순환 신경망으로 순차 데이터 모델링

16.1 순차 데이터 소개

16.1 순차 데이터 소개

- **순차 데이터 소개**
  - 시퀀스 데이터 또는 **시퀀스**(sequence)로 불리는 순차 데이터의 특징에 관해 알아보면서 RNN을 소개하겠음
  - 시퀀스에는 다른 종류의 데이터와는 구별되는 독특한 성질이 있음
  - 시퀀스 데이터를 표현하는 방법과 시퀀스 데이터를 위한 여러 가지 모델을 살펴보겠음
  - RNN과 시퀀스 사이의 관계를 이해하는 데 도움이 될 것

16.1 순차 데이터 소개

- **순차 데이터 모델링: 순서를 고려한다**
  - 다른 데이터 타입과 다르게 시퀀스는 특별함
  - 시퀀스 원소들은 특정 순서가 있으므로 상호 독립적이지 않기 때문임
  - 일반적으로 지도 학습을 위한 머신 러닝 알고리즘은 입력 데이터가 **독립 동일 분포**(Independent and Identically Distributed, IID)라고 가정함
  - 즉, 훈련 샘플이 상호 독립(mutually independent)적이고 같은 분포에 속한다는 의미

16.1 순차 데이터 소개

- **순차 데이터 모델링: 순서를 고려한다**
  - 상호 독립 가정에 기반한다는 점에서 모델에 전달되는 훈련 샘플의 순서는 관계가 없음
  - 예를 들어 n개의 훈련 샘플 $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ..., $\mathbf{x}^{(n)}$으로 구성된 데이터셋이 있다면 머신 러닝 알고리즘을 훈련하기 위해 데이터를 사용하는 순서는 중요하지 않음
  - 이런 예로 이전에 사용해 보았던 붓꽃 데이터셋이 있음
  - 붓꽃 데이터셋에서 각 꽃은 개별적으로 측정되었고 한 꽃의 측정값이 다른 꽃의 측정값에 영향을 미치지 않음

16.1 순차 데이터 소개

- **순차 데이터 모델링: 순서를 고려한다**
  - 시퀀스를 다룰 때는 이런 가정이 유효하지 않음
  - 시퀀스의 정의가 순서를 고려하기 때문임
  - 특정 주식의 가격을 예측하는 것이 이런 경우에 해당
  - 예를 들어 n개의 훈련 샘플을 가지고 있다면 각 훈련 샘플은 특정한 날의 이 주식 가격을 나타냄
  - 다음 3일 동안의 주식 가격을 예측하는 작업이라면 훈련 샘플을 랜덤 순서로 사용하는 것이 아니라 날짜 순서대로 정렬된 이전 주식 가격을 고려하여 트렌드를 감지하는 것이 합리적

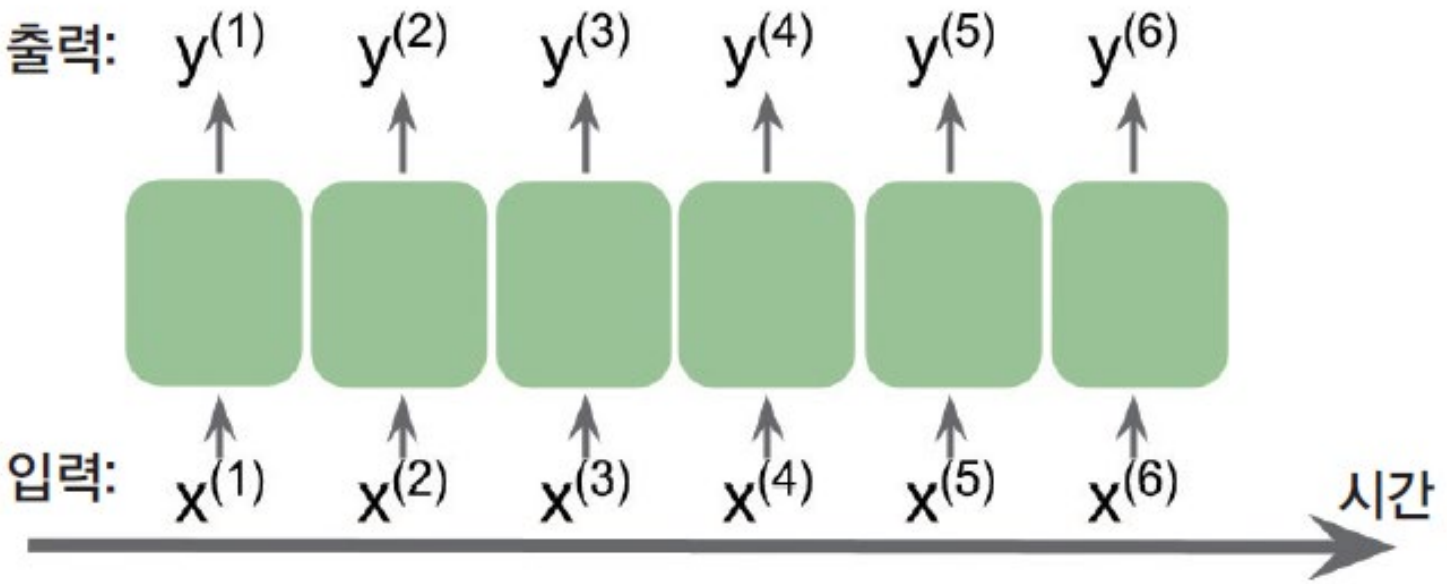
16.1 순차 데이터 소개

- **시퀀스 표현**
  - 순차 데이터에서 데이터 포인트 사이의 순서가 중요하다는 것을 이해했음
  - 머신 러닝 모델에서 이런 순서 정보를 사용할 수 있는 방법을 찾아야 함
  - 이 장에서 시퀀스를 $<\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}>$로 표현하겠음
  - 위 첨자는 샘플 순서를 나타냄
  - **T**는 시퀀스 길이
  - 시퀀스의 좋은 예는 시계열 데이터
  - 여기서 각 샘플 포인트 $\mathbf{x}^{(t)}$는 특정 시간 t에 속함

16.1 순차 데이터 소개

- **시퀀스 표현**
  - 그림 16-1은 시계열 데이터 예를 보여 줌
  - 입력 특성(**x**)과 타깃 레이블(**y**)은 시간 축을 따라 순서대로 나열되어 있음
  - **x**와 **y**는 시퀀스 데이터

▼ **그림 16-1 시계열 데이터 예**



16.1 순차 데이터 소개

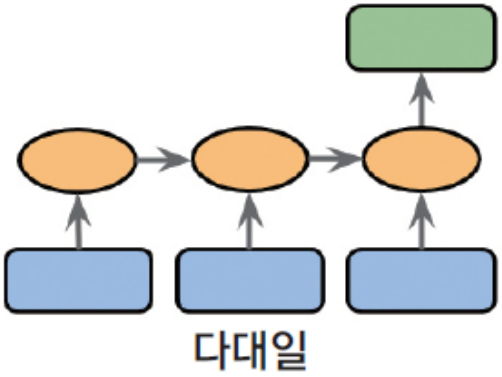
- **시퀀스 표현**
  - 다층 퍼셉트론(MLP)이나 이미지 데이터를 위한 CNN과 같이 지금까지 다루었던 일반 신경망 모델은 훈련 샘플이 서로 독립적이어서 순서 정보와 연관이 없다고 가정함
  - 이런 모델은 이전에 본 훈련 샘플을 기억하는 메모리가 없다고 말함
  - 예를 들어 샘플이 정방향 계산과 역전파 단계를 통과하면 가중치는 훈련 샘플의 처리 순서에 상관없이 독립적으로 업데이트
  - 이와 대조적으로 RNN은 시퀀스 모델링을 위해 고안되었으며 과거 정보를 기억하고 이에 맞추어 새로운 샘플을 처리할 수 있기 때문에 시퀀스 데이터를 다룰 때 장점을 가짐

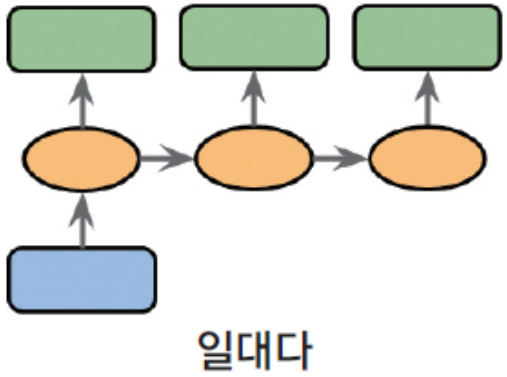
16.1 순차 데이터 소개

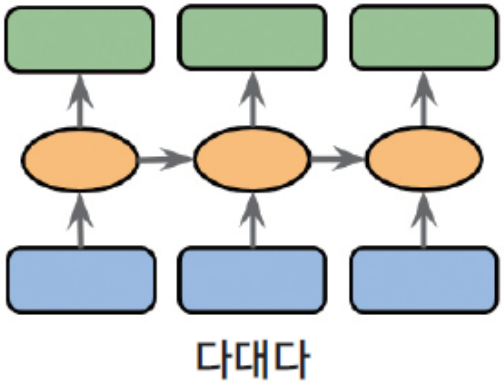
- **시퀀스 모델링의 종류**
  - 시퀀스 모델링에는 언어 번역(예를 들어 영어 텍스트를 독어로 번역), 이미지 캡셔닝(captioning), 텍스트 생성과 같은 매력적인 애플리케이션이 많음
  - 적절한 구조와 방법을 찾으려면 여러 종류의 시퀀스 모델링 작업 사이의 차이점을 이해하고 구별할 수 있어야 함
  - 안드레이 카패시(Andrej Karpathy)가 훌륭하게 설명한 글을 기반으로 그림 16-2에서는 가장 널리 사용하는 시퀀스 모델링 작업을 입력과 출력 데이터의 관계에 따라 요약하고 있음

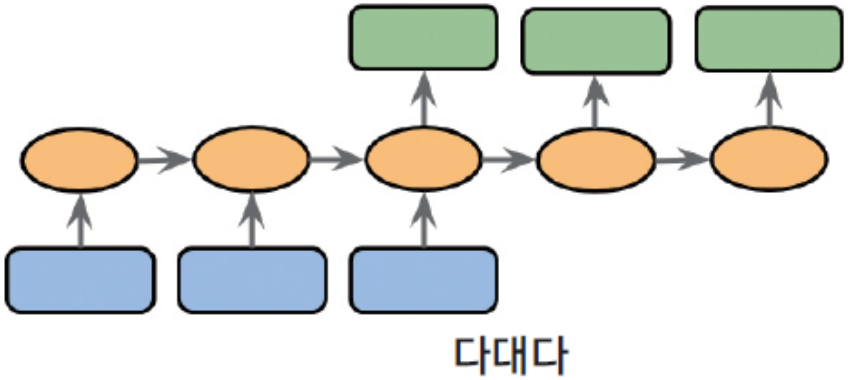
16.1 순차 데이터 소개

▼ 그림 16-2 시퀀스 모델링의 종류









16.1 순차 데이터 소개

- **시퀀스 모델링의 종류**
  - 그림 16-2에 나와 있는 입력과 출력 데이터 사이에 나타나는 여러 관계에 대해 자세히 논의해 보자
  - 입력과 출력 데이터가 시퀀스로 표현되지 않으면 일반 데이터이므로 간단히 다층 퍼셉트론(또는 이 책에서 이전에 소개한 다른 분류 모델)을 사용할 수 있음
  - 입력이나 출력 중 하나가 시퀀스라면 이런 모델링 작업은 다음 중 하나에 속할 것

    다대일(many-to-one): 입력 데이터가 시퀀스이지만 출력은 시퀀스가 아니고 고정

    크기의 벡터나 스칼라

    예를 들어 감성 분석에서 입력은 텍스트(예를 들어 영화 리뷰)이고 출력은 클래스

    레이블(예를 들어 리뷰어가 영화를 좋아하는지 나타내는 레이블)

16.1 순차 데이터 소개

- **시퀀스 모델링의 종류**
  - **일대다**(one-to-many): 입력 데이터가 시퀀스가 아니라 일반적인 형태이고 출력은 시퀀스

    이런 종류의 예로는 이미지 캡셔닝이 있음

    입력이 이미지이고 출력은 이미지 내용을 요약한 영어 문장

  - **다대다**(many-to-many): 입력과 출력 배열이 모두 시퀀스

    이런 종류는 입력과 출력이 동기적인지에 따라 더 나눌 수 있음

    동기적인 다대다 모델링 작업의 예는 각 프레임을 레이블링하는 비디오 분류

    지연이 있는 다대다 모델의 예는 한 언어에서 다른 언어로 번역하는 작업

    예를 들어 독일어로 번역하기 전에 전체 영어 문장을 읽어 처리

16.2 시퀀스 모델링을 위한 RNN

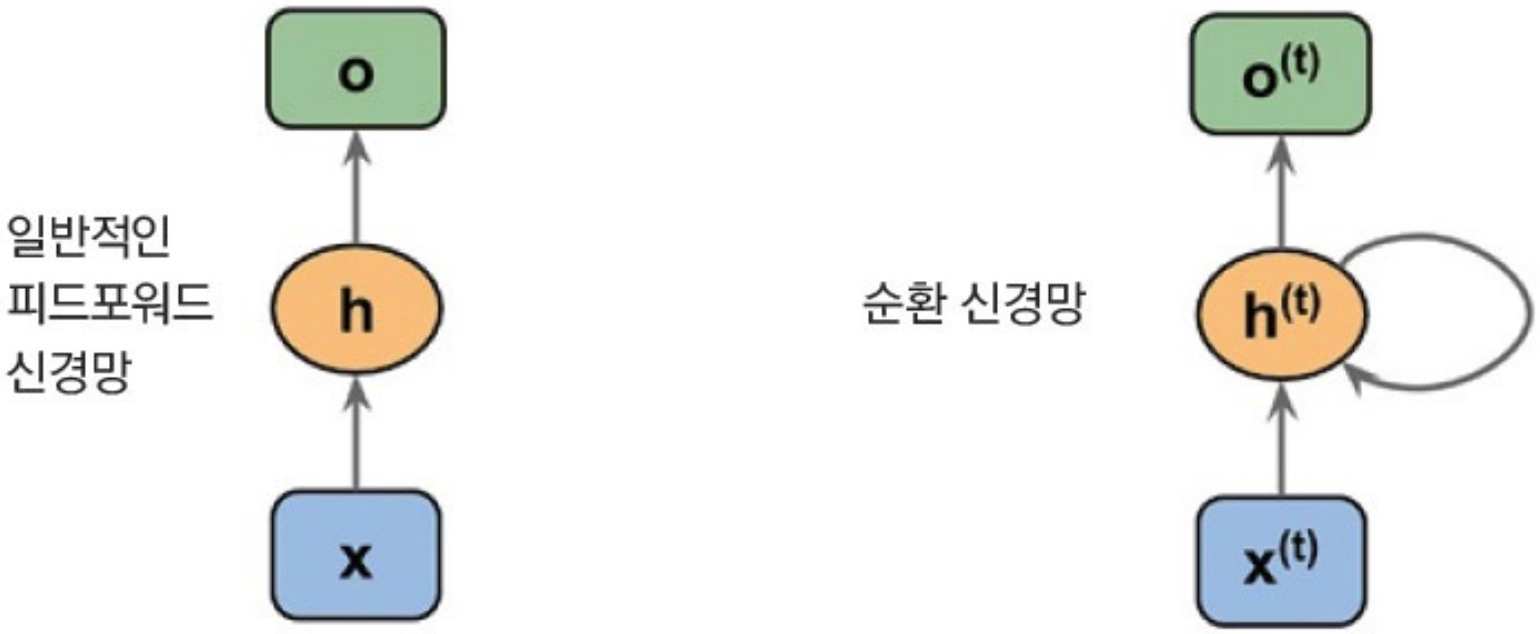
16.2 시퀀스 모델링을 위한 RNN

- **시퀀스 모델링을 위한 RNN**
  - 이 절에서 텐서플로로 RNN을 구현하기 전에 RNN의 주요 개념을 알아보겠음
  - 먼저 시퀀스 데이터를 모델링하기 위한 재귀적 구성 요소를 가진 전형적인 RNN 구조를 살펴보겠음
  - 이를통해 RNN 훈련의 어려움을 이해하고 이런 문제를 해결하기 위한 LSTM이나 GRU 같은 기법을 설명

16.2 시퀀스 모델링을 위한 RNN

- **RNN 반복 구조 이해**
  - RNN 구조부터 설명해 보자
  - 그림 16-3에서 일반적인 피드포워드 신경망과 RNN을 비교하기 위해 나란히 놓았음

▼ **그림 16-3 피드포워드 신경망과 순환 신경망 비교**



16.2 시퀀스 모델링을 위한 RNN

- **RNN 반복 구조 이해**
  - 두 네트워크 모두 하나의 은닉층만 있음
  - 그림 16-3에서는 유닛을 표시하지 않았음
  - 입력층(**x**), 은닉층(**h**), 출력층(**o**) 모두 벡터이고 여러 개의 유닛이 있다고 가정함

16.2 시퀀스 모델링을 위한 RNN

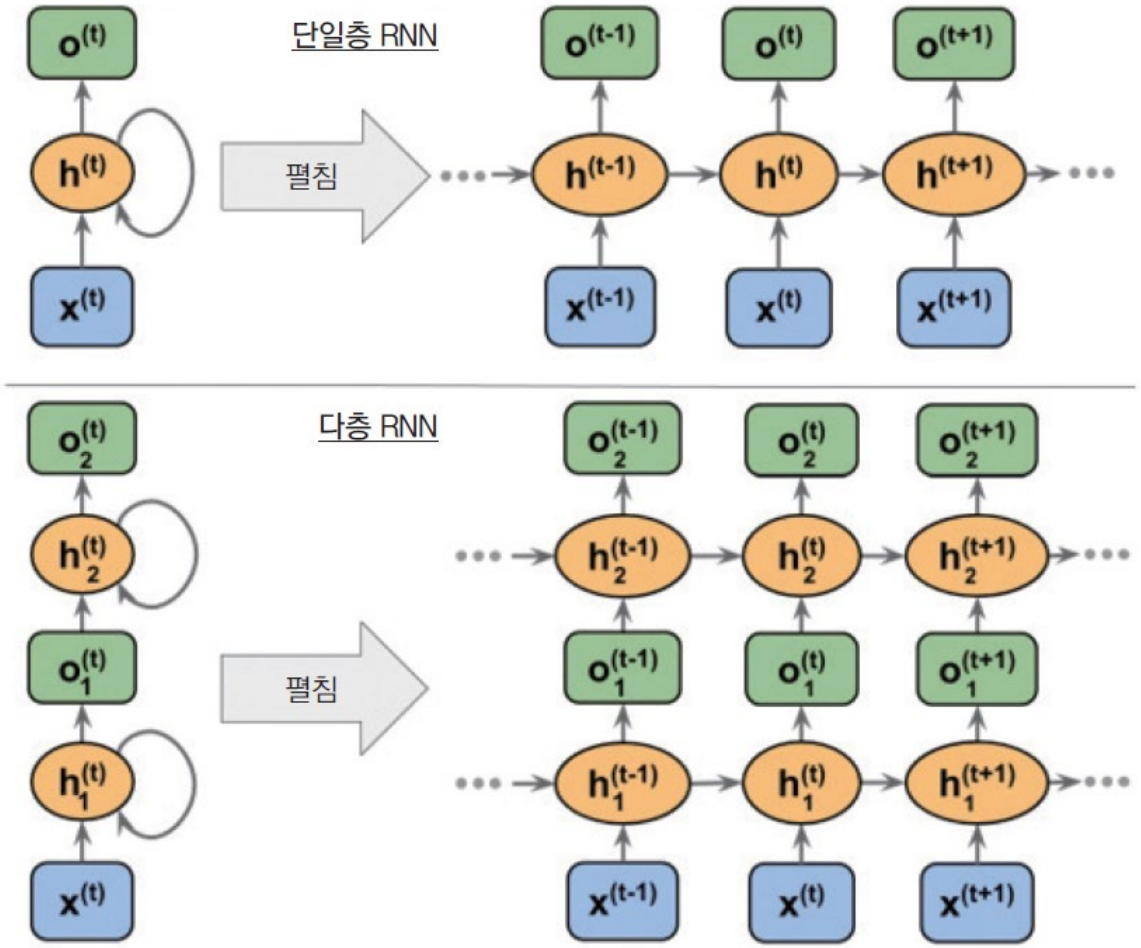
- **RNN 반복 구조 이해**
  - 기본 피드포워드 네트워크에서 정보는 입력에서 은닉층으로 흐른 후 은닉층에서 출력층으로 전달
  - 반면 순환 네트워크에서는 은닉층이 현재 타임 스텝(time step)의 입력층과 이전 타임 스텝의 은닉층으로부터 정보를 받음
  - 인접한 타임 스텝의 정보가 은닉층에 흐르기 때문에 네트워크가 이전 이벤트를 기억할 수 있음
  - 이 정보 흐름을 보통 루프(loop)로 표시
  - 그래프 표기법에서는 **순환 에지**(recurrent edge)라고도 하기 때문에 RNN 구조 이름이 여기서 유래

16.2 시퀀스 모델링을 위한 RNN

- **RNN 반복 구조 이해**
  - 다층 퍼셉트론과 비슷하게 RNN은 여러 개의 은닉층으로 구성할 수 있음
  - 하나의 은닉층을 가진 RNN을 관례적으로 단일층 RNN이라고 말함
  - 아달린이나 로지스틱 회귀와 같이 은닉층이 없는 단일층 신경망과 혼동하지 말자
  - 그림 16-4는 하나의 은닉층을 가진 RNN(위)과 두 개의 은닉층을 가진 RNN(아래)을 보여 줌

16.2 시퀀스 모델링을 위한 RNN

▼ 그림 16-4 하나의 은닉층을 가진 RNN과 여러 개의 은닉층을 가진 RNN



16.2 시퀀스 모델링을 위한 RNN

- **RNN 반복 구조 이해**
  - 일반 신경망의 은닉 유닛은 입력층에 연결된 최종 입력 하나만 받음
  - 반면 RNN의 은닉 유닛은 두 개의 다른 입력을 받음
  - 입력층으로부터 받은 입력과 같은 은닉층에서 t -1 타임 스텝의 활성화 출력을 받음
  - 맨 처음 t = 0에서는 은닉 유닛이 0 또는 작은 난수로 초기화됨
  - t > 0인 타임 스텝에서는 은닉 유닛이 현재 타입 스텝의 데이터 포인트 $\mathbf{x}^{(t)}$와 이전 타임 스텝 t–1의 은닉 유닛 값 $\mathbf{h}^{(t-1)}$을 입력으로 받음

16.2 시퀀스 모델링을 위한 RNN

- **RNN 반복 구조 이해**
  - 비슷하게 다층 RNN의 정보 흐름을 다음과 같이 요약할 수 있음

    - **layer = 1:** 은닉층의 출력을 $\mathbf{h}_1^{(t)}$로 표현
      데이터 포인트 $\mathbf{x}^{(t)}$와 이 은닉층의 이전 타임 스텝 출력 $\mathbf{h}_1^{(t-1)}$을 입력으로 받음
    - **layer = 2:** 두 번째 은닉층의 $\mathbf{h}_2^{(t)}$는 이전 층의 현재 타임 스텝 출력($\mathbf{o}_1^{(t)}$)와 이 은닉층의
      이전 타임 스텝 출력 $\mathbf{h}_2^{(t-1)}$ 을 입력으로 받음

  - 이 경우 각 은닉층은 시퀀스를 입력으로 받기 때문에 마지막을 제외하고 모든 순환 층은 시퀀스를 출력으로 반환해야 함(즉, return_sequences=True로 설정해야 함)
  - 마지막 순환 층의 동작은 문제 유형에 따라 결정

16.2 시퀀스 모델링을 위한 RNN

- **RNN의 활성화 출력 계산**
  - RNN의 구조와 일반적인 정보 흐름을 이해했음
  - 이제 구체적으로 은닉층과 출력층의 실제 활성화 출력을 계산해 보자
  - 간소하게 나타내기 위해 하나의 은닉층만 고려하지만 다층 RNN에도 동일한 개념이 적용

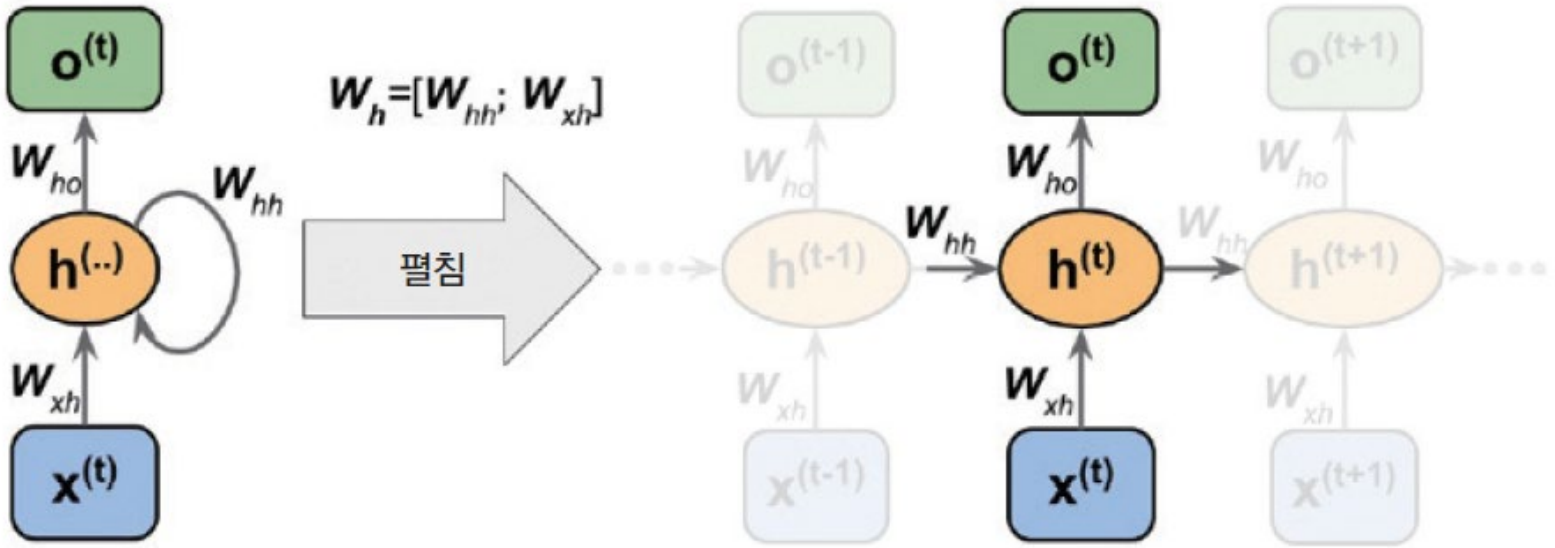
16.2 시퀀스 모델링을 위한 RNN

- **RNN의 활성화 출력 계산**
  - 그림 16-4에서 유향 에지(directed edge)(층 사이 연결과 순환 연결)는 가중치 행렬과 연관됨
  - 이 가중치는 특정 시간 t에 종속적이지 않고 전체 시간 축에 공유
  - 단일층 RNN의 각 가중치는 다음과 같음

    - $\mathbf{W}_{xh}$: 입력 $\mathbf{x}^{(t)}$와 은닉층 $\mathbf{h}$ 사이의 가중치 행렬
    - $\mathbf{W}_{hh}$: 순환 에지에 연관된 가중치 행렬
    - $\mathbf{W}_{ho}$: 은닉층과 출력층 사이의 가중치 행렬

16.2 시퀀스 모델링을 위한 RNN

▼ 그림 16-5 순환 신경망의 가중치



16.2 시퀀스 모델링을 위한 RNN

- **RNN의 활성화 출력 계산**
  - 구현에 따라 가중치 행렬 $\mathbf{W}_{xh}$와 $\mathbf{W}_{hh}$를 합쳐 연결된 행렬 $\mathbf{W}_h=[\mathbf{W}_{xh}; \mathbf{W}_{hh}]$를 사용
  - 나중에 이절에서 이런 표기법을 사용
  - 활성화 출력의 계산은 기본적인 다층 퍼셉트론이나 다른 피드포워드 신경망과 매우 비슷함
  - 은닉층의 최종 입력 $\mathbf{z}_h$(활성화 함수를 통과하기 전의 값)는 선형 조합으로 계산
  - 즉, 가중치 행렬과 대응되는 벡터를 곱해서 더한 후 절편 유닛을 더함

$$\boldsymbol{z}_h^{(t)} = \boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h$$

16.2 시퀀스 모델링을 위한 RNN

- **RNN의 활성화 출력 계산**
  - 그다음 타임 스텝 t에서 은닉층의 활성화를 계산

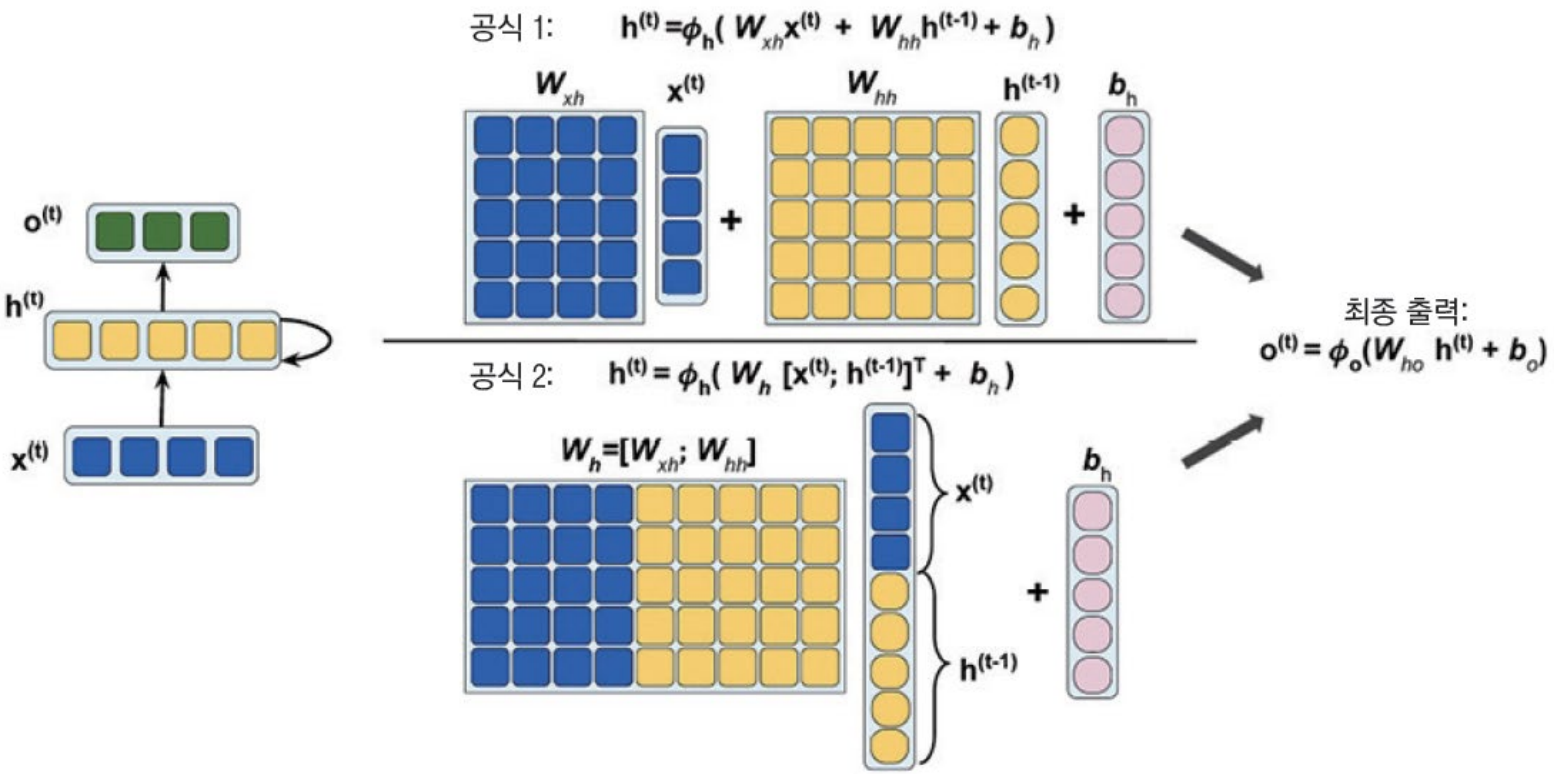
$$\boldsymbol{h}^{(t)} = \phi_h\left(\boldsymbol{z}_h^{(t)}\right) = \phi_h\left(\boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h\right)$$

  - 여기서 $\mathbf{b}_h$는 은닉 유닛의 절편 벡터이고 $\phi_h(\cdot)$는 은닉층의 활성화 함수
  - 가중치 행렬을 $\mathbf{W}_h=[\mathbf{W}_{xh}; \mathbf{W}_{hh}]$처럼 연결하면 은닉 유닛의 계산 공식은 다음과 같이 바뀜

$$\boldsymbol{h}^{(t)} = \phi_h\left([\boldsymbol{W}_{xh}; \boldsymbol{W}_{hh}]\begin{bmatrix}\boldsymbol{x}^{(t)} \\ \\ \boldsymbol{h}^{(t-1)}\end{bmatrix} + \boldsymbol{b}_h\right)$$

16.2 시퀀스 모델링을 위한 RNN

▼ 그림 16-6 순환 신경망의 계산

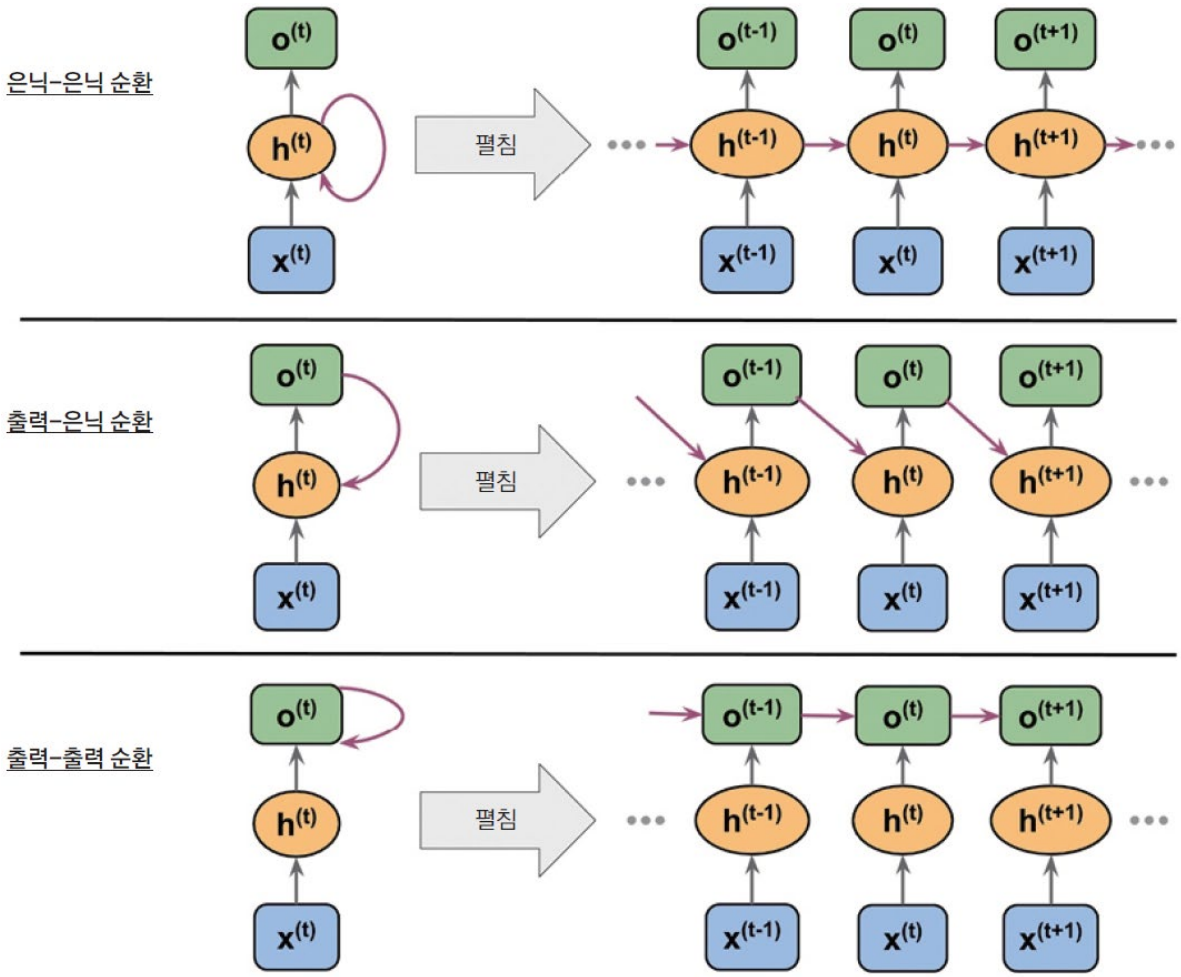


16.2 시퀀스 모델링을 위한 RNN

- **은닉 순환과 출력 순환**
  - 지금까지 은닉층에 순환 성질이 있는 순환 신경망을 보았음
  - 출력층에서 오는 순환 연결을 가진 모델도 있음
  - 이런 경우에 이전 타임 스텝의 출력층에서 오는 활성화 $\mathbf{o}^{t-1}$을 추가하는 방법은 다음 둘 중 하나
    - 현재 타임 스텝에서 은닉층 $\mathbf{h}^{t}$에 추가(그림 16-7의 출력-은닉 순환)
    - 현재 타임 스텝에서 출력층 $\mathbf{o}^{t}$에 추가(그림 16-7의 출력-출력 순환)

16.2 시퀀스 모델링을 위한 RNN

▼ 그림 16-7 순환 방식에 따른 RNN



16.2 시퀀스 모델링을 위한 RNN

- **은닉 순환과 출력 순환**
  - 그림 16-7에 있듯이 이런 구조의 차이는 순환 연결에 있음
  - 앞서 사용한 표기법을 따라서 순환 연결에 관련된 가중치는 은닉-은닉 순환의 경우 $\mathbf{W}_{hh}$로, 출력-은닉 순환의 경우 $\mathbf{W}_{oh}$로, 출력-출력 순환의 경우 $\mathbf{W}_{oo}$로 표시
  - 일부 논문이나 글에서 순환 연결 가중치를 $\mathbf{W}_{rec}$로 표시하기도 함

16.2 시퀀스 모델링을 위한 RNN

- **은닉 순환과 출력 순환**
  - 실제로 어떻게 동작하는지 보기 위해 이 순환 타입 중 하나의 정방향 계산을 수동으로 수행해 보겠음
  - 텐서플로 케라스 API의 SimpleRNN 클래스로 출력-출력 순환과 비슷한 순환 층을 정의 할 수 있음
  - 다음 코드에서 SimpleRNN으로 순환 층을 만들고 길이가 3인 입력 시퀀스에서 정방향 계산을 수행하여 출력을 만듦
  - 또한, 수동으로 정방향 계산을 수행하여 SimpleRNN의 결과와 비교해 보자

16.2 시퀀스 모델링을 위한 RNN

- **은닉 순환과 출력 순환**
  - 먼저 층을 만들고 수동 계산을 위해 가중치를 저장

```
>>> import tensorflow as tf
>>> tf.random.set_seed(1)
>>> rnn_layer = tf.keras.layers.SimpleRNN(
...     units=2, use_bias=True,
...     return_sequences=True)
>>> rnn_layer.build(input_shape=(None, None, 5))
>>> w_xh, w_oo, b_h = rnn_layer.weights
>>> print('W_xh 크기:', w_xh.shape)
>>> print('W_oo 크기:', w_oo.shape)
>>> print('b_h  크기:', b_h.shape)
W_xh 크기: (5, 2)
W_oo 크기: (2, 2)
b_h  크기: (2,)
```

16.2 시퀀스 모델링을 위한 RNN

- **은닉 순환과 출력 순환**
  - 이 층의 입력 크기는 (None, None, 5)
  - 첫 번째 차원은 배치 차원(가변적인 배치 크기를 위해 None으로 지정)이고 두 번째 차원은 시퀀스에 해당(가변적인 시퀀스 길이를 위해 None으로 지정함)
  - 마지막 차원은 특성에 해당
  - return_sequences=True로 지정했으므로 길이가 3인 시퀀스를 입력하면 출력 시퀀스 $<\mathbf{o}^{(0)}, \mathbf{o}^{(1)}, \mathbf{o}^{(2)}>$가 나옴
  - 그렇지 않으면 최종 출력 $\mathbf{o}^{(2)}$만 반환

16.2 시퀀스 모델링을 위한 RNN

- **은닉 순환과 출력 순환**
  - rnn_layer의 정방향 계산을 수행하고 각 타임 스텝에서 수동으로 출력을 계산하여 비교해 보자

```
>>> x_seq = tf.convert_to_tensor(
...     [[1.0]*5, [2.0]*5, [3.0]*5],
...     dtype=tf.float32)
>>> ## SimpleRNN의 출력:
>>> output = rnn_layer(tf.reshape(x_seq, shape=(1, 3, 5)))
>>> ## 수동으로 출력 계산하기:
>>> out_man = []
>>> for t in range(len(x_seq)):
```

16.2 시퀀스 모델링을 위한 RNN

- 은닉 순환과 출력 순환

```
...     xt = tf.reshape(x_seq[t], (1, 5))
...     print('타임 스텝 {} =>'.format(t))
...     print('   입력           :', xt.numpy())
...
...     ht = tf.matmul(xt, w_xh) + b_h
...     print('   은닉           :', ht.numpy())
...
...     if t>0:
...         prev_o = out_man[t-1]
...     else:
...         prev_o = tf.zeros(shape=(ht.shape))
...     ot = ht + tf.matmul(prev_o, w_oo)
```

16.2 시퀀스 모델링을 위한 RNN

- **은닉 순환과 출력 순환**

```
...     ot = tf.math.tanh(ot)
...     out_man.append(ot)
...     print('   출력 (수동)     :', ot.numpy())
...     print('   SimpleRNN 출력 :'.format(t),
...           output[0][t].numpy())
...     print()
타임 스텝 0 =>
   입력           : [[1. 1. 1. 1. 1.]]
   은닉           : [[0.41464037 0.96012145]]
   출력 (수동)     : [[0.39240566 0.74433106]]
   SimpleRNN 출력 : [0.39240566 0.74433106]

타임 스텝 1 =>
   입력           : [[2. 2. 2. 2. 2.]]
   은닉           : [[0.82928073 1.9202429 ]]
   출력 (수동)     : [[0.80116504 0.9912947 ]]
   SimpleRNN 출력 : [0.80116504 0.9912947 ]
```

16.2 시퀀스 모델링을 위한 RNN

- **은닉 순환과 출력 순환**

```
타임 스텝 2 =>
    입력            : [[3. 3. 3. 3. 3.]]
    은닉            : [[1.243921  2.8803642]]
    출력 (수동)     : [[0.95468265 0.9993069 ]]
    SimpleRNN 출력 : [0.95468265 0.9993069 ]
```

16.2 시퀀스 모델링을 위한 RNN

- **은닉 순환과 출력 순환**
  - 수동으로 정방향 계산을 할 때 하이퍼볼릭 탄젠트(tanh) 활성화 함수를 사용
  - SimpleRNN에서 이 함수를 사용하기 때문임(활성화 함수 기본값)
  - 출력 결과에서 볼 수 있듯이 수동으로 계산한 것과 SimpleRNN 층의 각 타임 스텝 출력이 정확히 동일함
  - 이 예제를 통해 순환 층의 미스테리가 풀렸기를 바람

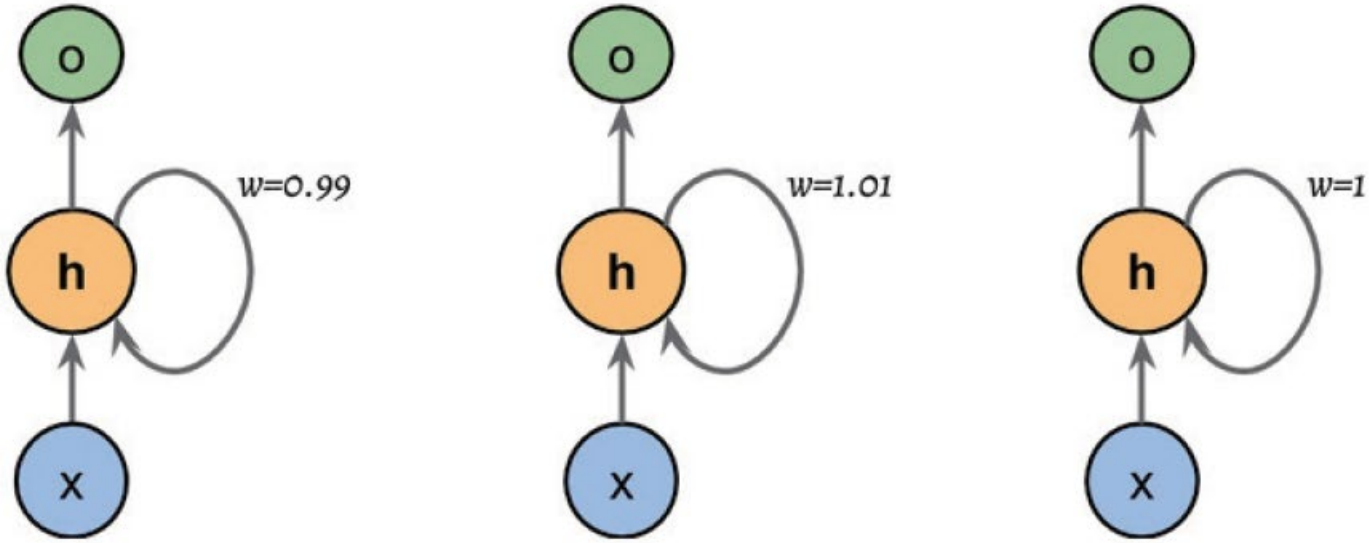
16.2 시퀀스 모델링을 위한 RNN

- **긴 시퀀스 학습의 어려움**
  - 앞서 간략히 소개한 BPTT(BackPropagation Through Time)는 새로운 문제를 야기시킴
  - 손실 함수의 그레이디언트를 계산할 때 곱셈 항 $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$ 때문에 소위 **그레이디언트 폭주**(exploding gradient) 또는 **그레이디언트 소실**(vanishing gradient) 문제가 발생
  - 이 문제를 그림 16-8에서 하나의 은닉 유닛이 있는 예를 들어 설명하겠음

16.2 시퀀스 모델링을 위한 RNN

▼ **그림 16-8 그레이디언트 소실과 폭주**

그레이디언트 소실: $|w_{hh}| < 1$ 그레이디언트 폭주: $|w_{hh}| > 1$ 그레이디언트 유지: $|w_{hh}| = 1$



16.2 시퀀스 모델링을 위한 RNN

- **긴 시퀀스 학습의 어려움**
  - 기본적으로 $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$ 는 t -k개의 곱셈으로 이루어짐
  - 즉, 가중치 w가 t -k번 곱해져 w^{t-k}가 됨
  - 결국 |w| < 1이면 t -k가 클 때 이 항이 매우 작아짐
  - 반면 순환 에지의 가중치 값이 |w| > 1이면 t -k가 클 때 w^{t-k}가 매우 커짐
  - t -k 값이 크다는 것은 긴 시간 의존성을 가진다는 의미
  - 그레이디언트 소실이나 폭주를 피하는 단순한 방법은 |w| = 1이 되도록 만드는 것
  - 이에 대한 자세한 내용은 관련 논문을 참고

16.2 시퀀스 모델링을 위한 RNN

- **긴 시퀀스 학습의 어려움**
  - 실전에서 이 문제에 대한 세 가지 해결책은 다음과 같음

    - 그레이디언트 클리핑
    - TBPTT(Truncated BackPropagation Through Time)
    - LSTM(Long Short-Term Memory)

16.2 시퀀스 모델링을 위한 RNN

- **긴 시퀀스 학습의 어려움**
  - 그레이디언트 클리핑을 사용하면 그레이디언트의 임계 값을 지정하고 이 값을 넘어서는 경우 임계 값을 그레이디언트 값으로 사용
  - 이와 달리 TBPTT는 정방향 계산 후 역전파될 수 있는 타임 스텝의 횟수를 제한
  - 예를 들어 시퀀스가 100개의 원소 또는 스텝을 가지더라도 가장 최근 20번의 타임 스텝만 역전파할 수 있음

16.2 시퀀스 모델링을 위한 RNN

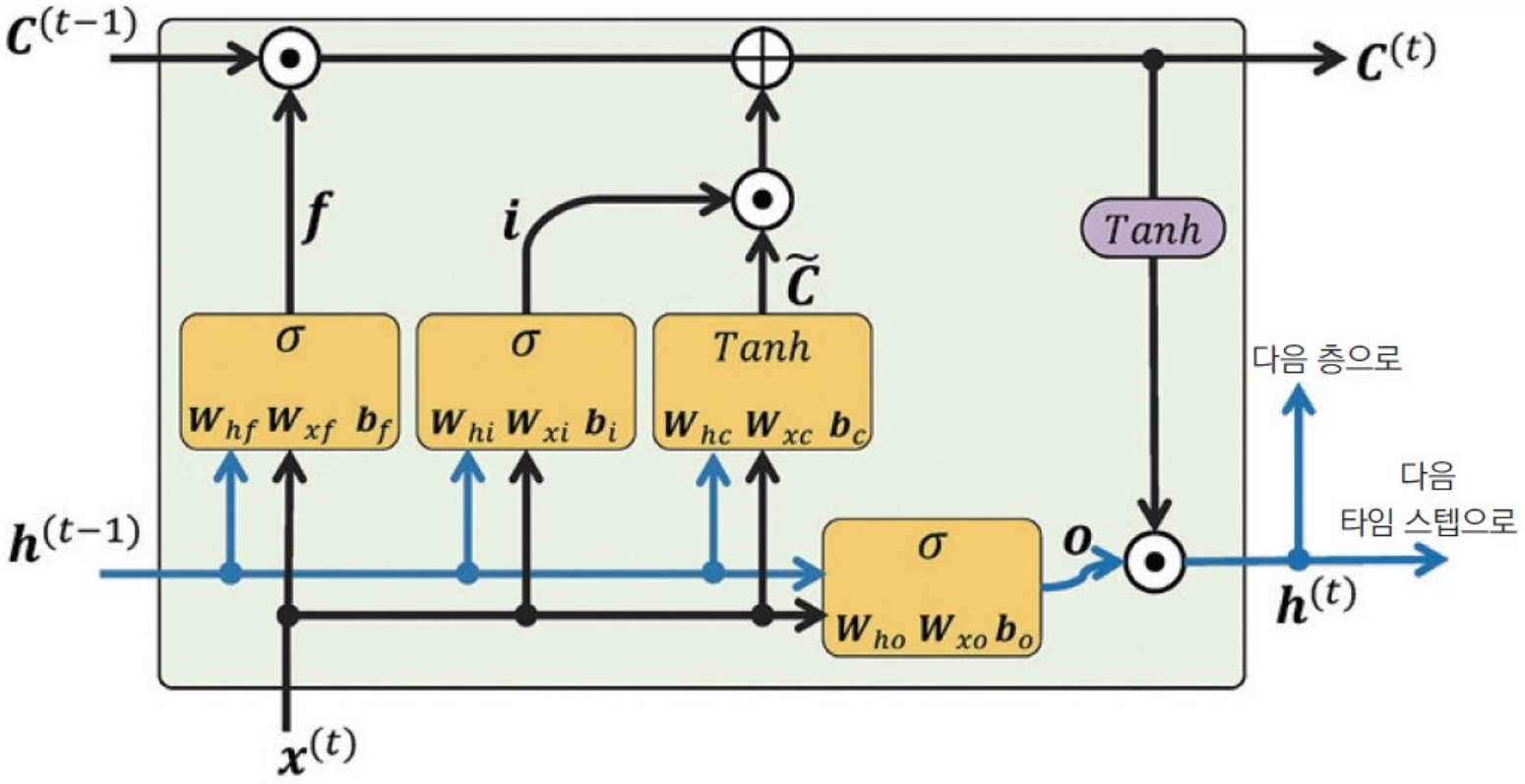
- **긴 시퀀스 학습의 어려움**
  - 그레이디언트 클리핑과 TBPTT가 그레이디언트 폭주 문제를 해결할 수 있지만 그레이디언트가 시간을 거슬러 적절하게 가중치가 업데이트될 수 있는 스텝을 제한
  - 다른 방법으로 1997년 호크라이터(Hochreiter)와 슈미트후버(Schmidhuber)가 고안한 LSTM은 메모리 셀을 사용해서 그레이디언트 소실과 폭주 문제를 극복하여 긴 시퀀스를 성공적으로 모델링할 수 있음

16.2 시퀀스 모델링을 위한 RNN

- **LSTM 셀**
  - LSTM은 그레이디언트 소실 문제를 극복하기 위해 처음 소개
  - 10 LSTM의 기본 구성 요소는 일반 RNN의 은닉층을 표현 또는 대체하는 **메모리 셀**(memory cell)
  - 그레이디언트 소실과 폭주 문제를 극복하기 위해 각 메모리 셀에 적절한 가중치 w = 1을 유지하는 순환 에지가 있음
  - 이 순환 에지의 출력을 **셀 상태**(cell state)라고 함

16.2 시퀀스 모델링을 위한 RNN

▼ 그림 16-9 LSTM 셀



16.2 시퀀스 모델링을 위한 RNN

- **LSTM 셀**
  - 이전 타임 스텝의 셀 상태 $\mathbf{C}^{(t-1)}$은 어떤 가중치와도 직접 곱해지지 않고 변경되어 현재 타임 스텝의 셀 상태 $\mathbf{C}^{(t)}$을 얻음
  - 메모리 셀의 정보 흐름은 다음에 기술된 몇 개의 연산 유닛(또는 게이트(gate))으로 제어
  - 그림 16-9에서 $\odot$ 는 원소별 곱셈(element-wise multiplication)을 의미하고, $\oplus$는 원소별 덧셈(elementwise addition)을 나타냄
  - $\mathbf{x}^{(t)}$은 타임 스텝 t에서 입력 데이터이고, $\mathbf{h}^{(t-1)}$은 타임 스텝 t -1에서 은닉 유닛의 출력
  - 네 개의 상자는 시그모이드 함수(σ)나 하이퍼볼릭 탄젠트(tanh) 활성화 함수와 일련의 가중치로 표시
  - 이 상자는 입력($\mathbf{h}^{(t-1)}$과 $\mathbf{x}^{(t)}$)에 대해 행렬-벡터 곱셈을 수행한 후 선형 조합
  - 시그모이드 함수로 계산하는 유닛을 게이트라고 하며 $\odot$ 를 통해 출력

16.2 시퀀스 모델링을 위한 RNN

- **LSTM 셀**
  - LSTM 셀에는 **삭제 게이트**(forget gate), **입력 게이트**(input gate), **출력 게이트**(output gate)가 있음

    - 삭제 게이트($\mathbf{f}_t$)는 메모리 셀이 무한정 성장하지 않도록 셀 상태를 다시 설정
      사실 삭제 게이트가 통과할 정보와 억제할 정보를 결정
      $\mathbf{f}_t$는 다음과 같이 계산

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_{xf}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hf}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f\right)$$

      삭제 게이트는 원본 LSTM 셀에 포함되어 있지 않았음
      초기 모델을 향상시키기 위해 몇 년 후에 추가되었음

16.2 시퀀스 모델링을 위한 RNN

- **LSTM 셀**
  - 입력 게이트($\mathbf{i}_t$)와 후보 값(candidate value) ($\tilde{\boldsymbol{c}}_t$)은 셀 상태를 업데이트 하는 역할을 담당하며, 다음과 같이 계산

$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_{xi}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hi}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i\right)$$

$$\tilde{\boldsymbol{C}}_t = \tanh\left(\boldsymbol{W}_{xc}\,\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hc}\,\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_c\right)$$

타임 스텝 t 에서 셀 상태는 다음과 같이 계산

$$\boldsymbol{C}^{(t)} = \left(\boldsymbol{C}^{(t-1)} \odot \boldsymbol{f}_t\right) \oplus \left(\boldsymbol{i}_t \odot \tilde{\boldsymbol{C}}_t\right)$$

16.2 시퀀스 모델링을 위한 RNN

- **LSTM 셀**
  - 출력 게이트($\mathbf{o}_t$)는 은닉 유닛의 출력 값을 업데이트

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_{xo}\,\boldsymbol{x}^{(t)} + \boldsymbol{W}_{ho}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o\right)$$

이를 가지고 현재 타임 스텝에서 은닉 유닛의 출력을 다음과 같이 계산

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{C}^{(t)}\right)$$

16.2 시퀀스 모델링을 위한 RNN

- **LSTM 셀**
  - LSTM 셀의 구조와 연산이 매우 복잡하고 구현하기 어려워 보일 수 있음
  - 다행히도 텐서플로에 최적화된 래퍼 함수로 이미 모두 구현되어 있어 간단하고 효율적으로 LSTM 셀을 정의할 수 있음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **텐서플로로 시퀀스 모델링을 위한 RNN 구현**
  - RNN 이론을 소개했으므로 텐서플로를 사용하여 RNN을 구현하는 이 장의 실습 파트로 넘어가보겠음
  - 이 장 나머지에서 RNN을 다음 두 문제에 적용해 보겠음

1. 감성 분석
2. 언어 모델링

  - 앞으로 구현할 이 두 프로젝트는 모두 흥미롭지만 복잡하기도 함
  - 모든 코드를 한 번에 드러내지 않고 구현 코드를 몇 단계로 나누어 제시하고 자세히 설명
  - 설명을 읽기 전에 전체 그림을 조망하고 코드를 한 번에 보고 싶다면 https://github.com/rickiepark/pythonmachine-learning-book-3rd-edition/blob/master/code/ch16을 참고

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 감성 분석은 문장이나 텍스트 문서에 표현된 의견을 분석하는 것
  - 이 절과 이어지는 절에서 감성 분석을 위해 다대일(many-to-one) 구조의 다층 RNN을 구현해 보자
  - 다음 절에서는 언어 모델링 애플리케이션을 위한 다대다(many-to-many) RNN을 구현하겠음
  - RNN의 주요 개념을 소개하기 위해 의도적으로 간단한 예를 선택했지만 언어 모델링에는 챗봇(chatbot)처럼 흥미로운 애플리케이션이 굉장히 많음
  - 챗봇은 컴퓨터로 사람과 직접 대화하고 소통하는 애플리케이션

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

  영화 리뷰 데이터 준비

  - 8장의 전처리 단계에서 만든 정제된 데이터셋인 movie_data.csv 파일을 여기서 다시 사용
  - 먼저 필요한 모듈을 임포트하고 판다스의 DataFrame으로 데이터를 읽음

```
>>> import tensorflow as tf
>>> import tensorflow_datasets as tfds
>>> import numpy as np
>>> import pandas as pd
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 데이터프레임 df에는 'review'와 'sentiment' 두 개의 컬럼이 있음
  - 'review'는 영화 리뷰 텍스트(입력 특성)를 담고 있고 'sentiment'는 예측하려는 타깃 레이블 (0은 부정적인 리뷰, 1은 긍정적인 리뷰를 의미)
  - 영화 리뷰 텍스트는 단어의 시퀀스
  - RNN 모델을 만들어서 각 시퀀스를 긍정적(1) 또는 부정적(0)인 리뷰로 분류하겠음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - RNN 모델에 데이터를 주입하기 전에 몇 가지 전처리 단계를 적용해야 함

1. 텐서플로 데이터셋 객체를 만들고 훈련, 테스트, 검증 데이터셋으로 나눔
2. 훈련 데이터셋에 있는 고유한 단어를 찾음
3. 고유한 단어를 고유한 정수로 매핑하고 리뷰 텍스트를 정수(고유 단어의 인덱스) 배열로 인코딩
4. 모델에 입력하기 위해 데이터셋을 미니 배치로 나눔

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 첫 번째 단계를 진행해 보자
  - 데이터프레임에서 텐서플로 데이터셋을 만듦

```
>>> ## 1단계: 데이터셋 만들기
>>> target = df.pop('sentiment')
>>> ds_raw = tf.data.Dataset.from_tensor_slices(
...     (df.values, target.values))
>>> ## 확인:
>>> for ex in ds_raw.take(3):
...     tf.print(ex[0].numpy()[0][ :50], ex[1])
b'In 1974, the teenager Martha Moxley (Maggie Grace)' 1
b'OK... so... I really like Kris Kristofferson and h' 0
b'***SPOILER*** Do not read this, if you think about' 0
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 이제 훈련, 테스트, 검증 데이터셋으로 나눌 수 있음
  - 전체 데이터셋은 5만 개의 샘플을 담고 있음
  - 처음 2만 5,000개의 샘플은 평가를 위해 떼어 놓음(홀드아웃 테스트 데이터셋)
  - 그 다음 2만 개의 샘플은 훈련에 사용하고 5,000개의 샘플은 검증에 사용
  - 코드는 다음과 같음

```
>>> tf.random.set_seed(1)
>>> ds_raw = ds_raw.shuffle(
...     50000, reshuffle_each_iteration=False)
>>> ds_raw_test = ds_raw.take(25000)
>>> ds_raw_train_valid = ds_raw.skip(25000)
>>> ds_raw_train = ds_raw_train_valid.take(20000)
>>> ds_raw_valid = ds_raw_train_valid.skip(20000)
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 신경망의 입력으로 데이터를 준비하기 위해 단계 **2**~**3**에서 언급했던 것처럼 숫자 값으로 인코딩해야 함
  - 이렇게 하기 위해 먼저 훈련 데이터셋에서 고유한 단어(토큰)를 찾음
  - 데이터셋을 사용하여 고유한 토큰을 찾을 수 있지만 파이썬 표준 라이브러리에 있는 collections 패키지의 Counter 클래스를 사용하는 것이 더 효율적임

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 다음 코드에서 Counter 객체(token_counts)를 만들어 고유한 단어의 빈도를 수집
  - 이 애플리케이션에서는 (BoW 모델과 달리) 고유 단어 집합에만 관심이 있고 부수적으로 만들어진 단어 카운트는 필요하지 않음
  - 텍스트를 단어(또는 토큰)로 나누려면 tensorflow_datasets 패키지가 제공하는 Tokenizer 클래스를 사용할 수 있음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 고유 토큰을 수집하는 코드는 다음과 같음

```
>>> ## 2단계: 고유 토큰 (단어) 찾기
>>> from collections import Counter
>>> tokenizer = tfds.deprecated.text.Tokenizer()
>>> token_counts = Counter()
>>> for example in ds_raw_train:
...     tokens = tokenizer.tokenize(example[0].numpy()[0])
...     token_counts.update(tokens)
>>> print('어휘 사전 크기:', len(token_counts))
어휘 사전 크기: 87007
```

  - Counter 클래스에 대한 더 자세한 정보는 공식 문서(https://docs.python.org/3/library/collections.html#collections.Counter)를 참고

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 그다음 각각의 고유 단어를 고유 정수로 매핑
  - 파이썬 딕셔너리를 사용하여 수동으로 처리할 수 있음
  - 키는 고유 토큰(단어)이고 키에 매핑된 값은 고유한 정수
  - tensorflow_datasets 패키지는 이런 매핑과 전체 데이터셋을 인코딩할 수 있는 TokenTextEncoder 클래스를 제공
  - 먼저 고유한 토큰을 전달하여 TokenTextEncoder 클래스로 encoder 객체를 만듦(token_counts는 토큰과 횟수를 포함하고 있지만 여기서는 횟수는 필요하지 않으므로 무시함)
  - encoder.encode() 메서드를 호출하여 입력 텍스트를 정수 리스트로 변환

```
>>> ## 3단계: 고유 토큰을 정수로 인코딩하기
>>> encoder = tfds.deprecated.text.TokenTextEncoder(token_counts)
>>> example_str = 'This is an example!'
>>> print(encoder.encode(example_str))
[232, 9, 270, 1123]
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 검증 데이터와 테스트 데이터에 있는 토큰이 훈련 데이터에 없다면 매핑되지 않을 수 있음
  - q개(TokenTextEncoder에 전달한 token_counts의 크기, 여기서는 8만 7,007개)의 토큰이 있고 이전에 본 적이 없으며 token_counts에 포함되지 않은 모든 토큰은 정수 q+1에 할당(이 경우 8만 7,008개)
  - 다른 말로 하면 인덱스 q+1이 알려지지 않은 단어를 위해 예약
  - 예약된 또 다른 값은 정수 0
  - 시퀀스 길이를 조절하기 위한 용도로 사용

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 데이터셋 객체의 map() 메서드를 사용하여 다른 변환을 적용하듯이 데이터셋에 있는 각 텍스트를 변환할 수 있음
  - 여기에는 작은 문제가 있음
  - 텍스트 데이터가 텐서 객체에 들어가 있어 즉시 실행 모드에서 텐서의 numpy() 메서드로 참조할 수 있음
  - map() 메서드로 변환하는 동안에는 즉시 실행이 비활성화됨
  - 이 문제를 해결하기 위해 두 개의 함수를 정의
  - 첫 번째 함수는 즉시 실행 모드가 활성화된 것처럼 입력 텐서를 다룸

```
>>> ## 3-A단계: 변환을 위한 함수 정의
>>> def encode(text_tensor, label):
...     text = text_tensor.numpy()[0]
...     encoded_text = encoder.encode(text)
...     return encoded_text, label
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 두 번째 함수에서 첫 번째 함수를 tf.py_function으로 감싸서 map() 메서드에서 사용할 수 있는 텐서플로 연산으로 변환
  - 텍스트를 정수 리스트로 인코딩하는 과정은 다음 코드를 통해 수행됨

```
>>> ## 3-B단계: 함수를 TF 연산으로 변환하기
>>> def encode_map_fn(text, label):
...     return tf.py_function(encode, inp=[text, label],
...                           Tout=(tf.int64, tf.int64))
>>> ds_train = ds_raw_train.map(encode_map_fn)
>>> ds_valid = ds_raw_valid.map(encode_map_fn)
>>> ds_test = ds_raw_test.map(encode_map_fn)
>>> # 샘플의 크기 확인하기:
>>> tf.random.set_seed(1)
>>> for example in ds_train.shuffle(1000).take(5):
...     print('시퀀스 길이:', example[0].shape)
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
시퀀스 길이: (24,)
시퀀스 길이: (179,)
시퀀스 길이: (262,)
시퀀스 길이: (535,)
시퀀스 길이: (130,)
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 지금까지 단어 시퀀스를 정수 시퀀스로 변환
  - 여전히 해결할 문제가 하나 있음
  - 시퀀스 길이가 다름(앞의 코드에서 랜덤하게 선택한 다섯 개의 샘플 길이에서 알 수 있음)
  - 일반적으로 RNN은 다른 길이의 시퀀스를 다룰 수 있지만 미니 배치에 있는 시퀀스는 효율적으로 텐서에 저장하기 위해 동일한 길이가 되어야 함

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 크기가 다른 원소를 가진 데이터셋을 미니 배치로 나누기 위해 텐서플로는 (batch() 대신) padded_batch() 메서드를 제공
  - 이 메서드는 하나의 배치에 포함되는 모든 원소를 자동으로 0으로 패딩하여 배치에 있는 모든 시퀀스가 동일한 길이가 되도록 만듦

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 실제 예시로 확인해 보자
  - 훈련 데이터셋에서 크기가 8인 작은 데이터셋 ds_train을 만듦
  - batch_size=4로 padded_batch() 메서드를 적용해 보자
  - 미니 배치에 들어가기 전에 개별 원소의 길이와 만들어진 미니 배치의 차원을 출력

```
>>> ## 3-B단계: 함수를 TF 연산으로 변환하기
>>> def encode_map_fn(text, label):
...     return tf.py_function(encode, inp=[text, label],
...                           Tout=(tf.int64, tf.int64))
>>> ds_train = ds_raw_train.map(encode_map_fn)
>>> ds_valid = ds_raw_valid.map(encode_map_fn)
>>> ds_test = ds_raw_test.map(encode_map_fn)
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
>>> # 샘플의 크기 확인하기:
>>> tf.random.set_seed(1)
>>> for example in ds_train.shuffle(1000).take(5):
...     print('시퀀스 길이:', example[0].shape)
시퀀스 길이: (24,)
시퀀스 길이: (179,)
시퀀스 길이: (262,)
시퀀스 길이: (535,)
시퀀스 길이: (130,)
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 출력된 텐서 크기에서 알 수 있듯이 첫 번째 배치의 열 개수(즉, .shape[1])는 688
  - 처음 네개의 샘플이 하나의 배치가 되었고 이 샘플 중에 가장 큰 크기를 사용
  - 다시 말하면 이 배치에 있는 세 개의 다른 샘플을 이 크기에 맞도록 필요한 만큼 패딩을 추가
  - 비슷하게 두번째 배치의 열 크기는 다음 네 개의 샘플 중에 가장 큰 크기인 453
  - 여기서도 최대 길이보다 작은 다른 샘플에 패딩을 추가

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 세 개의 데이터셋을 모두 배치 크기 32의 미니 배치로 나누겠음

```
>>> train_data = ds_train.padded_batch(
...     32, padded_shapes=([-1],[]))
>>> valid_data = ds_valid.padded_batch(
...     32, padded_shapes=([-1],[]))
>>> test_data = ds_test.padded_batch(
...     32, padded_shapes=([-1],[]))
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 이제 데이터가 이어지는 절에서 구현할 RNN 모델에 적합한 포맷이 되었음
  - 그전에 먼저 다음 절에서 특성 **임베딩**(embedding)에 대해 다루어 보겠음
  - 필수적인 것은 아니지만 단어 벡터의 차원을 줄여 주기 때문에 매우 권장되는 전처리 단계

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

  문장 인코딩을 위한 임베딩 층

  - 이전의 데이터 준비 단계에서 동일한 길이의 시퀀스를 생성
  - 이 시퀀스의 원소는 고유한 단어의 인덱스에 해당하는 정수
  - 이런 단어 인덱스를 입력 특성으로 변환하는 몇 가지 방법이 있음
  - 간단하게 원-핫 인코딩을 적용하여 인덱스를 0 또는 1로 이루어진 벡터로 변환
  - 각 단어는 전체 데이터셋의 고유한 단어의 수에 해당하는 크기를 가진 벡터로 변환
  - 고유한 단어의 수(어휘 사전의 크기)가 10^4-10^5 단위가 될 수 있으며 입력 특성의 개수도 마찬가지임
  - 이렇게 많은 특성에서 훈련된 모델은 **차원의 저주**(curse of dimensionality)로 인한 영향을 받음
  - 또 하나를 제외하고 모든 원소가 0이므로 특성 벡터가 매우 희소해짐

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

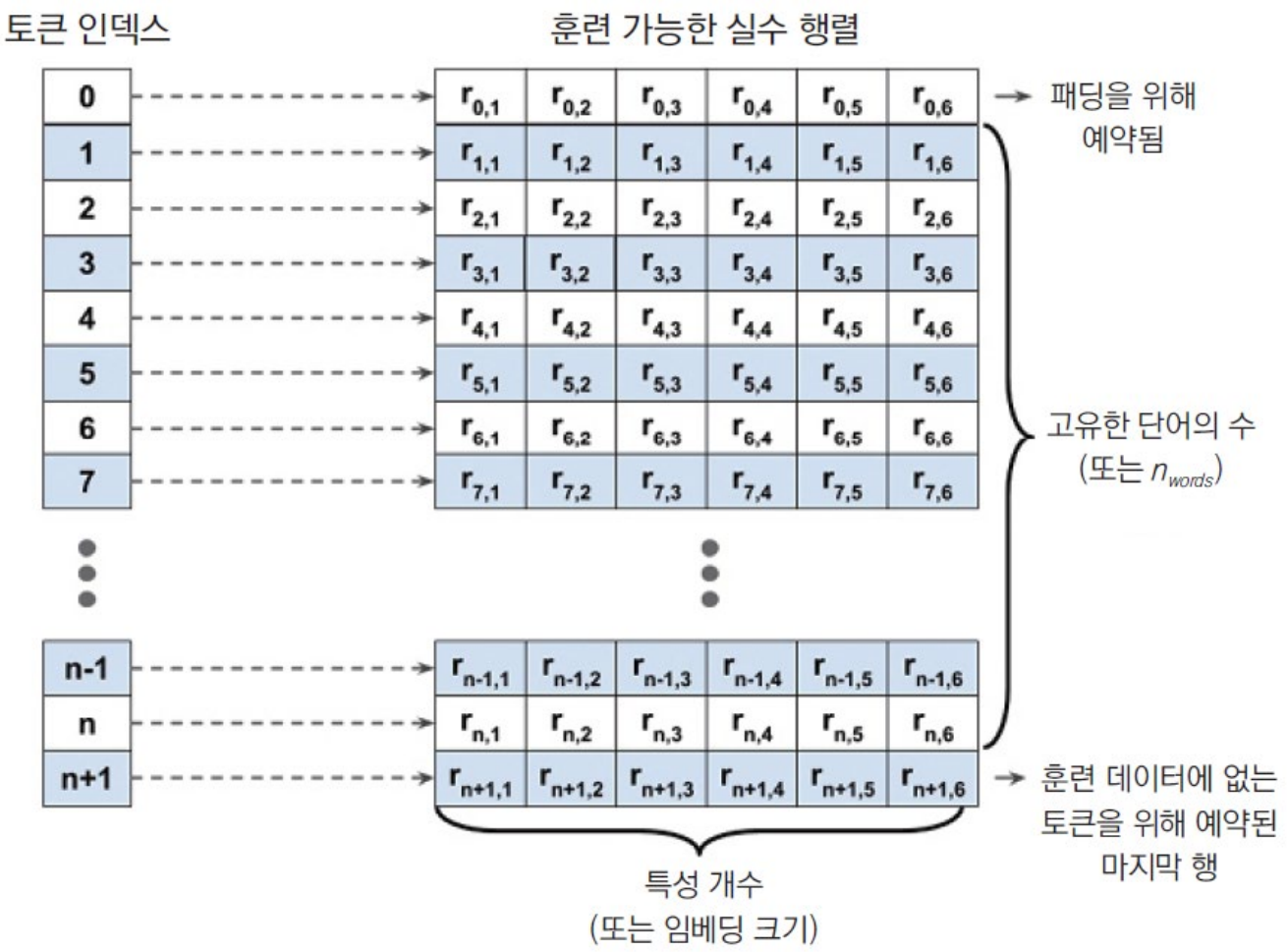
- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 좀 더 고급스러운 방법은 각 단어를 (정수가 아닌) 실수 값을 가진 고정된 길이의 벡터로 변환하는 것
  - 원-핫 인코딩과 달리 고정된 길이의 벡터를 사용하여 무한히 많은 실수를 표현할 수 있음(이론적으로 [-1, 1] 사이에서 무한한 실수를 뽑을 수 있음)
  - **임베딩**(embedding)이라고 하는 특성 학습 기법을 사용하여 데이터셋에 있는 단어를 표현하는 데 중요한 특성을 자동으로 학습할 수 있음
  - 고유한 단어의 수를 n_{words}라고 하면 고유한 단어의 수보다 훨씬 작게(embedding_dim << n_{words}) 임베딩 벡터(또는 임베딩 차원) 크기를 선택하여 전체 어휘를 입력 특성으로 나타냄

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 원-핫 인코딩에 비해 임베딩의 장점은 다음과 같음
    - 특성 공간의 차원이 축소되므로 차원의 저주로 인한 영향을 감소시킴
    - 신경망에서 임베딩 층이 최적화(학습)되기 때문에 중요한 특성이 추출됨

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

▼ **그림 16-10 토큰 인덱스와 임베딩 행렬 매핑**



16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - n + 2 크기(토큰 개수 n에 패딩을 위해 예약된 인덱스 0과 토큰 집합에 없는 단어를 위해 예약된 인덱스 n + 1이 추가)의 토큰 집합이 주어지면 (n + 2)×embedding_dim 크기의 임베딩 행렬이 만들어짐
  - 이 행렬의 행은 토큰에 연관된 수치 특성을 표현
  - 정수 i가 입력으로 임베딩 층에 주어지면 인덱스 i에 해당하는 행렬의 행을 찾아 이 수치 특성을 반환
  - 임베딩 행렬은 신경망 모델의 입력층의 역할을 하게 됨
  - 실제로는 tf.keras.layers.Embedding을 사용하여 임베딩 층을 간단히 만들 수 있음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 다음 예제에서 모델을 만들고 임베딩 층을 추가해 보자

```
>>> from tensorflow.keras.layers import Embedding
>>> model = tf.keras.Sequential()
>>> model.add(Embedding(input_dim=100,
...                     output_dim=6,
...                     input_length=20,
...                     name='embed-layer'))
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embed-layer (Embedding)      (None, 20, 6)             600
=================================================================
Total params: 6,00
Trainable params: 6,00
Non-trainable params: 0
_________________________________________________________________
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 이 모델의 입력(임베딩 층)은 batchsize×input_length 차원을 가진 랭크 2여야 함
  - 여기서 input_length는 시퀀스 길이(이 예에서는 input_length 매개변수에 20을 지정했음)
  - 예를 들어 미니 배치에 <14, 43, 52, 61, 8, 19, 67, 83, 10, 7, 42, 87, 56, 18, 94, 17, 67, 90, 6,39>와 같은 입력 시퀀스가 있을 수 있음
  - 이 시퀀스의 각 원소는 고유한 단어의 인덱스
  - 출력 차원은 batchsize×input_length×embedding_dim
  - embedding_dim은 임베딩 특성의 크기(여기서는 output_dim 매개변수에 6을 지정했음)
  - 임베딩 층에 지정한 또 다른 매개변수 input_dim은 모델이 입력으로 받을 고유한 정수 값에 해당(예를 들어 n+2, 여기서는 100으로 지정함)
  - 이 예에서 임베딩 행렬의 크기는 100×6이 됨

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

  RNN 모델 만들기

  - 이제 RNN 모델을 만들 준비가 되었음
  - 케라스 Sequential 클래스를 사용하여 임베딩 층, RNN 층, 완전 연결 층을 연결
  - 순환 층에는 다음과 같은 클래스를 사용할 수 있음

    - **SimpleRNN:** 완전 연결 순환 층인 기본 RNN
    - **LSTM:** 긴 의존성을 감지할 수 있는 LSTM RNN
    - **GRU:** LSTM의 대안인 GRU 유닛을 사용한 순환 층

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 이런 순환 층 중 하나를 사용해서 다층 RNN 모델을 만드는 방법을 알아보자
  - 다음 예제에서 input_dim=1000과 output_dim=32로 지정한 임베딩 층으로 RNN 모델을 시작
  - 그다음 두개의 SimpleRNN 순환 층을 추가
  - 마지막으로 완전 연결 층을 출력층으로 추가
  - 이 층은 예측으로 하나의 출력을 반환할 것

```
>>> from tensorflow.keras import Sequential
>>> from tensorflow.keras.layers import Embedding
>>> from tensorflow.keras.layers import SimpleRNN
>>> from tensorflow.keras.layers import Dense
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
>>> model = Sequential()
>>> model.add(Embedding(input_dim=1000, output_dim=32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32))
>>> model.add(Dense(1))
>>> model.summary()
Model: "sequential"
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 32)          32000
_________________________________________________________________
simple_rnn (SimpleRNN)       (None, None, 32)          2080
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                2080
_________________________________________________________________
dense (Dense)                (None, 1)                 33
=================================================================
Total params: 36,193
Trainable params: 36,193
Non-trainable params: 0
_________________________________________________________________
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

  감성 분석 작업을 위한 RNN 모델 만들기

  - 시퀀스 길이가 길기 때문에 길게 미치는 영향을 감지하기 위해 LSTM 층을 사용
  - 또한, Bidirectional 래퍼 클래스로 LSTM 층을 감싸겠음
  - 이 층은 입력 시퀀스를 처음부터 끝까지 그리고 끝에서 처음까지 양방향으로 순환 층을 통과하도록 함

```
>>> embedding_dim = 20
>>> vocab_size = len(token_counts) + 2
>>> tf.random.set_seed(1)
>>> ## 모델 만들기
>>> bi_lstm_model = tf.keras.Sequential([
...     tf.keras.layers.Embedding(
...         input_dim=vocab_size,
...         output_dim=embedding_dim,
...         name='embed-layer'),
...
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
...     tf.keras.layers.Bidirectional(
...         tf.keras.layers.LSTM(64, name='lstm-layer'),
...         name='bidir-lstm'),
...
...     tf.keras.layers.Dense(64, activation='relu'),
...
...     tf.keras.layers.Dense(1, activation='sigmoid')
>>> ])
>>> bi_lstm_model.summary()
>>> ## 컴파일과 훈련
>>> bi_lstm_model.compile(
...     optimizer=tf.keras.optimizers.Adam(1e-3),
...     loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
...     metrics=['accuracy'])
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
>>> history = bi_lstm_model.fit(
...     train_data,
...     validation_data=valid_data,
...     epochs=10)
>>> ## 테스트 데이터에서 평가
>>> test_results = bi_lstm_model.evaluate(test_data)
>>> print('테스트 정확도: {:.2f}%'.format(test_results[1]*100))
Epoch 1/10
625/625 [==============================] - 96s 154ms/step - loss: 0.4410 - accuracy:
0.7782 - val_loss: 0.0000e+00 - val_accuracy: 0.0000e+00
Epoch 2/10
625/625 [==============================] - 95s 152ms/step - loss: 0.1799 - accuracy:
0.9326 - val_loss: 0.4833 - val_accuracy: 0.8414
...(생략)...
테스트 정확도: 85.15%
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 열 번의 에포크 동안 모델을 훈련한 후 테스트 데이터에서 평가하여 85% 정확도를 얻었음
  - (이 결과가 최신 모델과 비교했을 때 IMDb 데이터셋에서 얻을 수 있는 최댓값은 아니고 여기서는 RNN 모델을 만드는 방법을 알아보는 데 초점을 두겠음)

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - SimpleRNN과 같은 다른 종류의 순환 층을 사용할 수도 있음
  - 확인해 보면 알 수 있지만 일반적인 순환 층으로 만든 모델은 (훈련 데이터에서도) 좋은 예측 성능을 달성하지 못함
  - 예를 들어 이전 코드에서 양방향 LSTM 층을 단방향 SimpleRNN 층으로 바꾸고 전체 길이를 사용한 시퀀스로 모델을 훈련하면 훈련하는 동안 손실이 전혀 감소하지 않음
  - 이 데이터셋에 있는 시퀀스가 너무 길기 때문임
  - SimpleRNN 층을 사용한 모델은 장기간 의존성을 학습할 수 없고 그레이디언트 감소나 폭주로 인한 영향을 받음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - SimpleRNN으로 이 데이터셋에서 납득할 수 있는 수준의 예측 성능을 얻으려면 시퀀스 길이를 줄여야 함
  - 경험상 쉽게 알 수 있는 '도메인 지식'을 사용하면 영화 리뷰의 마지막 문장에 감성에 관한 정보가 많이 담겨 있다고 가정할 수 있음
  - 각 리뷰의 마지막 부분에만 초점을 맞추어 보자
  - preprocess_datasets()라는 헬퍼 함수를 만들어 전처리 단계 **2**~**4**를 연결
  - 이 함수의 매개변수는 각 리뷰에서 얼마나 많은 토큰을 사용할지 결정하는 max_seq_length
  - 예를 들어 max_seq_length=100으로 지정하면 100개 이상의 토큰을 가진 리뷰에서 마지막 100개의 토큰만 사용
  - max_seq_length=None으로 지정하면 이전처럼 전체 길이의 시퀀스가 사용
  - max_seq_length의 값을 바꾸어 보면 긴 시퀀스를 다룰 수 있는 RNN 모델의 능력을 비교해 볼 수 있음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - preprocess_datasets() 함수의 코드는 다음과 같음

```
>>> from collections import Counter
>>> def preprocess_datasets(
...     ds_raw_train,
...     ds_raw_valid,
...     ds_raw_test,
...     max_seq_length=None,
...     batch_size=32):
...
...     ## 1단계: (데이터셋 만들기 이미 완료)
...     ## 2단계: 고유 토큰 찾기
...     tokenizer = tfds.deprecated.text.Tokenizer()
...     token_counts = Counter()
...
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
...     for example in ds_raw_train:
...         tokens = tokenizer.tokenize(example[0].numpy()[0])
...         if max_seq_length is not None:
...             tokens = tokens[-max_seq_length:]
...         token_counts.update(tokens)
...
...     print('어휘 사전 크기:', len(token_counts))
...
...     ## 3단계: 텍스트 인코딩하기
...     encoder = tfds.deprecated.text.TokenTextEncoder(
...                   token_counts)
...     def encode(text_tensor, label):
...         text = text_tensor.numpy()[0]
...         encoded_text = encoder.encode(text)
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
...         if max_seq_length is not None:
...             encoded_text = encoded_text[-max_seq_length:]
...         return encoded_text, label
...
...     def encode_map_fn(text, label):
...         return tf.py_function(encode, inp=[text, label],
...                               Tout=(tf.int64, tf.int64))
...
...     ds_train = ds_raw_train.map(encode_map_fn)
...     ds_valid = ds_raw_valid.map(encode_map_fn)
...     ds_test = ds_raw_test.map(encode_map_fn)
...
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
...     ## 4단계: 배치 데이터 만들기
...     train_data = ds_train.padded_batch(
...         batch_size, padded_shapes=([-1],[]))
...
...     valid_data = ds_valid.padded_batch(
...         batch_size, padded_shapes=([-1],[]))
...
...     test_data = ds_test.padded_batch(
...         batch_size, padded_shapes=([-1],[]))
...
...     return (train_data, valid_data,
...             test_data, len(token_counts))
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 그다음 여러 가지 구조의 모델을 더 쉽게 만들기 위해 build_rnn_model()이라는 헬퍼 함수를 정의

```
>>> from tensorflow.keras.layers import Embedding
>>> from tensorflow.keras.layers import Bidirectional
>>> from tensorflow.keras.layers import SimpleRNN
>>> from tensorflow.keras.layers import LSTM
>>> from tensorflow.keras.layers import GRU
>>> def build_rnn_model(embedding_dim, vocab_size,
...                     recurrent_type='SimpleRNN',
...                     n_recurrent_units=64,
...                     n_recurrent_layers=1,
...                     bidirectional=True):
...
...     tf.random.set_seed(1)
...
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
...     # 모델 생성
...     model = tf.keras.Sequential()
...
...     model.add(
...         Embedding(
...             input_dim=vocab_size,
...             output_dim=embedding_dim,
...             name='embed-layer')
...     )
...
...     for i in range(n_recurrent_layers):
...         return_sequences = (i < n_recurrent_layers-1)
...
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
...         if recurrent_type == 'SimpleRNN':
...             recurrent_layer = SimpleRNN(
...                 units=n_recurrent_units,
...                 return_sequences=return_sequences,
...                 name='simprnn-layer-{}'.format(i))
...         elif recurrent_type == 'LSTM':
...             recurrent_layer = LSTM(
...                 units=n_recurrent_units,
...                 return_sequences=return_sequences,
...                 name='lstm-layer-{}'.format(i))
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
...         elif recurrent_type == 'GRU':
...             recurrent_layer = GRU(
...                 units=n_recurrent_units,
...                 return_sequences=return_sequences,
...                 name='gru-layer-{}'.format(i))
...
...         if bidirectional:
...             recurrent_layer = Bidirectional(
...                 recurrent_layer, name='bidir-' +
...                 recurrent_layer.name)
...
...         model.add(recurrent_layer)
...
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
...     model.add(tf.keras.layers.Dense(64, activation='relu'))
...     model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
...
...     return model
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 이제 범용적이고 편리한 두 헬퍼 함수를 사용하여 다양한 길이의 입력 시퀀스로 여러 RNN 모델을 비교할 수 있음
  - 예를 들어 다음 코드에서 SimpleRNN 층 하나를 가진 모델을 최대 길이가 토큰 100개인 시퀀스로 훈련해 보자

```
>>> batch_size = 32
>>> embedding_dim = 20
>>> max_seq_length = 100
>>> train_data, valid_data, test_data, n = preprocess_datasets(
...     ds_raw_train, ds_raw_valid, ds_raw_test,
...     max_seq_length=max_seq_length,
...     batch_size=batch_size
... )
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
>>> vocab_size = n + 2
>>> rnn_model = build_rnn_model(
...     embedding_dim, vocab_size,
...     recurrent_type='SimpleRNN',
...     n_recurrent_units=64,
...     n_recurrent_layers=1,
...     bidirectional=True)
>>> rnn_model.summary()
Model: "sequential"
어휘 사전 크기: 58063
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embed-layer (Embedding)      (None, None, 20)          1161300
_________________________________________________________________
bidir-simprnn-layer-0        (Bidir (None, 128)        10880
_________________________________________________________________
Dense    (Dense)             (None, 64)                8256
_________________________________________________________________
dense_1  (Dense)             (None, 1)                 65
=================================================================
Total params: 1,180,501
Trainable params: 1,180,501
Non-trainable params: 0
_________________________________________________________________
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
>>> rnn_model.compile(
...     optimizer=tf.keras.optimizers.Adam(1e-3),
...     loss=tf.keras.losses.BinaryCrossentropy(
...         from_logits=False), metrics=['accuracy'])
>>> history = rnn_model.fit(
...     train_data,
...     validation_data=valid_data,
...     epochs=10)
Epoch 1/10
625/625 [==============================] - 73s 118ms/step - loss: 0.6996 - accuracy:
0.5074 - val_loss: 0.6880 - val_accuracy: 0.5476
Epoch 2/10
...(생략)...
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**

```
>>> results = rnn_model.evaluate(test_data)
>>> print('테스트 정확도: {:.2f}%'.format(results[1]*100))
테스트 정확도: 80.70%
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **첫 번째 프로젝트: IMDb 영화 리뷰의 감성 분석**
  - 시퀀스를 100개의 토큰까지로 자르고 양방향 SimpleRNN 층을 사용하면 80% 정도의 분류 정확도를 얻음
  - 이전의 양방향 LSTM 모델(이 데이터셋에서 85.15% 정확도를 얻음)에 비해 정확도가 조금 낮지만 줄어든 시퀀스에서 성능이 SimpleRNN으로 전체 영화 리뷰에서 달성한 것보다 훨씬 나음
  - 연습 문제로 앞서 정의한 두 헬퍼 함수로 이를 확인해 보자
  - build_rnn_model() 헬퍼 함수에 max_seq_length=None, bidirectional=False로 지정해서 테스트해 보자(깃허브의 주피터 노트북에 이에 대한 답이 있음)

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

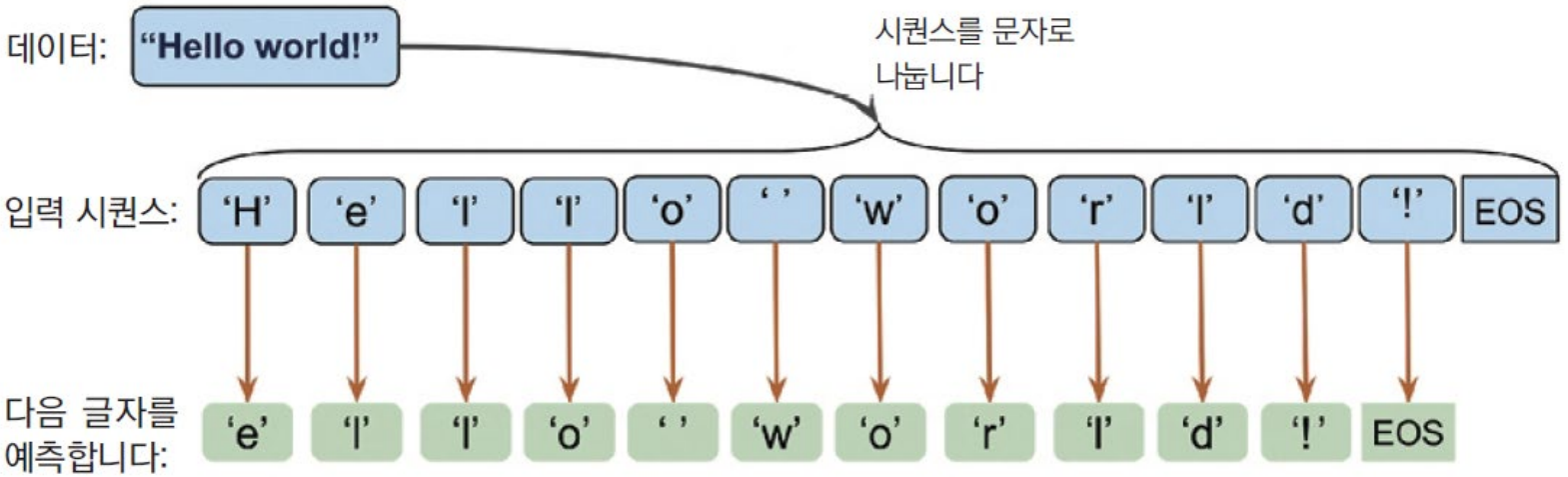
- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 언어 모델링(language modeling)은 영어 문장 생성처럼 기계가 사람의 언어와 관련된 작업을 수행하도록 만드는 흥미로운 애플리케이션
  - 이 분야에서 관심을 끄는 결과물 중 하나는 서스키버(Sutskever), 마틴(Martens), 힌튼(Hinton)의 작업
  - 앞으로 만들 모델의 입력은 텍스트 문서
  - 입력 문서와 비슷한 스타일로 새로운 텍스트를 생성하는 모델을 만드는 것이 목표
  - 입력 데이터는 책이나 특정 프로그래밍 언어로 만든 컴퓨터 프로그램일 수 있음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 글자 단위 언어 모델링에서 입력은 글자의 시퀀스로 나뉘어 한 번에 글자 하나씩 네트워크에 주입
  - 이 네트워크는 지금까지 본 글자와 함께 새로운 글자를 처리하여 다음 글자를 예측
  - 그림 16-11은 글자 단위 언어 모델링의 예를 보여 줌(EOS는 시퀀스의 끝(end of sequence)을 의미)

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

▼ 그림 16-11 글자 단위 언어 모델링



16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 이 구현을 데이터 전처리, RNN 모델 구성, 다음 글자를 예측하고 새로운 텍스트를 생성하는 세개의 단계로 나누겠음

데이터셋 전처리

- 이 절에서 글자 수준의 언어 모델링을 위한 데이터를 준비
- 수천 권의 무료 전자책을 제공하는 구텐베르크(Gutenberg) 프로젝트 웹 사이트(https://www.gutenberg.org/)에서 입력 데이터를 구하겠음
- 이 예제에서는 쥘 베른(Jules Verne)이 1874년 출간한 <신비의 섬(The Mysterious Island)> 책의 텍스트를 사용(http://www.gutenberg.org/files/1268/1268-0.txt)

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 앞의 링크를 통해 직접 내려받거나 macOS나 리눅스 시스템을 사용한다면 터미널에서 다음 명령으로 이 파일을 내려받을 수 있음

```
> curl -O http://www.gutenberg.org/files/1268/1268-0.txt
```

  - 이 링크에 접근할 수 없다면 책 코드 저장소의 16장 폴더에 있는 복사본을 사용(https://github.com/rickiepark/python-machine-learning-book-3rd-edition/tree/master/ch16)

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 이 데이터를 내려받으면 보통의 텍스트로 파이썬에서 읽을 수 있음
  - 다음 코드에서 다운로드 파일을 직접 읽어 시작과 끝부분을 삭제(구텐베르크 프로젝트에 대한 설명 부분)
  - 그 다음 파이썬 변수 char_set을 만들어 이 텍스트에 있는 고유한 단어 집합을 저장

```
>>> import numpy as np
>>> ## 텍스트 읽고 전처리하기
>>> with open('1268-0.txt', 'r', encoding='UTF8') as fp:
...     text=fp.read()
>>> start_indx = text.find('THE MYSTERIOUS ISLAND')
>>> end_indx = text.find('End of the Project Gutenberg')
>>> text = text[start_indx:end_indx]
>>> char_set = set(text)
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**

```
>>> print('전체 길이:', len(text))
전체 길이: 1112350
>>> print('고유한 문자:' len(char_set))
고유한 문자: 80
```

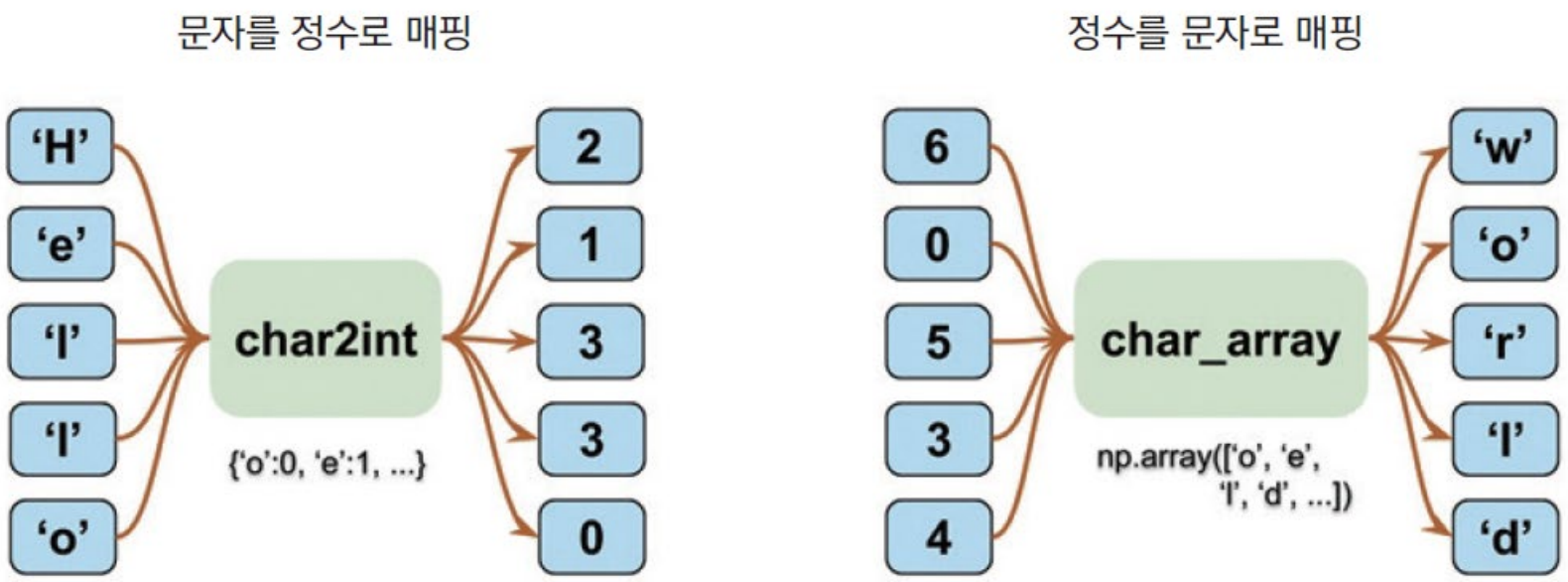
16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 텍스트를 내려받고 전처리하여 총 111만 2,350개의 문자와 80개의 고유한 문자로 구성된 시퀀스를 얻었음
  - 대부분의 신경망 라이브러리와 RNN 구현은 문자열 형태의 입력 데이터를 다룰 수 없음
  - 이 때문에 텍스트 데이터를 숫자 형태로 바꾸어야 함
  - 이를 위해 파이썬 딕셔너리 char2int를 만들어 각 문자를 정수로 매핑하겠음
  - 또한, 모델의 출력 결과를 텍스트로 변환하는 역 매핑도 필요함
  - 정수와 문자를 키와 값으로 연결한 딕셔너리로 역 매핑을 수행할 수도 있지만 인덱스와 고유 문자를 매핑한 넘파이 배열을 사용하는 것이 훨씬 효율적임

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 그림 16-12는 "Hello"와 "world"를 사용해서 문자를 정수로 변환하고 그 반대로 변환하는 예를 보여 줌

▼ **그림 16-12 문자와 인덱스 매핑**



16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 다음 코드는 문자를 정수로 매핑하는 딕셔너리를 만드는 것과 넘파이 배열의 인덱싱을 사용하여 반대로 매핑하는 예를 보여 줌

```
>>> chars_sorted = sorted(char_set)
>>> char2int = {ch:i for i,ch in enumerate(chars_sorted)}
>>> char_array = np.array(chars_sorted)

>>> text_encoded = np.array(
...     [char2int[ch] for ch in text],
...     dtype=np.int32)
>>> print('인코딩된 텍스트 크기: ', text_encoded.shape)
인코딩된 텍스트 크기: (1112350,)
>>> print(text[:15], '     == 인코딩 ==> ', text_encoded[:15])
>>> print(text_encoded[15:21], ' == 디코딩 ==> ',
...     ''.join(char_array[text_encoded[15:21]]))
THE MYSTERIOUS       == 인코딩 ==>  [44 32 29  1 37 48 43 44 29 42 33 39 45 43  1]
[33 43 36 25 38 28]  == 디코딩 ==>  ISLAND
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 넘파이 배열 text_encoded는 텍스트에 있는 모든 문자에 대한 인코딩 값을 담고 있음
  - 이 배열을 사용하여 텐서플로 데이터셋을 만들겠음

```
>>> import tensorflow as tf
>>> ds_text_encoded = tf.data.Dataset.from_tensor_slices(
...                     text_encoded)
>>> for ex in ds_text_encoded.take(5):
...     print('{} -> {}'.format(ex.numpy(), char_array[ex.numpy()]))
44 -> T
32 -> H
29 -> E
1 ->
37 -> M
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 지금까지 텍스트에 나타난 순서대로 문자를 담기 위해 반복 가능한 Dataset 객체를 만들었음
  - 이제 한걸음 물러서서 앞으로 하려는 일에 대해 큰 그림을 그려 보자
  - 텍스트 생성 작업의 경우 이를 분류 작업으로 표현할 수 있음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

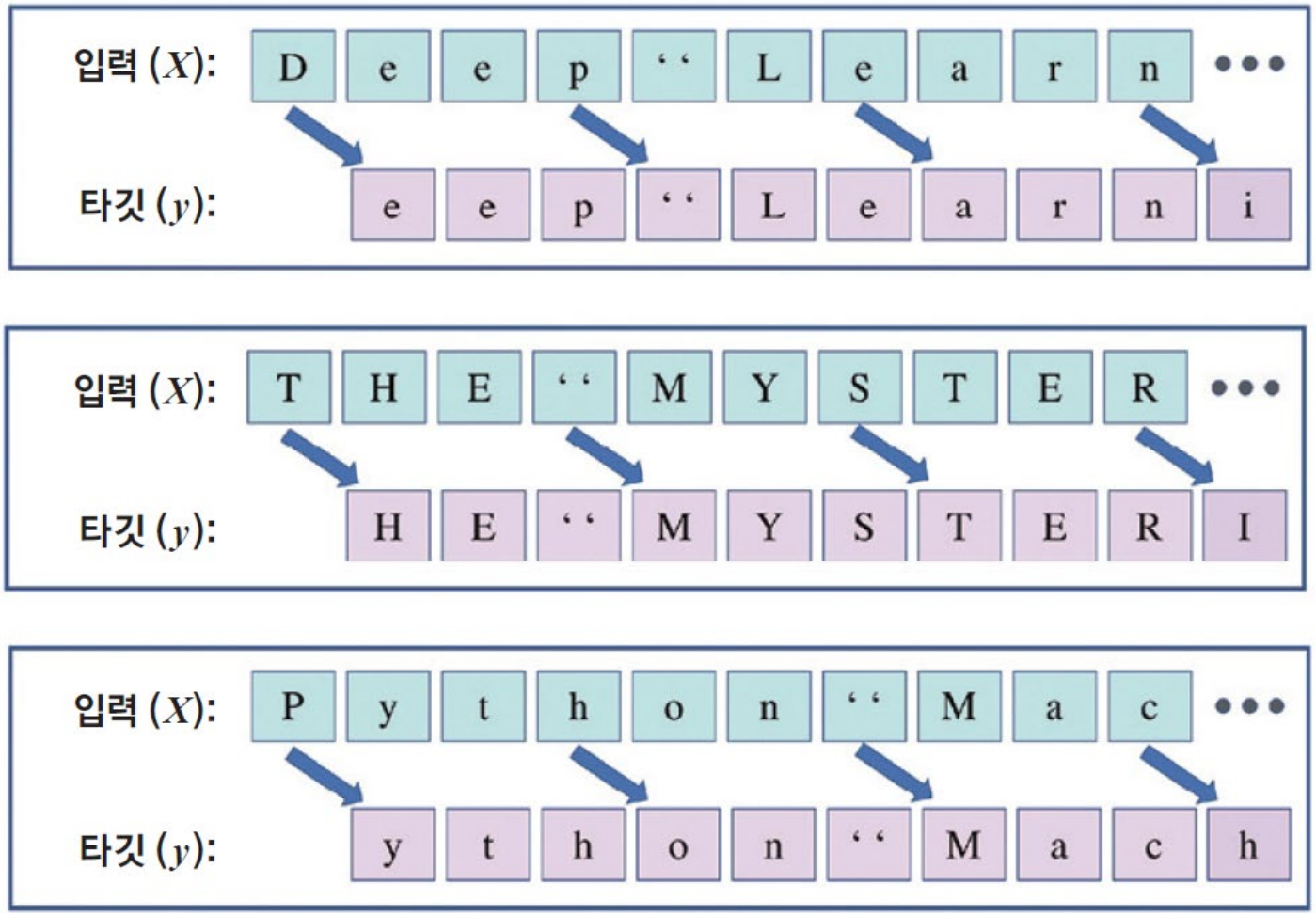
▼ 그림 16-13 불완전한 문자 시퀀스 집합 예

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 그림 16-13에서 왼쪽 박스에 있는 시퀀스를 입력으로 생각할 수 있음
  - 새로운 텍스트를 생성하기 위해 입력 시퀀스가 주어졌을 때 다음 문자를 예측하는 모델을 만드는 것이 목표
  - 이 입력 시퀀스는 불완전한 텍스트
  - 예를 들어 "Deep Learn"을 주입한 후 모델은 다음 문자로 "i"를 예측해야 함
  - 80개의 고유한 문자가 있으므로 이 문제는 다중 분류 작업이 됨
  - 그림 16-14에서 다중 분류 방식을 기반으로 길이 1인 시퀀스(즉, 하나의 글자)로 시작해서 새로운 텍스트를 반복하여 생성할 수 있음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

▼ 그림 16-14 새로운 텍스트 생성 예



16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 텐서플로로 텍스트 생성 모델을 구현하기 위해 먼저 시퀀스 길이를 40으로 자르겠음
  - 즉, 입력 텐서 x가 40개의 토큰으로 구성된다는 의미
  - 실제로 시퀀스 길이는 생성된 텍스트의 품질에 영향을 미침
  - 긴 시퀀스가 더 의미 있는 문장을 만들 수 있음
  - 짧은 시퀀스일 경우 모델이 대부분 문맥을 무시하고 개별 단어를 정확히 감지하는 데 초점을 맞출 수 있음
  - 긴 시퀀스가 보통 더 의미 있는 문장을 만들지만 이전에 언급한 것처럼 긴 시퀀스에서 RNN 모델이 장기간 의존성을 감지하기 어려움
  - 실제로 적절한 시퀀스 길이를 찾는 것은 경험적으로 평가해야 하는 하이퍼파라미터 최적화 문제
  - 여기서는 적절한 균형을 유지하기 위해 40을 선택

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 그림 16-14에서 볼 수 있듯이 입력 **x**와 타깃 **y**는 한 글자씩 어긋나 있음
  - 텍스트를 41 문자씩 나누겠음
  - 처음부터 40개의 문자는 입력 시퀀스 **x**가 되고 마지막 40개 문자는 타깃 시퀀스 **y**가 됨

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - Dataset 객체 ds_text_encoded에 인코딩된 전체 텍스트를 원본 문자 순서대로 저장해 놓았음
  - 이 장('영화 리뷰 데이터 준비' 절)에서 이미 보았던 데이터셋 변환 기법을 사용하여 이전 그림에 있는 입력 **x**와 타깃 **y**를 만들 방법은 매우 간단함
  - 먼저 batch() 메서드를 사용해서 41개의 문자로 구성된 텍스트 조각을 만듦
  - 즉, batch_size=41로 지정함
  - 마지막 배치 길이가 41보다 작으면 이 배치는 버림
  - 만들어진 ds_chunks 데이터셋은 항상 길이가 41인 시퀀스를 담고 있음
  - 이 41개의 문자 조각을 사용해서 시퀀스 **x**(입력)와 시퀀스 **y**(타깃)를 만듦
  - 두 시퀀스 모두 40개의 원소로 구성
  - 예를 들어 시퀀스 **x**는 [0, 1, ..., 39] 인덱스의 원소로 구성
  - 시퀀스 **y**는 **x**보다 하나 앞서가기 때문에 인덱스 [1, 2, ..., 40]에 해당

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 그다음 map() 메서드를 사용하여 시퀀스 **x**와 **y**로 나누는 변환 함수를 적용

```
>>> seq_length = 40
>>> chunk_size = seq_length + 1
>>> ds_chunks = ds_text_encoded.batch(chunk_size,
...                                   drop_remainder=True)
>>> ## x & y를 나누기 위한 함수를 정의합니다
>>> def split_input_target(chunk):
...     input_seq = chunk[:-1]
...     target_seq = chunk[1:]
...     return input_seq, target_seq
>>> ds_sequences = ds_chunks.map(split_input_target)
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 변환된 데이터셋에서 몇 개의 샘플을 확인해 보자

```
>>> for example in ds_sequences.take(2):
...     print('입력 (x): ',
...           repr(''.join(char_array[example[0].numpy()])))
...     print('타깃 (y): ',
...           repr(''.join(char_array[example[1].numpy()])))
...     print()
입력 (x):  'THE MYSTERIOUS ISLAND ***\n\n\n\n\nProduced b'
타깃 (y):  'HE MYSTERIOUS ISLAND ***\n\n\n\n\nProduced by'
입력 (x):  ' Anthony Matonak, and Trevor Carlson\n\n\n\n'
타깃 (y):  'Anthony Matonak, and Trevor Carlson\n\n\n\n\n'
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 데이터셋 준비의 마지막 단계로 이 데이터셋을 미니 배치로 나눔
  - 데이터셋을 배치로 나누기 위해 첫 번째 전처리 단계에서 문장 조각을 만들었음
  - 각 조각이 하나의 훈련 샘플에 대응하는 문장을 표현
  - 이제 훈련 샘플을 섞고 입력을 미니 배치로 나누겠음
  - 각 배치는 여러개의 훈련 샘플을 가지고 있을 것

```
>>> BATCH_SIZE = 64
>>> BUFFER_SIZE = 10000
>>> ds = ds_sequences.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**

  문자 수준의 RNN 모델 만들기

  - 데이터셋이 준비되었으므로 모델을 만드는 것은 비교적 쉬움
  - 코드를 재사용하기 위해 케라스 Sequential 클래스로 RNN 모델을 만드는 build_model 함수를 정의
  - 그다음 매개 변수와 함께 이 함수를 호출하여 RNN 모델을 만듦

```
>>> def build_model(vocab_size, embedding_dim, rnn_units):
...     model = tf.keras.Sequential([
...         tf.keras.layers.Embedding(vocab_size, embedding_dim),
...         tf.keras.layers.LSTM(
...             rnn_units,
...             return_sequences=True),
...         tf.keras.layers.Dense(vocab_size)
...     ])
...     return model
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**

```
>>> ## 매개변수 설정
>>> charset_size = len(char_array)
>>> embedding_dim = 256
>>> rnn_units = 512
>>> tf.random.set_seed(1)
>>> model = build_model(
...     vocab_size=charset_size,
...     embedding_dim=embedding_dim,
...     rnn_units=rnn_units)
>>> model.summary()
Model: "sequential"
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 256)         20480
_________________________________________________________________
lstm (LSTM)                  (None, None, 512)         1574912
_________________________________________________________________
dense (Dense)                (None, None, 80)          41040
=================================================================
Total params: 1,636,432
Trainable params: 1,636,432
Non-trainable params: 0
_________________________________________________________________
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 이 모델의 LSTM 층의 출력 크기는 (None, None, 512)로 랭크 3
  - 첫 번째 차원은 배치 차원
  - 두 번째 차원은 출력 시퀀스 길이이고 마지막 차원은 은닉 유닛의 개수에 해당

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - LSTM 층이 랭크 3의 출력을 만드는 이유는 이 LSTM 층을 만들 때 return_sequences=True로 지정했기 때문임
  - 완전 연결 층(Dense)이 LSTM 층의 출력을 받아 출력 시퀀스의 각 원소마다 로짓을 계산
  - 이 모델의 최종 출력도 랭크 3 텐서가 됨

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 마지막 완전 연결 층을 activation=None으로 설정
  - 새로운 텍스트를 생성하기 위해 모델 예측 값에서 샘플링할 수 있도록 로짓 출력이 필요하기 때문임
  - 먼저 모델을 훈련해 보자

```
>>> model.compile(
...     optimizer='adam',
...     loss=tf.keras.losses.SparseCategoricalCrossentropy(
...         from_logits=True
...     ))
>>> model.fit(ds, epochs=20)
Epoch 1/20
424/424 [==============================] - 80s 189ms/step - loss: 2.3437
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**

```
Epoch 2/20
424/424 [==============================] - 79s 187ms/step - loss: 1.7654
...(생략)...
Epoch 20/20
424/424 [==============================] - 79s 187ms/step - loss: 1.0478
```

  - 이제 모델 평가를 통해 짧은 문자열에서 시작하여 새로운 텍스트를 생성할 수 있음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**

  평가 단계: 새로운 텍스트 생성

  - 이전 절에서 훈련한 RNN 모델은 각 문자에 대해 80개 크기의 로짓을 반환
  - 소프트맥스 함수를 사용해서 이 로짓을 쉽게 확률로 바꿀 수 있음
  - 이 확률을 사용해서 어떤 문자가 다음에 올지 결정
  - 시퀀스에서 다음 문자를 예측하기 위해 간단히 가장 큰 로짓 값을 가진 원소를 선택할 수 있음
  - 항상 가장 높은 확률을 가진 문자를 선택하는 대신 출력에서 (랜덤하게) 샘플링하려고 함
  - 이렇게 하지 않으면 모델이 항상 동일한 텍스트를 만듦
  - 텐서플로에서 제공하는 tf.random.categorical() 함수를 사용하여 범주형 분포에서 랜덤하게 샘플링할 수 있음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 어떻게 사용하는지 보기 위해 입력 로짓이 [1, 1, 1]일 때 세 개의 범주 [0, 1, 2]에서 랜덤하게 샘플링해 보겠음

```
>>> tf.random.set_seed(1)
>>> logits = [[1.0, 1.0, 1.0]]
>>> print('확률:', tf.math.softmax(logits).numpy()[0])
확률: [0.33333334 0.33333334 0.33333334]
>>> samples = tf.random.categorical(
...     logits=logits, num_samples=10)
>>> tf.print(samples.numpy())
array([[0, 0, 1, 2, 0, 0, 0, 0, 1, 0]])
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 여기서 볼 수 있듯이 로짓이 같으므로 이 범주는 동일한 확률을 가짐(즉, 범주의 선택 가능성이 동일함)
  - 샘플 크기가 크면(num_samples→∞) 각 범주가 등장할 횟수는 샘플 크기의 ≈1/3에 이를 것으로 기대할 수 있음
  - 로짓을 [1, 1, 3]으로 바꾸면 (그리고 이 로짓 분포에서 샘플링을 많이 수행하면) 범주 2가 더 많이 등장할 것

```
>>> tf.random.set_seed(1)
>>> logits = [[1.0, 1.0, 3.0]]
>>> print('확률: ', tf.math.softmax(logits).numpy()[0])
확률: [0.10650698 0.10650698 0.78698605]
>>> samples = tf.random.categorical(
...     logits=logits, num_samples=10)
>>> tf.print(samples.numpy())
array([[2, 0, 2, 2, 2, 0, 1, 2, 2, 0]])
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - tf.random.categorical 함수를 사용하면 모델이 출력한 로짓을 기반으로 문자를 생성할 수 있음
  - 짧은 시작 문자열 starting_str을 받아 새로운 generated_str을 생성하는 sample() 함수를 정의
  - generated_str은 초기에 입력 값으로 설정
  - 그다음 generated_str의 마지막에서 max_input_length 크기의 문자열을 선택하여 정수 시퀀스 encoded_input으로 인코딩
  - encoded_input을 RNN 모델에 전달하여 로짓을 계산
  - 이 RNN 모델의 마지막 순환 층에서 return_sequences=True로 설정했기 때문에 입력 시퀀스와 동일한 길이의 로짓 시퀀스가 출력
  - RNN 모델의 출력에 있는 각 원소는 모델이 입력 시퀀스를 관찰한 후 다음 문자를위한 로짓을 표현(여기서는 전체 문자 개수인 80개의 원소를 가진 벡터)

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 여기서 출력 logits의 마지막 원소($\mathbf{o}^{(T)}$)만 tf.random.categorical() 함수로 전달하여 새로운 샘플을 생성
  - 새로운 샘플을 문자로 변환하고 생성된 문자열 generated_text 끝에 추가하여 길이를 1만큼 늘림
  - 그다음 이 과정을 반복
  - 지정한 문자 길이만큼 생성될 때까지 generated_text에서 마지막 max_input_length개의 문자를 선택하고 이를 사용하여 새로운 문자를 생성
  - 새로운 원소를 만들기 위해 생성된 시퀀스를 입력으로 사용하는 과정을 자기회귀(auto-regression)라고 부름

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - sample() 함수의 코드는 다음과 같음

```
>>> def sample(model, starting_str,
...            len_generated_text=500,
...            max_input_length=40,
...            scale_factor=1.0):
...     encoded_input = [char2int[s] for s in starting_str]
...     encoded_input = tf.reshape(encoded_input, (1, -1))
...
...     generated_str = starting_str
...
...     model.reset_states()
...     for i in range(len_generated_text):
...         logits = model(encoded_input)
...         logits = tf.squeeze(logits, 0)
...
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**

```
...         scaled_logits = logits * scale_factor
...         new_char_indx = tf.random.categorical(
...             scaled_logits, num_samples=1)
...
...         new_char_indx = tf.squeeze(new_char_indx)[-1].numpy()
...
...         generated_str += str(char_array[new_char_indx])
...
...         new_char_indx = tf.expand_dims([new_char_indx], 0)
...         encoded_input = tf.concat(
...             [encoded_input, new_char_indx],
...             axis=1)
...         encoded_input = encoded_input[:, -max_input_length:]
...
...     return generated_str
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 그럼 이제 새로운 텍스트를 생성해 보자

```
>>> tf.random.set_seed(1)
>>> print(sample(model, starting_str='The island'))
```

```
The island is probable that the view of the vegetable discharge on unexplainst felt, a
thore, did not
refrain it existing to the greatest
possing bain and production, for a hundred streamled
established some branches of the
holizontal direction. It was there is all ready, from one things from
contention of the Pacific
acid, and
according to an occurry so
summ on the rooms. When numbered the prud Spilett received an exceppering from their
head, and by went inhabited.
"What are the most abundance a report
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 결과에서 볼 수 있듯이 이 모델은 거의 정확한 단어를 생성
  - 몇몇 문장은 부분적으로 의미가 있음
  - 훈련 시 입력 시퀀스, 모델 구조, 샘플링 파라미터(예를 들어 max_input_length) 같은 훈련 파라미터를 더 튜닝해 볼 수 있음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 생성된 샘플의 예측 가능성을 조절하기 위해(즉, 생성된 텍스트가 훈련 텍스트에서 학습한 패턴을 따르게 할지 랜덤하게 생성할지 조절하기 위해) RNN이 계산한 로짓을 tf.random.categorical() 샘플링 함수로 전달하기 전에 스케일을 조정할 수 있음
  - 스케일링 인자 α를 물리학에 있는 온도의 역수로 해석할 수 있음
  - 온도가 높으면 무작위성이 커지고 온도가 낮으면 예측 가능한 행동을 만듦

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - α < 1로 로짓의 스케일을 조정하면 소프트맥스 함수가 계산한 확률은 다음 코드처럼 더 균일해짐

```
>>> logits = np.array([[1.0, 1.0, 3.0]])
>>> print('스케일 조정 전의 확률: ',
...       tf.math.softmax(logits).numpy()[0])
>>> print('0.5배 조정 후 확률:  ',
...       tf.math.softmax(0.5*logits).numpy()[0])
>>> print('0.1배 조정 후 확률:  ',
...       tf.math.softmax(0.1*logits).numpy()[0])
스케일 조정 전의 확률:  [0.10650698 0.10650698 0.78698604]
0.5배 조정 후 확률:   [0.21194156 0.21194156 0.57611688]
0.1배 조정 후 확률:   [0.31042377 0.31042377 0.37915245]
```

  - 여기서 볼 수 있듯이 α =0.1로 로짓의 스케일을 조정하면 거의 균등한 확률 [0.31, 0.31, 0.38]을 얻음

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - 아래에서 α =2.0과 α =0.5로 생성한 텍스트를 비교해 보자

    - α=2.0 → 예측 가능성이 높아짐

```
>>> tf.random.set_seed(1)
>>> print(sample(model, starting_str='The island',
...              scale_factor=2.0))
The island spoke of heavy torn into the island from the sea.
The noise of the inhabitants of the island was to be feared that the colonists had
come a project with a straight be put to the bank of the island was the surface of
the lake and sulphuric acid, and several supply of her animals. The first stranger
carried a sort of accessible to break these screen barrels to their distance from
the palisade.
"The first huntil," said the reporter, "and his companions the reporter extended to
build a few days a
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - α=0.5 → 무작위성이 높아짐

```
>>> tf.random.set_seed(1)
>>> print(sample(model, starting_str='The island',
...              scale_factor=0.5))
The island
glissed in
ascercicedly useful? loigeh, Cyrus,
Spileots," henseporvemented
House to a left
the centlic moment. Tonsense craw.
Pencrular ed/ of times," tading had coflently often above anzand?"
"Wat;" then:y."
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**

```
Ardivify he acpearly, howcovered--he hassime; however, fenquests hen adgents!'.? Let
us Neg eqiAl?.
GencNal, my surved thirtyin" ou; is Harding; treuths. Osew apartarned. "N,
the poltuge of about-but durired with purteg.
Chappes wason!
Fears," returned Spilett; "if
you tear 8t trung
```

16.3 텐서플로로 시퀀스 모델링을 위한 RNN 구현

- **두 번째 프로젝트: 텐서플로로 글자 단위 언어 모델 구현**
  - α =0.5로 로짓의 스케일을 조정하면 (온도를 높이면) 더 랜덤한 텍스트가 생성
  - 올바른 텍스트와 신선한 텍스트 생성 사이에서 절충점을 찾아야 함
  - 이 절에서 시퀀스-투-시퀀스(sequence-to-sequence, seq2seq) 모델링 작업인 문자 수준의 텍스트 생성 문제를 다루었음
  - 이것 자체로는 아주 유용하지는 않지만 이런 종류의 모델에 맞는 애플리케이션이 있음
  - 예를 들어 비슷한 RNN 모델을 훈련하여 간단한 질문에 답변하는 챗봇으로 사용할 수 있음

16.4 트랜스포머 모델을 사용한 언어 이해

16.4 트랜스포머 모델을 사용한 언어 이해

- **트랜스포머 모델을 사용한 언어 이해**
  - 이 장에서 RNN 기반의 신경망으로 두 가지 시퀀스 모델링 문제를 풀었음
  - 최근에 새롭게 등장한 한 모델 구조가 여러 NLP 작업에서 RNN 기반의 seq2seq 모델의 성능을 능가하는 것으로 나타났음

16.4 트랜스포머 모델을 사용한 언어 이해

- **트랜스포머 모델을 사용한 언어 이해**
  - 이 구조가 **트랜스포머**(Transformer)
  - 2017년 NeurIPS 논문에서 소개되었으며 입력과 출력 시퀀스 사이에 있는 전역 의존성(global dependency)을 모델링할 수 있음
  - 트랜스포머 구조는 **어텐션**(attention)이라는 개념을 기반으로 함
  - 좀 더 구체적으로는 **셀프 어텐션 메커니즘**(self-attention mechanism)을 기반으로 함
  - 이 경우에 어텐션 메커니즘을 사용하면 감성에 관련이 높은 입력 시퀀스의 일부분에 초점을 맞추어서 훈련할 수 있음

16.4 트랜스포머 모델을 사용한 언어 이해

- **셀프 어텐션 메커니즘 이해**
  - NLP 작업에서 트랜스포머 모델이 중요한 시퀀스 부분에 초점을 맞추는 데 어떻게 도움을 주는지 설명
  - 첫 번째 절에서 셀프 어텐션의 기본 구조를 소개하면서 텍스트 표현을 학습하는 이면에 있는 전반적인 아이디어를 설명
  - 그 다음 여러 가중치 파라미터를 추가하여 트랜스포머 모델에서 사용하는 셀프 어텐션 메커니즘을 완성

16.4 트랜스포머 모델을 사용한 언어 이해

- **셀프 어텐션 메커니즘 이해**

  셀프 어텐션 기본 구조

  - 셀프 어텐션의 기본 아이디어를 소개하기 위해 길이가 T인 입력 시퀀스 $\mathbf{x}^{(0)}$, $\mathbf{x}^{(1)}$, ..., $\mathbf{x}^{(T)}$와 출력 시퀀스 $\mathbf{o}^{(0)}$, $\mathbf{o}^{(1)}$, ..., $\mathbf{o}^{(T)}$가 있다고 가정하겠음
  - 시퀀스의 각 원소 $\mathbf{x}^{(t)}$와 $\mathbf{o}^{(t)}$는 크기가 d인 벡터(즉, $\mathbf{x}^{(T)} \in R^d$)
  - 그다음 seq2seq 작업에서 셀프 어텐션은 입력 원소에 대한 출력 시퀀스에 있는 각 원소의 의존성을 모델링하는 것이 목적
  - 이를 위해 어텐션 메커니즘은 세 단계로 구성
  - 첫째 현재 원소와 시퀀스에 있는 다른 모든 원소 사이의 유사도를 기반으로 중요도 가중치를 계산
  - 둘째 익숙한 소프트맥스 함수를 사용하여 이 가중치를 정규화함
  - 셋째 이 가중치를 해당하는 시퀀스 원소와 결합하여 어텐션 값을 계산

16.4 트랜스포머 모델을 사용한 언어 이해

- **셀프 어텐션 메커니즘 이해**
  - 식으로 나타내면 셀프 어텐션의 출력은 모든 입력 시퀀스의 가중치 합
  - 예를 들어 i번째 입력 원소에 해당하는 출력은 다음과 같이 계산

$$\boldsymbol{o}^{(i)} = \sum_{j=0}^{T} \boldsymbol{W}_{ij} \boldsymbol{x}^{(j)}$$

16.4 트랜스포머 모델을 사용한 언어 이해

- **셀프 어텐션 메커니즘 이해**
  - 여기서 가중치 $\mathbf{W}_{ij}$는 현재 입력 원소 $\mathbf{x}^{(i)}$와 입력 시퀀스에 있는 다른 모든 원소 사이의 유사도를 기반으로 계산
  - 좀 더 구체적으로 말하면 이 유사도는 현재 입력 원소 $\mathbf{x}^{(i)}$ 와 입력 시퀀스에 있는 다른 원소 $\mathbf{x}^{(j)}$의 점곱으로 계산

$$\omega_{ij} = \boldsymbol{x}^{(i)\top}\boldsymbol{x}^{(j)}$$

16.4 트랜스포머 모델을 사용한 언어 이해

- **셀프 어텐션 메커니즘 이해**
  - i번째 입력과 시퀀스에 있는 모든 입력(**x**$^{(i)}$에서 **x**$^{(T)}$까지)에 대해 유사도 기반 가중치를 계산한 후
  - 입력 가중치(ω_{ij})에서 (ω_{iT})까지를 다음과 같이 소프트맥스 함수로 정규화함

$$W_{ij} = \frac{\exp(\omega_{ij})}{\sum_{j=0}^{T} \exp(\omega_{ij})} = softmax\left([\omega_{ij}]_{j=0\ldots T}\right)$$

  - [illegible]를 적용하기 때문에 정규화된 가중치 합은 1이 됨

$$\sum_{j=0}^{T} W_{ij} = 1$$

16.4 트랜스포머 모델을 사용한 언어 이해

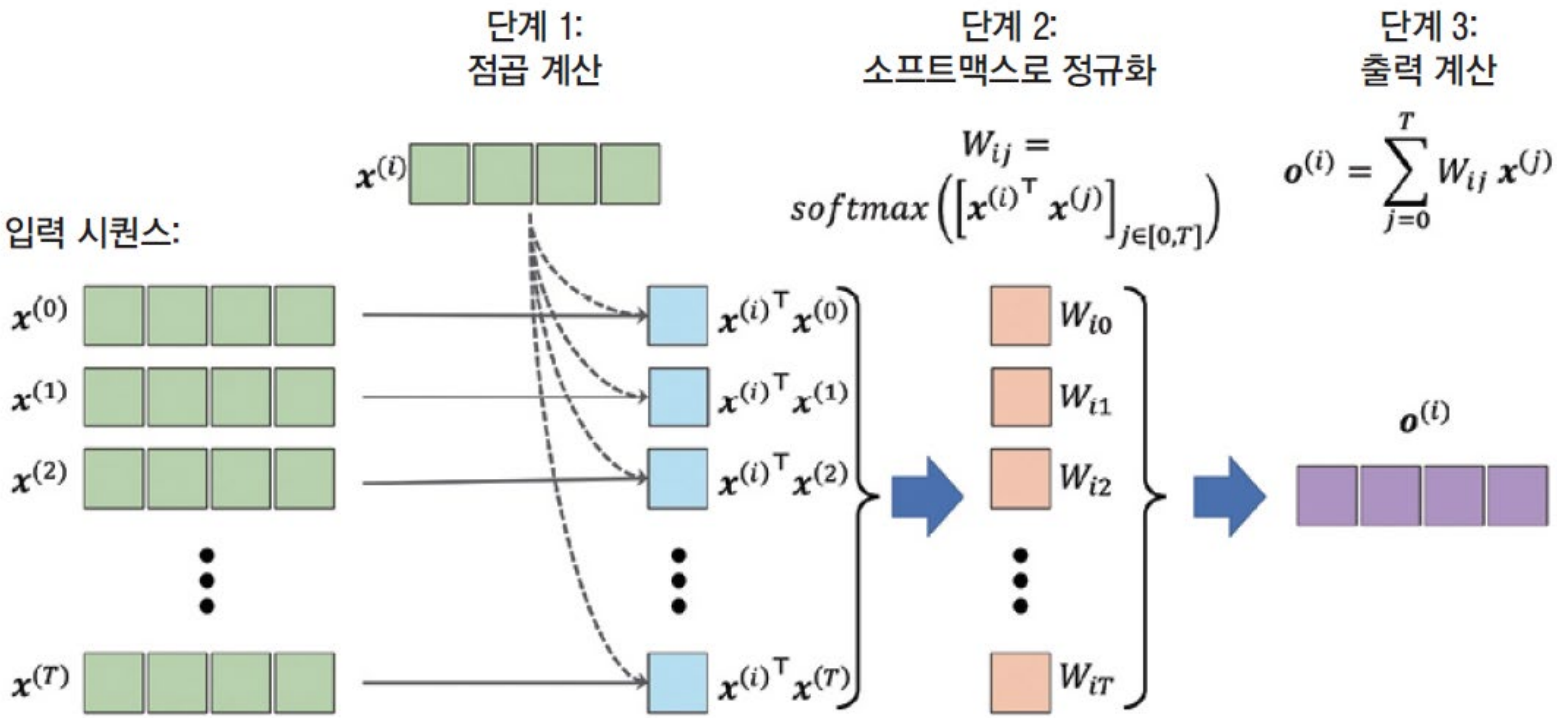
- **셀프 어텐션 메커니즘 이해**
  - 정리하면 셀프 어텐션 연산의 주요 세 단계는 다음과 같음

    - 주어진 입력 원소 $\mathbf{x}^{(i)}$와 [0, T] 범위에 있는 j 번째 원소에 대해 점곱 $\mathbf{x}^{(i)T}\mathbf{x}^{(j)}$를 계산
    - 소프트맥스 함수로 이 점곱을 정규화하여 가중치 $\mathbf{W}_{ij}$를 얻음
    - 전체 입력 시퀀스에 대한 가중치 합으로 출력 $\mathbf{o}^{(i)}$를 계산

$$\boldsymbol{o}^{(i)} = \sum_{j=0}^{T} W_{ij}\, \boldsymbol{x}^{(j)}$$

16.4 트랜스포머 모델을 사용한 언어 이해

▼ 그림 16-15 셀프 어텐션 과정



16.4 트랜스포머 모델을 사용한 언어 이해

- **셀프 어텐션 메커니즘 이해**

  쿼리, 키, 값 가중치를 가진 셀프 어텐션 메커니즘

  - 이 절에서는 트랜스포머 모델에서 사용하는 고급 셀프 어텐션 메커니즘을 정리해 보겠음
  - 이전 절에서는 출력을 계산할 때 학습되는 파라미터를 전혀 사용하지 않았음
  - 언어 모델을 훈련할 때 분류 오차를 최소화하는 것 같이 목적 함수를 최적화하려면 입력 원소 $\mathbf{x}^{(i)}$ 가 되는 단어 임베딩(즉, 입력 벡터)을 바꾸어야 함
  - 다르게 말해 앞서 소개한 기본적인 셀프 어텐션 메커니즘을 사용하면 트랜스포머 모델이 주어진 시퀀스에서 모델을 최적화하는 동안 어텐션 값을 바꾸거나 업데이트하는 데 제한적임
  - 셀프 어텐션 메커니즘을 모델 최적화에 대해 유연하고 적응할 수 있게 만들기 위해 추가적인 가중치 행렬을 사용
  - 이 가중치는 모델을 훈련하는 동안 학습되는 모델 파라미터

16.4 트랜스포머 모델을 사용한 언어 이해

- **셀프 어텐션 메커니즘 이해**
  - 이 세 가중치 행렬을 $\mathbf{U}_q$, $\mathbf{U}_K$, $\mathbf{U}_V$로 표시
  - 이 가중치는 입력을 쿼리(query), 키(key), 값(value) 시퀀스로 만들기 위해 사용

    - **쿼리 시퀀스:** $\mathbf{q}^{(i)}=\mathbf{U}_q\mathbf{x}^{(i)}$ $i\in[0, T]$일 때
    - **키 시퀀스:** $\mathbf{k}^{(i)}=\mathbf{U}_K\mathbf{x}^{(i)}$ $i\in[0, T]$일 때
    - **값 시퀀스:** $\mathbf{v}^{(i)}=\mathbf{U}_V\mathbf{x}^{(i)}$ $i\in[0, T]$일 때

16.4 트랜스포머 모델을 사용한 언어 이해

- **셀프 어텐션 메커니즘 이해**
  - 여기서 $\mathbf{q}^{(i)}$와 $\mathbf{k}^{(i)}$ 는 크기가 d_k인 벡터
  - 투영된 행렬 $\mathbf{U}_q$와 $\mathbf{U}_k$의 크기는 $d_k \times d$
  - 반면 $\mathbf{U}_v$의 크기는 $d_v \times d$
  - 설명하기 좋게 이 벡터의 크기를 $m=d_k=d_v$로 동일하게 가정하겠음
  - 입력 시퀀스 원소 $\mathbf{x}^{(i)}$와 j번째 시퀀스 원소 $\mathbf{x}^{(j)}$ 사이에 점곱으로 정규화되지 않은 가중치를 계산하는 대신에 쿼리와 키 사이에 점곱을 계산

$$\omega_{ij} = \boldsymbol{q}^{(i)^{\top}} \boldsymbol{k}^{(j)}$$

16.4 트랜스포머 모델을 사용한 언어 이해

- **셀프 어텐션 메커니즘 이해**
  - 그다음 소프트맥스 함수로 가중치 ω_{ij}를 정규화하고 $1/\sqrt{m}$로 스케일을 조정함

$$W_{ij} = softmax\left(\frac{\omega_{ij}}{\sqrt{m}}\right)$$

  - $1/\sqrt{m}$로 ω_{ij}의 스케일을 조정하면 가중치 벡터의 유클리드 길이가 거의 같은 범위에 있게 됨

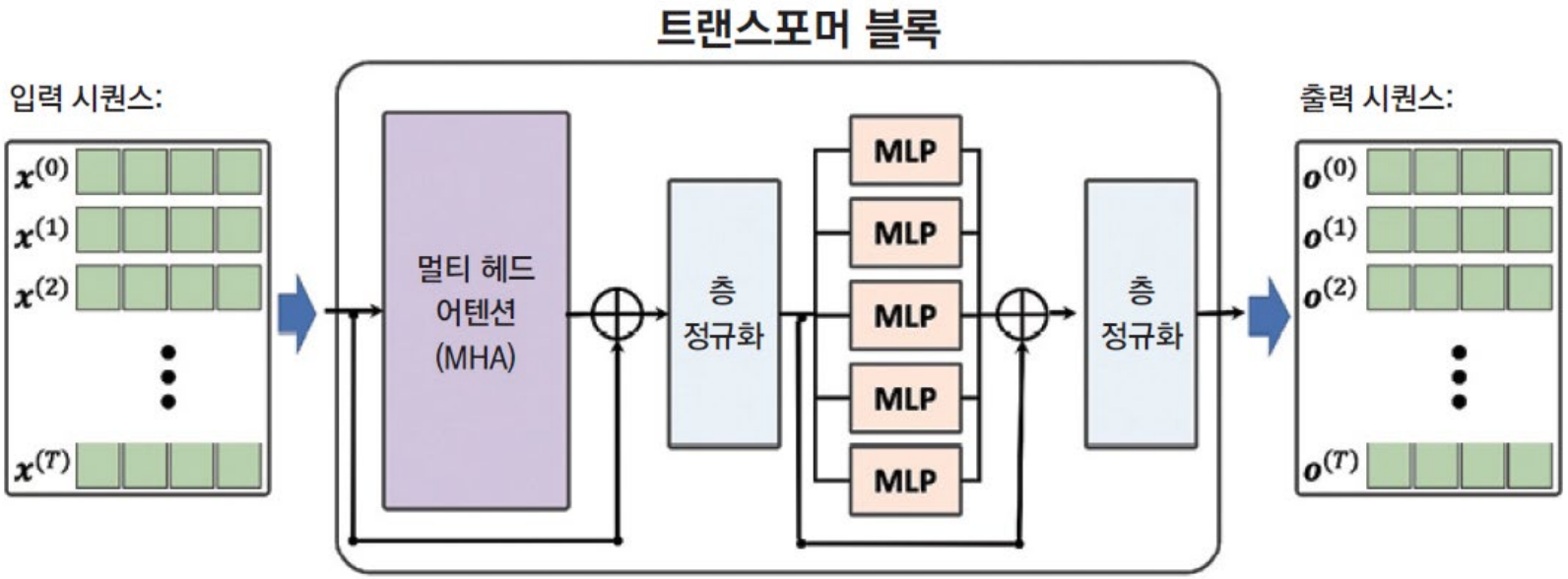
16.4 트랜스포머 모델을 사용한 언어 이해

- **멀티-헤드 어텐션과 트랜스포머 블록**
  - 셀프 어텐션 메커니즘의 판별 능력을 크게 높이는 또 다른 기술은 여러 개의 셀프 어텐션 연산을 합친 **멀티 헤드 어텐션**(Multi-Head Attention, MHA)
  - 이 경우 각 셀프 어텐션 메커니즘을 헤드(head)라고 부르며 병렬로 계산할 수 있음
  - r개의 병렬 헤드를 사용하여 각 헤드는 크기가 m인 벡터 **h**를 만듦
  - 이 벡터를 연결하여 크기가 r×m인 벡터 **z**를 얻음
  - 마지막으로 이 연결된 벡터와 출력 행렬 $\mathbf{W}^o$를 점곱하여 다음과 같이 최종 출력을

$$\boldsymbol{o}^{(i)} = W_{ij}^{o}\,\boldsymbol{z}$$

16.4 트랜스포머 모델을 사용한 언어 이해

▼ 그림 16-16 트랜스포머 블록



16.4 트랜스포머 모델을 사용한 언어 이해

- **멀티-헤드 어텐션과 트랜스포머 블록**
  - 그림 16-16에 나온 트랜스포머 구조에는 아직 언급하지 않은 두 가지 구성 요소가 추가되어 있음
  - 이 중 하나는 **잔차 연결**(residual connection)
  - 잔차 연결은 층(또는 층 그룹)의 출력을 입력에 더함
  - 즉, **x** +layer(**x**)
  - 잔차 연결로 층(또는 층 그룹)을 구성하는 블록을 **잔차 블록**(residual block)이라고 함
  - 앞의 그림에 나온 트랜스포머 블록은 두 개의 잔차 블록을 포함

16.4 트랜스포머 모델을 사용한 언어 이해

- **멀티-헤드 어텐션과 트랜스포머 블록**
  - 다른 하나는 **층 정규화**(layer normalization)
  - 17장에서 소개할 배치 정규화를 포함하여 정규화 층의 한 종류
  - 지금은 층 정규화를 각 층에서 신경망의 입력과 활성화 출력을 정규화 또는 스케일을 조정하는 고급 방법이라고 생각해도 좋음

16.4 트랜스포머 모델을 사용한 언어 이해

- **멀티-헤드 어텐션과 트랜스포머 블록**
  - 그림 16-16에 있는 트랜스포머 모델로 돌아가서 이 모델이 어떻게 동작하는지 설명해 보겠음
  - 먼저 입력 시퀀스가 앞서 언급한 셀프 어텐션 메커니즘을 기반으로 하는 MHA 층으로 전달
  - 또한, 입력 시퀀스가 잔차 연결을 통해 MHA 층의 출력에 더해짐
  - 이렇게 하면 훈련하는 동안 앞쪽의 층이 충분한 그레이디언트 신호를 받게 됨
  - 훈련 속도와 수렴을 향상시키기 위해 자주 사용하는 기법
  - 관심 있다면 잔차 연결 개념에 대한 자세한 내용은 관련 논문을 참고

16.4 트랜스포머 모델을 사용한 언어 이해

- **멀티-헤드 어텐션과 트랜스포머 블록**
  - 입력 시퀀스가 MHA 층의 출력에 더해진 후 이 출력이 층 정규화를 통해 정규화됨
  - 정규화된 신호가 연속된 MLP(완전 연결) 층과 잔차 연결을 통과
  - 마지막으로 잔차 블록의 출력을 다시 정규화하여 출력 시퀀스로 반환해서 시퀀스 분류나 시퀀스 생성에 사용할 수 있음
  - 트랜스포머 모델을 구현하고 훈련하는 자세한 설명은 포함하지 못했음
  - 관심 있는 독자는 텐서플로 공식 문서에 있는 훌륭한 구현과 설명을 참고

16.5 요약

16.5 요약

- **요약**
  - 이 장에서 먼저 구조적인 데이터나 이미지 같은 데이터와 다른 시퀀스의 성질에 대해 배웠음
  - 그다음 시퀀스 모델링을 위해 RNN의 기초를 다루었음
  - 기본적인 RNN 모델의 동작 방식을 배웠고 시퀀스 데이터의 장기간 의존성 감지에 관한 제약을 설명했음
  - 다음으로 기본 RNN 모델에서 종종 발생하는 그레이디언트 폭주와 소실 영향을 줄이기 위한 게이트(gate) 메커니즘으로 구성된 LSTM 셀을 다루었음

16.5 요약

- **요약**
  - RNN의 주요 개념을 설명한 후 케라스 API를 사용하여 여러 가지 순환 층으로 몇 개의 RNN 모델을 구현했음
  - 특히 감성 분석을 위한 RNN 모델과 텍스트 생성을 위한 RNN 모델을 만들었음
  - 마지막으로 관련 있는 시퀀스 부분에 초점을 맞추기 위해 셀프 어텐션 메커니즘을 활용하는 트랜스포머 모델을 다루었음