을 입력 받아 다른 문장으로 출력

- 모델
  - 입력을 위한 신경망(인코더)와 출력을 위한 신경망(디코더)로 구성
  - Example
    - 인코더 : 원문, 문장
    - 디코더 : 인코더가 번역한 결과물을 입력받아 처리
    - 디코더 출력 결과물을 번역된 결과물과 비교하면서 학습

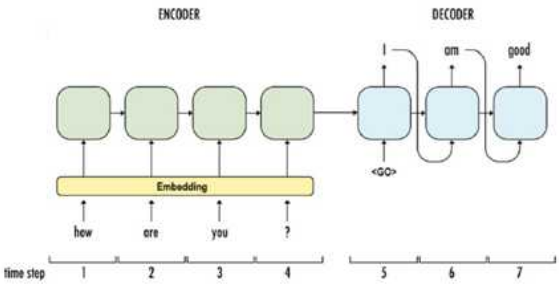


그림 11.12 번역을 위한 Sequence to sequence 모델

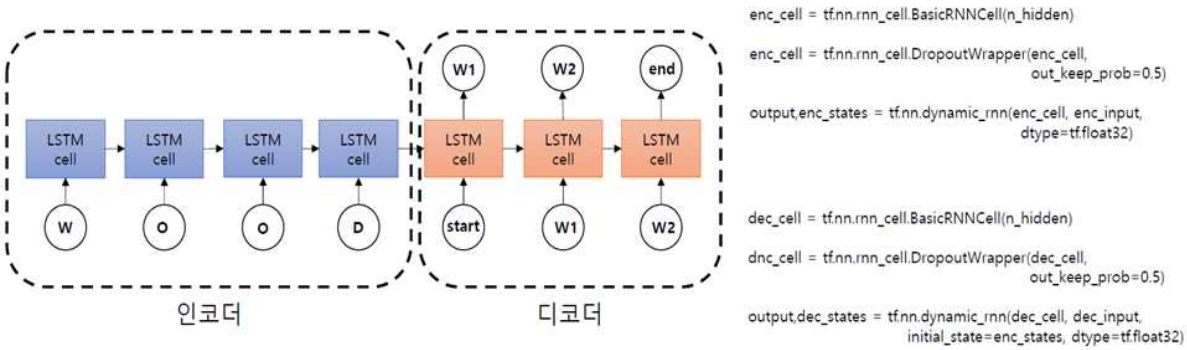


그림 11.13 Sequence to sequence 모델에 사용되는 tf 함수

< Example 11-2 > RNN을 이용한 단어 자동 완성

```
# 자연어 처리나 음성 처리 분야에 많이 사용되는 RNN 의 기본적인 사용법.
# 4개의 글자를 가진 단어를 학습시켜, 3글자만 주어지면 나머지 한 글자를 추천하여 단어를 완성하는 프로그램
import tensorflow as tf
import numpy as np

char_arr = ['a', 'b', 'c', 'd', 'e', 'f', 'g',
            'h', 'i', 'j', 'k', 'l', 'm', 'n',
            'o', 'p', 'q', 'r', 's', 't', 'u',
            'v', 'w', 'x', 'y', 'z']

# one-hot 인코딩 사용 및 디코딩을 하기 위해 연관 배열을 만듭니다.
# {'a': 0, 'b': 1, 'c': 2, ..., 'j': 9, 'k', 10, ...}
num_dic = {n: i for i, n in enumerate(char_arr)}
dic_len = len(num_dic)

# 다음 배열은 입력값과 출력값으로 다음처럼 사용할 것 입니다.
# wor -> X, d -> Y
# woo -> X, d -> Y
seq_data = ['word', 'wood', 'deep', 'dive', 'cold', 'cool', 'load', 'love', 'kiss', 'kind']

def make_batch(seq_data):
    input_batch = []
    target_batch = []

    for seq in seq_data:
        # 여기서 생성하는 input_batch 와 target_batch 는
        # 알파벳 배열의 인덱스 번호 입니다.
        # [22, 14, 17] [22, 14, 14] [3, 4, 4] [3, 8, 21] ...
        input = [num_dic[n] for n in seq[:-1]]
        # 3, 3, 15, 4, 3 ...
        target = num_dic[seq[-1]]
        # one-hot 인코딩을 합니다.
        # if input is [0, 1, 2]:
        # [[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
        #  [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
        #  [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.]]
        input_batch.append(np.eye(dic_len)[input])
        # 지금까지 손실함수로 사용하던 softmax_cross_entropy_with_logits 함수는
        # label 값을 one-hot 인코딩으로 넘겨줘야 하지만,
        # 이 예제에서 사용할 손실 함수인 sparse_softmax_cross_entropy_with_logits 는
        # one-hot 인코딩을 사용하지 않으므로 index 를 그냥 넘겨주면 됩니다.
        target_batch.append(target)

    return input_batch, target_batch

#옵션 설정
learning_rate = 0.01
n_hidden = 128
total_epoch = 30
# 타입 스텝: [1 2 3] => 3
# RNN 을 구성하는 시퀀스의 갯수입니다.
n_step = 3
# 입력값 크기. 알파벳에 대한 one-hot 인코딩이므로 26개가 됩니다.
# 예) c => [0 0 1 0 0 0 0 0 0 0 0 0 ... 0]
# 출력값도 입력값과 마찬가지로 26개의 알파벳으로 분류합니다.
n_input = n_class = dic_len
```

< Example 11-2 > RNN을 이용한 단어 자동 완성(계속)

```
# 신경망 모델 구성
X = tf.placeholder(tf.float32, [None, n_step, n_input])
# 비용함수에 sparse_softmax_cross_entropy_with_logits 을 사용하므로
# 출력값과의 계산을 위한 원본값의 형태는 one-hot vector가 아니라 인덱스 숫자를 그대로 사용하기
때문에
# 다음처럼 하나의 값만 있는 1차원 배열을 입력값으로 받습니다.
# [3] [3] [15] [4] ...
# 기존처럼 one-hot 인코딩을 사용한다면 입력값의 형태는 [None, n_class] 여야합니다.
Y = tf.placeholder(tf.int32, [None])

W = tf.Variable(tf.random_normal([n_hidden, n_class]))
b = tf.Variable(tf.random_normal([n_class]))

# RNN 셀을 생성합니다.
cell1 = tf.nn.rnn_cell.BasicLSTMCell(n_hidden)
# 과적합 방지를 위한 Dropout 기법을 사용합니다.
cell1 = tf.nn.rnn_cell.DropoutWrapper(cell1, output_keep_prob=0.5)
# 여러개의 셀을 조합해서 사용하기 위해 셀을 추가로 생성합니다.
cell2 = tf.nn.rnn_cell.BasicLSTMCell(n_hidden)

# 여러개의 셀을 조합한 RNN 셀을 생성합니다.
multi_cell = tf.nn.rnn_cell.MultiRNNCell([cell1, cell2])

# tf.nn.dynamic_rnn 함수를 이용해 순환 신경망을 만듭니다.
# time_major=True
outputs, states = tf.nn.dynamic_rnn(multi_cell, X, dtype=tf.float32)

# 최종 결과는 one-hot 인코딩 형식으로 만듭니다
outputs = tf.transpose(outputs, [1, 0, 2])
outputs = outputs[-1]
model = tf.matmul(outputs, W) + b

cost = tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(
                logits=model, labels=Y))

optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)

#신경망 모델 학습
sess = tf.Session()
sess.run(tf.global_variables_initializer())

input_batch, target_batch = make_batch(seq_data)

print(target_batch)

for epoch in range(total_epoch):
    _, loss = sess.run([optimizer, cost],
                       feed_dict={X: input_batch, Y: target_batch})

    print('Epoch:', '%04d' % (epoch + 1),
          'cost =', '{:.6f}'.format(loss))

print('최적화 완료!')
```

< Example 11-2 > RNN을 이용한 단어 자동 완성(계속)

```
# 결과 확인
# 레이블값이 정수이므로 예측값도 정수로 변경
prediction = tf.cast(tf.argmax(model, 1), tf.int32)
# one-hot 인코딩이 아니므로 입력값을 그대로 비교합니다.
prediction_check = tf.equal(prediction, Y)
accuracy = tf.reduce_mean(tf.cast(prediction_check, tf.float32))

input_batch, target_batch = make_batch(seq_data)

predict, accuracy_val = sess.run([prediction, accuracy],
                                 feed_dict={X: input_batch, Y: target_batch})

predict_words = []
for idx, val in enumerate(seq_data):
    last_char = char_arr[predict[idx]]
    predict_words.append(val[:3] + last_char)

print('\n=== 예측 결과 ===')
print('입력값:', [w[:3] + ' ' for w in seq_data])
print('예측값:', predict_words)
print('정확도:', accuracy_val)
```

< Example 11-3 > Sequence to sequence를 이용한 번역 테스트

```
# 챗봇, 번역, 이미지 캡셔닝등에 사용되는 시퀀스 학습/생성 모델인 Seq2Seq 을 구현해봅니다.
# 영어 단어를 한국어 단어로 번역하는 프로그램을 만들어봅니다.
import tensorflow as tf
import numpy as np

# S: 디코딩 입력의 시작을 나타내는 심볼
# E: 디코딩 출력을 끝을 나타내는 심볼
# P: 현재 배치 데이터의 time step 크기보다 작은 경우 빈 시퀀스를 채우는 심볼
#     예) 현재 배치 데이터의 최대 크기가 4 인 경우
#         word -> ['w', 'o', 'r', 'd']
#         to   -> ['t', 'o', 'P', 'P']
char_arr = [c for c in 'SEPabcdefghijklmnopqrstuvwxyz단어나무놀이소녀키스사랑']
num_dic = {n: i for i, n in enumerate(char_arr)}
dic_len = len(num_dic)

# 영어를 한글로 번역하기 위한 학습 데이터
seq_data = [['word', '단어'], ['wood', '나무'],
            ['game', '놀이'], ['girl', '소녀'],
            ['kiss', '키스'], ['love', '사랑']]


def make_batch(seq_data):
    input_batch = []
    output_batch = []
    target_batch = []

    for seq in seq_data:
        # 인코더 셀의 입력값. 입력단어의 글자들을 한글자씩 떼어 배열로 만든다.
        input = [num_dic[n] for n in seq[0]]
        # 디코더 셀의 입력값. 시작을 나타내는 S 심볼을 맨 앞에 붙여준다.
        output = [num_dic[n] for n in ('S' + seq[1])]
        # 학습을 위해 비교할 디코더 셀의 출력값. 끝나는 것을 알려주기 위해 마지막에 E 를 붙인다.
        target = [num_dic[n] for n in (seq[1] + 'E')]

        input_batch.append(np.eye(dic_len)[input])
        output_batch.append(np.eye(dic_len)[output])
        # 출력값만 one-hot 인코딩이 아님 (sparse_softmax_cross_entropy_with_logits 사용)
        target_batch.append(target)

    return input_batch, output_batch, target_batch


#########
# 옵션 설정
######
learning_rate = 0.01
n_hidden = 128
total_epoch = 100
# 입력과 출력의 형태가 one-hot 인코딩으로 같으므로 크기도 같다.
n_class = n_input = dic_len
```

< Example 11-3 > Sequence to sequence를 이용한 번역 테스트(계속)

```
#신경망 모델 구성
# Seq2Seq 모델은 인코더의 입력과 디코더의 입력의 형식이 같다.
# [batch size, time steps, input size]
enc_input = tf.placeholder(tf.float32, [None, None, n_input])
dec_input = tf.placeholder(tf.float32, [None, None, n_input])
# [batch size, time steps]
targets = tf.placeholder(tf.int64, [None, None])


# 인코더 셀을 구성한다.
with tf.variable_scope('encode'):
    enc_cell = tf.nn.rnn_cell.BasicRNNCell(n_hidden)
    enc_cell = tf.nn.rnn_cell.DropoutWrapper(enc_cell, output_keep_prob=0.5)

    outputs, enc_states = tf.nn.dynamic_rnn(enc_cell, enc_input,
                                            dtype=tf.float32)

# 디코더 셀을 구성한다.
with tf.variable_scope('decode'):
    dec_cell = tf.nn.rnn_cell.BasicRNNCell(n_hidden)
    dec_cell = tf.nn.rnn_cell.DropoutWrapper(dec_cell, output_keep_prob=0.5)

    # Seq2Seq 모델은 인코더 셀의 최종 상태값을
    # 디코더 셀의 초기 상태값으로 넣어주는 것이 핵심.
    outputs, dec_states = tf.nn.dynamic_rnn(dec_cell, dec_input,
                                            initial_state=enc_states,
                                            dtype=tf.float32)


model = tf.layers.dense(outputs, n_class, activation=None)


cost = tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(
                logits=model, labels=targets))

optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)


#########
# 신경망 모델 학습
######
sess = tf.Session()
sess.run(tf.global_variables_initializer())

input_batch, output_batch, target_batch = make_batch(seq_data)

for epoch in range(total_epoch):
    _, loss = sess.run([optimizer, cost],
                       feed_dict={enc_input: input_batch,
                                  dec_input: output_batch,
                                  targets: target_batch})

    print('Epoch:', '%04d' % (epoch + 1),
          'cost =', '{:.6f}'.format(loss))

print('최적화 완료!')
```

< Example 11-3 > Sequence to sequence를 이용한 번역 테스트(계속)

```
#번역 테스트
# 단어를 입력받아 번역 단어를 예측하고 디코딩하는 함수
def translate(word):
    # 이 모델은 입력값과 출력값 데이터로 [영어단어, 한글단어] 사용하지만,
    # 예측시에는 한글단어를 알지 못하므로, 디코더의 입출력값을 의미 없는 값인 P 값으로 채운다.
    # ['word', 'PPPP']
    seq_data = [word, 'P' * len(word)]

    input_batch, output_batch, target_batch = make_batch([seq_data])

    # 결과가 [batch size, time step, input] 으로 나오기 때문에,
    # 2번째 차원인 input 차원을 argmax 로 취해 가장 확률이 높은 글자를 예측 값으로 만든다.
    prediction = tf.argmax(model, 2)

    result = sess.run(prediction,
                      feed_dict={enc_input: input_batch,
                                 dec_input: output_batch,
                                 targets: target_batch})

    # 결과 값인 숫자의 인덱스에 해당하는 글자를 가져와 글자 배열을 만든다.
    decoded = [char_arr[i] for i in result[0]]

    # 출력의 끝을 의미하는 'E' 이후의 글자들을 제거하고 문자열로 만든다.
    end = decoded.index('E')
    translated = ''.join(decoded[:end])

    return translated


print('\n=== 번역 테스트 ===')

print('word ->', translate('word'))
print('wodr ->', translate('wodr'))
print('love ->', translate('love'))
print('loev ->', translate('loev'))
print('abcd ->', translate('abcd'))
```