Chapter 14. Recurrent Neural Networks

1. **Recurrent Neurons**

**- Memory Cells**

**- Input and Output Sequences**

1. **Basic RNNs in TensorFlow**

**- Static Unrolling Through Time**

**- Dynamic Unrolling Through Time**

**- Handling Variable Length Input Sequences**

**- Handling Variable-Length Output Sequences**

1. **Training RNNs**

**- Training a Sequence Classifier**

**- Training to Predict Time Series**

**- Creative RNN**

1. **Deep RNNs**

**- Distributing a Deep RNN Across Multiple GPUs**

**- Applying Dropout**

**- The Difficulty of Training over Many Time Steps**

1. **LSTM Cell**

**- Peephole Connections**

1. **GRU Cell**
2. **Natural Language Processing**

**- Word Embeddings**

**- An Encoder– Decoder Network for Machine Translation**

Recurrent Neural Networks

타자가 공을 때린다. 공의 궤도를 예상하면서 즉시 달리기를 시작합니다. 당신은 그것을 추적하고 움직임을 적응시키고, 마지막으로 그것을 잡습니다.

미래를 예측하는 것은 당신이 친구의 문장을 끝내거나 아침 식사 때 커피 냄새를 얘기하고 있든지간에 항상하는 일입니다. 이 장에서는 미래를 예측할 수있는 recurrent neural networks (RNN)(이하 순환신경망)을 논의 할 것입니다. 물론 미래까지도. 그들은 주가와 같은 시계열 데이터를 분석하고 언제 사거나 팔것을 말할 수 있습니다. 자율 주행 시스템에서는 자동차 궤도를 예측하고 사고를 피할 수 있습니다. 보다 일반적으로, 우리는 지금까지 논의한 모든 네트워크와 같은 고정 크기의 입력보다는 임의의 길이의 순서로 작업 할 수 있습니다. 예를 들어, 문장, 문서 또는 오디오 샘플을 입력으로 가져 와서 자동 번역, 음성 텍스트 또는 감정 분석과 같은 자연 언어 처리 (NLP) 시스템에 매우 유용합니다.

또한 RNN의 예상 능력으로 인해 놀라운 창의력을 발휘할 수 있습니다. 멜로디에서 가장 가능성이 높은 다음 음표를 예측 한 다음 임의로 이 음표 중 하나를 선택하여 연주하도록 요청할 수 있습니다. 그런 다음 가장 가능성이 높은 다음 음표를 인터넷에 요청하여 재생하고 반복적으로 과정을 반복하십시오. 그것을 알기도 전에 인터넷은 Google의 Magenta 프로젝트에서 제작한 것과 같은 멜로디를 구성합니다.

마찬가지로 RNN은 문장, 이미지 캡션 등을 생성 할 수 있습니다. 그 결과는 셰익스피어 나 모짜르트가 아니지만 지금부터 몇 년 후에 무엇을 생산할 것인지 누가 알 수 있습니까?

이 장에서는 RNN의 근본적인 개념, 즉 그들이 직면하는 주요 문제와 LSTM 및 GRU 셀과 싸우기 위해 널리 사용되는 솔루션을 살펴볼 것입니다. TensorFlow를 사용하여 RNN을 구현하는 방법을 보여줍니다. 마지막으로 기계 번역 시스템의 아키텍처를 살펴 보겠습니다.

1. **Recurrent Neurons**

지금까지는 피드 포워드 신경 네트워크를 주로 보았습니다. 여기서는 활성화가 입력 레이어에서 출력 레이어로만 한 방향으로만 흐릅니다. 순환 신경망은 피드 포워드 신경망과 매우 흡사합니다. 단, 역방향 연결도 있습니다.

그림 14-1 (왼쪽)에서 볼 수 있듯이, 입력을 수신하고, 출력을 생성하고, 그 출력을 다시 자체로 보내는 하나의 뉴런으로 구성된 가장 간단한 가능한 RNN을 살펴 보겠습니다. 각 시간 단계 t (프레임이라고도 함)에서 이 반복적인 뉴런은 이전 시간 단계 y (t-1)에서 자신의 출력뿐만 아니라 입력 x (t)도받습니다.

그림 14-1 (오른쪽)과 같이 시간 축에 대해 이 작은 네트워크를 나타낼 수 있습니다. 이것은 시간이 지남에 따라 네트워크 풀림이라고합니다.

Y(b3) 
X(t-2) 
Time 
Figure 14-1. A recurrent neuron (left), unrolled 
through time (right) 

반복적인 뉴런 레이어를 쉽게 만들 수 있습니다. 각 시간 단계 t에서, 모든 뉴런은 그림 14-2에서와 같이 입력 벡터 x (t)와 이전 시간 스텝 y (t-1)의 출력 벡터를 모두 받습니다. 입력과 출력은 모두 벡터입니다 (단일 뉴런이있을 때 출력은 스칼라 였음).

(0) 
(1) 
Time 
Figure 14-2. A layer of recurrent neurons (left), 
unrolled through time (right) 

각각의 반복 뉴런은 두 세트의 가중치를 가진다 : 하나는 입력 x (t)에 대한 것이고 다른 하나는 이전 시간 스텝의 출력에 대한 y (t-1)이다. 이 가중치 벡터를 wx와 wy라고합시다. 단지 하나의 반복 뉴런 대신 전체 반복층을 고려한다면, 모든 가중치 벡터를 Wx와 Wy의 두 가중치 행렬에 둘 수 있습니다. 그러면 전체 반복 레이어의 출력 벡터는 식 14-1과 같이 예상대로 계산할 수 있습니다

(b는 바이어스 벡터이고 φ (·)는 활성화 함수, 예를 들어 ReLU1이다).

Equation 14-1. Output of a recurrent layer for a 
single instance 

피드 포워드 신경망과 마찬가지로 입력 행렬 X (t)에 시간 단계 t에서 모든 입력을 놓음으로써 전체 미니 배치에 대한 반복적 인 레이어의 출력을 한 번에 계산할 수 있습니다 (식 14-2 참조).

Equation 14-2. Outputs of a layer of recurrent 
neurons for all instances in a mini-batch 
= with W = 

* Y (t)는 미니 배치의 각 인스턴스에 대해 시간 단계 t에서 계층의 출력을 포함하는 mxnururons 행렬입니다 (m은 미니 배치의 인스턴스 수이고 nneurons는 뉴런 수임).
* X (t)는 모든 인스턴스에 대한 입력을 포함하는 m × ninputs 행렬입니다 (ninputs는 입력 피쳐의 수입니다).
* Wx는 현재 시간 간격의 입력에 대한 연결 가중치를 포함하는 ninputs × nneurons 행렬입니다.
* Wy는 이전 시간 단계의 출력에 대한 연결 가중치가 포함 된 nneurons × nneurons 행렬입니다.
* b는 각 뉴런의 바이어스 항을 포함하는 크기 n 뉴런의 벡터입니다.
* 가중치 행렬 Wx 및 Wy는 종종 수직적으로 단일 가중치 행렬 W (ninputs + nneurons) × nneurons으로 연결됩니다 (식 14-2의 두 번째 행 참조).
* 표기법 [X (t) Y (t-1)]은 행렬 X (t)와 Y (t-1)의 수평 연결을 나타냅니다.

Y (t)는 X (t-1)와 Y (t-2)의 함수 인 X (t)와 Y 2) 및 Y (t-3) 등이 있습니다. 이것은 시간 t = 0 (즉, X (0), X (1), ..., X (t)) 이후 Y (t)를 모든 입력의 함수로 만든다. 첫 번째 단계 t = 0에서 이전 출력이 없으므로 일반적으로 모두 0으로 가정합니다.

**1.1 Memory Cells**

시간 단계 t에서 반복적인 뉴런의 출력은 이전 시간 단계의 모든 입력 값의 함수이므로 메모리의 형태를 가질 수 있습니다. 시간 단계에 따라 일부 상태를 보존하는 신경망의 일부를 메모리 셀 (또는 단순히 셀)이라고합니다. 하나의 반복적인 뉴런 또는 반복적인 뉴런의 층은 매우 기본적인 세포이지만 이 장의 뒷부분에서 좀 더 복잡하고 강력한 유형의 세포를 살펴볼 것입니다.

일반적으로 h (t) ( "h"는 "hidden"을 나타냄)로 표시된 시간 단계 t에서의 셀의 상태는 이전 시간 단계에서 그 시간 단계 및 그 상태에서의 입력의 함수이다.

h = f (h (t-1), x (t))

y (t)로 표시된 시간 단계 t에서의 출력은 이전 상태와 현재 입력의 함수이기도합니다. 지금까지 설명한 기본 셀의 경우 출력은 상태와 동일하지만 더 복잡한 셀의 경우 항상 그렇지는 않습니다 (그림 14-3 참조).

Time 
Figure 14-3. A cell's hidden state and its output 
may be different 

**1.2 Input and Output Sequences**

(그림 14-4, 왼쪽 상단 네트워크 참조)

RNN은 일련의 입력을 동시에 받아 일련의 출력을 생성 할 수 있습니다 . 예를 들어,이 유형의 네트워크는 주가와 같은 시계열을 예측하는 데 유용합니다. 마지막 N 일 동안의 가격을 피드로 제공하고 미래에 1 일 이동한 가격을 출력해야합니다 (예 : N - 1 일 내일부터).

(오른쪽 상단 네트워크 참조)

네트워크에 일련의 입력을 제공하고 마지막 입력을 제외한 모든 출력을 무시할 수 있습니다 . 다시 말해, 이것은 sequence-to-vector 네트워크입니다. 예를 들어 영화 리뷰에 해당하는 일련의 단어를 네트워크에 제공 할 수 있으며 네트워크는 정서 점수를 출력합니다 (예 : -1 [증오]에서 +1 [사랑]).

(0) 
(0) 
(1) 
(1) 
(2) 
(2) 
(3) 
(3) 
Ignored outputs 
Encoder 
Figure 14-4. Seq to seq (top left), seq to vector 
(top right), vector to seq (bottom left), delayed 
seq to seq (bottom right) 

(왼쪽 하단 네트워크 참조)

반대로 첫 번째 단계에서 네트워크에 단일 입력 (다른 모든 시간 단계는 0)을 제공하고 시퀀스 출력을 허용 할 수 있습니다 . 이것은 벡터 대 시퀀스 네트워크입니다. 예를 들어 입력은 이미지일 수 있으며 출력은 해당 이미지의 캡션이 될 수 있습니다.

(오른쪽 하단 네트워크 참조)

인코더라고하는 시퀀스 - 벡터 네트워크와 디코더라고하는 벡터 - 시퀀스 네트워크를 사용할 수 있습니다 .

예를 들어, 한 언어에서 다른 언어로 문장을 번역하는 데 사용할 수 있습니다. 네트워크에 하나의 언어로 문장을 보내면 인코더는 이 문장을 단일 벡터 표현으로 변환한 다음 디코더가 이 벡터를 다른 언어의 문장으로 디코딩합니다. Encoder-Decoder라고 하는 이 두 단계 모델은 시퀀스 순차 RNN(왼쪽 상단에 표시된 것)을 사용하여 즉석에서 번역하는 것보다 효과적입니다 .

문장의 마지막 단어가 번역의 첫 단어에 영향을 줄 수 있기 때문에, 번역하기 전에 전체 문장을 들을 때까지 기다려야합니다.

1. **Basic RNNs in TensorFlow**

먼저 TensorFlow의 RNN 작업을 사용하지 않고 매우 단순한 RNN 모델을 구현하여 두드러진 상황을보다 잘 이해합시다. 우리는 tanh 활성화 함수를 사용하여 5 개의 순환 뉴런 (그림 14-2에 표시된 RNN과 같은)의 레이어로 구성된 RNN을 생성합니다. RNN은 각 시간 단계에서 크기가 3 인 입력 벡터를 사용하여 두 단계 만 실행한다고 가정합니다.

(0) 
(1) 
Time 
Figure 14-2. A layer of recurrent neurons (left), 
unrolled through time (right) 

다음 코드는 이 RNN을 빌드하고 두 단계를 거쳐 전개합니다.

In [3] : 
reset _ graph ( ) 
XO = 
n _ inputs 
n neurons = 
5 
tf.placeholder (tf. float32, [None, n_inputs] ) 
XI = tf.placeholder (tf. float32, [None, n_inputs] ) 
tf.Variab1e(tf .random_normal (shape=[n_inputs, n_neurons] flo 
Wx = 
at32)) 
Wy = tf.Variab1e(tf . random normal (shape=[n neurons,n neurons] . flo 
at32)) 
b = tf. zeros( [1, n _ neurons], dtype=tf float32) ) 
YO 
= tf.tanh(tf .matmu1(XO, 
YI = tf.tanh(tf 
Wx) + b) 
WY) + tf.matmu1(X1, Wx) + b) 
init = tf. global _ variables _ initializer( ) 

이 네트워크는 두 계층의 피드 포워드 신경망과 비슷합니다. 먼저 동일한 가중치와 바이어스 조건이 두 계층에서 공유되며 두 번째로 각 계층에서 입력을 공급하고 각 계층의 출력을 얻습니다. 모델을 실행하려면 두 시간 단계에서 입력을 피드에 제공해야합니다.

In 
7, 
5, 
2, 
import numpy as np 
XO_batch = 1, 
Xl_batch = [9, 8, 
with tf.Session() as sess: 
init. run( ) 
7], 
[O, 
4, 
o, 
5], 
0], 
[6, 
8], 
4], 
YO_va1, Yl val = 
sess . run( [YO , 
tch} ) 
[9, 
[3, 
XO batch, 
Xl: Xl ba 

이 미니 배치에는 4 개의 인스턴스가 포함되어 있으며 각 인스턴스에는 정확히 2 개의 입력으로 구성된 입력 시퀀스가 ​​있습니다.

마지막으로, Y0\_val과 Y1\_val은 모든 뉴런에 대한 시간 단계와 미니 배치의 모든 인스턴스에 대한 네트워크 출력을 포함합니다.

In [5] : 
In 
o. 
pr int ( YO_vaI ) 
[ [-0 .0664006 
[ 0.9977755 
[ 0.99999774 
[1. 
pr int ( Yl_val ) 
[-0.12210421 
0.99999827 
0.99928284 
0.96257669 
-0.71978903 
-o . 99898803 
-1. 
-1. 
0.62805271 
-o . 9999994 
-o . 99999815 
0.68105793 
-0.99657607 
-0.99999893 
-1. 
-1. 
0.96718431 
-0.9999975 
-0.99990582 
0.70918542 
0.96739239 
0.99677622 
-0.99818915 
0.40200251 
-0.99371219 
-0.85943311 
0.98579627 
-0.89821601] 
-0.99989718] 
-0.99999988] 
0.99950868]] 
_ 0.99999982] 
-0.25839362] 
-0.99998808] 
- 92205757]] 

그렇게 어려운 것은 아니지만 100 단계 이상의 RNN을 실행하려면 그래프가 상당히 커야합니다.

이제 TensorFlow의 RNN 작업을 사용하여 동일한 모델을 만드는 방법을 살펴 보겠습니다.

**2.1 Static Unrolling Through Time**

static\_rnn () 함수는 셀을 연결하여 전개되지 않은 RNN 네트워크를 만듭니다.

다음 코드는 이전 모델과 완전히 동일한 모델을 만듭니다.

In [8]: 
n inputs = 
3 
n neurons 
reset _ graph ( ) 
XO = tf .p1aceh01der(tf.f10at32, [None, n inputs]) 
Xl = tf.placeholder(tf. float32, [None, n _ inputs] ) 
basic _ cell = tf . contrib.rnn.BasicRNNCe11(num units=n neurons) 
output_seqs, states = tf . [XO, Xl] 
dtype=tf. float32 ) 
YO, Yl = output_segs 

* 입력 placeholder를 생성합니다.
* BasicRNNCell을 생성합니다.이 BasicRNNCell은 셀의 복사본을 생성하여 풀린 RNN을 생성하는 팩토리로 생각할 수 있습니다 (각 시간 단계마다 하나씩).
* static\_rnn ()을 호출하여 셀 팩토리와 입력 텐서를 제공하고 입력의 데이터 유형을 알려줍니다. static\_rnn () 함수는 입력마다 한 번씩 셀 팩토리의 \_\_call \_\_ () 함수를 호출하여 공유 가중치 및 바이어스 조건을 사용하여 셀의 복사본 두 개를 만들고 이전과 같이 연결합니다 .   
  static\_rnn () 함수는 두 개의 객체를 반환합니다. 첫 번째는 각 시간 단계에 대한 출력 텐서를 포함하는 Python 목록입니다. 두 번째는 네트워크의 최종 상태를 포함하는 텐서입니다. 기본 셀을 사용하는 경우 최종 상태는 단순히 마지막 출력과 같습니다.

50 단계의 시간 간격이 있다면 50 개의 입력 placeholder 및 50 개의 출력 텐서를 정의하는 것이 편리하지 않습니다. 또한 실행시 50 개의 각 placeholder에 피드를 제공하고 50 개의 출력을 조작해야 합니다.

이것을 단순화합시다.

In [15] : 
In [16]: 
n _ steps — 
inputs = 
3 
n 
n 
neurons 
reset_graph ( ) 
X = tf . placeholder(tf . float32, [None, n steps, n _ inputs] ) 
X_seqs = tf.unstack(tf. transpose(X, perm=[l, O, 2] ) ) 
basic cell 
. contrib. rnn.BasicRNNCe11 (num units=n neurons ) 
states = tf . contrib.rnn. static_rnn(basic_cell, X_seqs , 
output_seqs , 
dtype=tf. float32 ) 
outputs = tf . transpose(tf . perm=[l, O, 2]) 

다음 코드는 동일한 RNN을 다시 작성합니다.

* 이번에는 [None, n\_steps, n\_inputs] 모양의 단일 입력 placeholder를 취합니다. 여기서 첫 번째 차원은 미니 배치 크기입니다. 그런 다음 각 시간 단계에 대한 입력 시퀀스 목록을 추출합니다.
* X\_seqs는 n\_steps 텐서의 모양 [None, n\_inputs]의 파이썬 목록입니다. 여기서 첫 번째 차원은 미니 배치 크기입니다.  
  이렇게 하려면 먼저 transpose () 함수를 사용하여 처음 두 차원을 바꿔 시간 단계가 이제 첫 번째 차원이되도록합니다.  
  그런 다음 unstack () 함수를 사용하여 첫 번째 차원 (즉, 시간 단계 당 하나의 텐서)을 따라 파이썬 목록의 텐서를 추출합니다.
* 다음 두 줄은 이전과 같습니다.
* 마지막으로, stack () 함수를 사용하여 모든 출력 텐서를 하나의 텐서로 병합하고, 처음 두 개의 치수를 교환하여 [None, n\_steps, n\_neurons]의 최종 출력 텐서를 얻습니다 (다시 첫 번째 치수는 미니 - 배치 크기).

이제 모든 미니 배치 시퀀스가 ​​포함 된 단일 텐서를 공급하여 네트워크를 실행할 수 있습니다.

In [18] : 
X _ batch = np. array ( [ 
[16, 7 
[9, 
[0, 
[6, 
[3, 
8, 
o, 
5, 
2, 
# instance 1 
# instance 2 
instance 3 
# instance 4 
with tf. Session() as sess: 
init . run( ) 
outputs_val = outputs . eval ( 
X _ batch ) 

모든 인스턴스, 모든 시간 단계 및 모든 뉴런에 대해 하나의 outputs\_val 텐터를 얻습니다.

In 【 19 〕 : 
print(outputs Val) 
一 0 · 45 6 5 2 32 4 
-0 , 94 2 8 80 0 3 
【 【 一 0 · 80015349 
-0 , 63 711601 
【 【 一 0 · 93605185 
-0 , 916 53 8 6 
0 · 99 2 7 3 6 8 8 
-0 , 02 7 4 6 33 4 
一 0 . 6 8 0 64123 
- 0 , 9 9 9 8 8 6 87 
一 0 . 9 9 218 2 6 7 
0 , 113 0 0 9 3 2 
一 0 . 9 9 9 8 3 7 8 8 
- 0 , 9 9 4 5 6 0 4 2 
一 0 . 9 9 819 3 3 2 
- 0 , 7 3191994 
0 · 4 0 9 3 823 7 
0 , 94 0 5 5 814 
0 · 7 817 7 9 71 
0 , 5 7 9 8 4 3 7 
0 · 93 0 8 8 6 6 9 
0 , 8 9 6 0 5415 
一 0 · 55 5 4 3 64 3 
0 , 7 82 7 8 72 
0 . 6 3104504 
0 , 9 9 9 9 9 8 51 
0 . 9 9 710 31 
0 , 4 3105593 
0 . 9 9 9 9 8152 
0 , 9 9 9 8 719 7 
0 . 9 9 89 0 31 
0 , 9 52 5 6 817 
一 0 · 45732826 】 
-0 · 9999997 】 亅 
一 0 · 99646091 】 
-0 · 63716984 】 亅 
一 0 · 99998295 】 
-0 · 99997509 】 亅 
一 0 · 9953323 】 
-0 · 97817713 】 月 

그러나 이 방법은 여전히 ​​시간 단계마다 하나의 셀을 포함하는 그래프를 작성합니다. 50 단계의 시간 간격이 있다면, 그래프는 꽤보기 흉하게 보일 것입니다.

큰 그래프와 같은 경우, backpropagation 중에 OOM (out-of-memory) 오류가 발생할 수도 있습니다(특히 GPU 카드의 제한된 메모리로). 순방향 패스 중에는 모든 텐서 값을 저장해야하므로 역 통과 동안 그래디언트를 계산하는데 사용할 수 있습니다.

다행히도 더 나은 솔루션인 dynamic\_rnn () 함수가 있습니다.

**2.2 Dynamic Unrolling Through Time**

dynamic\_rnn () 함수는 while\_loop () 함수를 사용하여 셀을 적절한 횟수만큼 실행하고 OOM 오류를 피하기 위해 역전파 중 GPU의 메모리를 CPU 메모리로 스왑하려는 경우 swap\_memory = True로 설정할 수 있습니다. 편리하게도, 모든 시간 단계 (shape [None, n\_steps, n\_inputs])에서 모든 입력에 대해 단일 텐서를 받아들이고 모든 시간 단계 (shape [None, n\_steps, n\_neurons])에서 모든 출력에 대해 단일 텐서를 출력합니다. stack, unstack, 또는 transpose 할 필요가 없습니다.

다음 코드는 dynamic\_rnn () 함수를 사용하여 이전과 동일한 RNN을 만듭니다.

In [21]: 
In [22]: 
n _ steps 
n_inputs = 3 
n neurons 
reset _ graph ( ) 
X = tf. placeholder (tf. float32, [None, n_steps, n_inputs] ) 
basic_cell = tf . contrib. rnn. BasicRNNCe11 (num units=n_neurons) 
outputs, states = tf . X, dtype=tf. float32 ) 

backpropagation 중에 while\_loop () 연산은 뛰어나게 수행합니다. 정방향 패스 중에 각 반복에 대한 텐서 값을 저장하므로 역방향 패스 중 그라디언트를 계산할 때 사용할 수 있습니다.

**2.3 Handling Variable Length Input Sequences**

지금까지 고정 크기 입력 시퀀스 (모두 정확히 두 단계) 만 사용했습니다. 입력 시퀀스의 길이가 가변적인 경우? 이 경우에는 dynamic\_rnn () (또는 static\_rnn ()) 함수를 호출 할 때 sequence\_length 인수를 설정해야 합니다. 각 인스턴스에 대한 입력 시퀀스의 길이를 나타내는 1D 텐서 여야합니다.

In [27]: 
In [28]: 
n steps 
2 
n_inputs 
5 
n neurons = 
reset_graph ( ) 
X = tf . placeholder(tf . float32, [None, n steps, n _ inputs] ) 
basic_cell = tf . contrib. rnn.BasicRNNCe11(num units—n neurons ) 
seq_length = tf . placeholder (tf . int32, [None] ) 
outputs, states = tf (basic_cell, X, dtype=tf. float32, 
sequence_length=seq_length) 

예를 들어 두 번째 입력 시퀀스에 두 개가 아닌 하나의 입력만 있다고 가정합니다. 입력 텐서 X에 맞춰 지려면 0 벡터로 채워야합니다 (입력 텐서의 두 번째 크기는 가장 긴 시퀀스의 크기 즉, 2이므로).

In [30]: 
X_batch = np.array( [ 
# step O 
seq_length_batch = 
step 
1 
instance 
instance 
instance 
instance 
1 
2 (padded with zero vectors) 
3 
4 
np. array( [2, 
1, 

물론 이제 X 및 seq\_length placeholder에 대한 값을 모두 입력해야 합니다.

In [31] : 
with tf.session() as sess: 
init . run( ) 
outputs val, states val 
= sess . run( 
[ outputs, states] , 
atch} ) 
X_batch, seq_length : 

이제 RNN은 입력 시퀀스 길이를 지난 모든 시간 단계에 대해 0 벡터를 출력합니다 (두 번째 시간 단계의 두 번째 인스턴스 출력보기).

In [32] : 
o. 
o. 
o. 
o. 
o. 
print ( outputs_val ) 
o. 73155707 
[-0.61832315 
o. 8438704 
o. 91159219 
[-0.77545464 
[[-0.99952269 
[-0.62120003 
o. 34835717 
0.99999994 
o. 99697548 
o. 99999052 
o. 99987745 
o. 99977839 
o. 54000849 
0.50581998 
o. 86584806 
o. 84789461 
o. 95954454 
0.97847283 
0.59744656 
o. 96938425 
-o. 22882833 
o. 97933769 
o. 4303219 
o. 81892216 
o. 73172009 
o. 99309695 
o. 19893228 
-0.44022718] 
o. 9902246 ] ] 
0.09275495] 
0.57740247] 
0.888776 ] ] 
0.99847412] 
o. 1944804 ] ] ] 

또한 상태 텐서에는 각 셀의 최종 상태가 포함됩니다 (제로 벡터 제외).

In 
[33] : 
pr int ( states_val ) 
[ [-0.61832315 
0.8438704 
[-0.77545464 
[-0.62120003 
0.99999994 
0.99697548 
0.99987745 
0.54000849 
0.86584806 
0.84789461 
0.97847283 
0.96938425 
0.97933769 
0.4303219 
0.73172009 
0.19893228 
0.9902246 ] 
0.09275495] 
0.888776 ] 
0.1944804 ] ] 

**2.4 Handling Variable-Length Output Sequences**

출력 시퀀스에도 가변 길이가 있다면 어떻게 될까요?

각 시퀀스의 길이를 미리 알고있는 경우 (예 : 입력 시퀀스와 길이가 같음을 알고있는 경우) 위에서 설명한대로 sequence\_length 매개 변수를 설정할 수 있습니다. 불행히도, 일반적으로 이것은 불가능합니다.

예를 들어, 번역된 문장의 길이는 일반적으로 입력된 문장의 길이와 다릅니다. 이 경우, 가장 일반적인 해결책은 EOS 토큰 (end-of-sequence token)이라고 하는 특별한 출력을 정의하는 것입니다. EOS 이후의 모든 출력은 무시해야 합니다.

이제는 RNN 네트워크 (또는 더 정확하게는 시간이 지나면서 풀린 RNN 네트워크)를 만드는 방법을 알았습니다.

어떻게 훈련 시키는지 살펴 보겠습니다.

1. **Training RNNs**

RNN을 훈련 시키려면 시간을 거쳐 언롤링한 다음 규칙적인 backpropagate를 사용하면됩니다 (그림 14-5 참조).

이 전략은 시간에 따른 역전파 (backpropagation through time : BPTT)라고합니다.

(3)' 
w,b 
w,b 
(2)' 
w,b 
(4) 
w,b 
Figure 14-5. Backpropagation through time 

* 정기적인 역전파와 마찬가지로 전개되지 않은 네트워크를 통과하는 첫 번째 전달이 있습니다 (점선 화살표로 표시). 출력 시퀀스는 비용 함수를 사용하여 평가됩니다. (여기서 tmin 및 tmax는 무시된 출력을 계산하지 않은 첫 번째 및 마지막 출력 시간 단계임)
* 그 비용 함수의 기울기는 전개되지 않은 네트워크 (실선 화살표로 표시됨)를 통해 역방향으로 전파됩니다.
* 마지막으로 모델 파라미터는 BPTT 동안 계산된 기울기를 사용하여 업데이트됩니다.

그라디언트는 최종 출력뿐만 아니라 비용 함수에서 사용되는 모든 출력을 거쳐 후방으로 흐릅니다.

그림 14-5에서 비용 함수는 네트워크의 마지막 세 출력인 Y (2), Y (3) 및 Y (4)를 사용하여 계산되므로 그래디언트가 이 세 개의 출력을 통과합니다. Y (0) 및 Y (1)는 통하지 않습니다.

더욱이, 동일한 파라미터 W 및 b가 각 시간 단계에서 사용되기 때문에, 역전파는 작동을 잘 하고 모든 시간 단계에 걸쳐 합계합니다.

**3.1 Training a Sequence Classifier**

MNIST 이미지를 분류하기 위해 RNN을 교육합시다. CNN은 이미지 분류에 더 적합하지만, 이는 이미 익숙한 간단한 예를 만듭니다. 우리는 이미지를 각각 28 픽셀의 28 행의 시퀀스로 취급할 것입니다 (각 MNIST 이미지는 28 × 28 픽셀이기 때문에). 우리는 150 개의 뉴런 셀과 마지막 시간 단계의 출력에 연결된 10 개의 뉴런 (클래스 당 하나)을 포함하는 완전히 연결된 레이어와 softmax 레이어 (그림 14-6 참조)를 사용합니다.

Softmax 
Fully Connected 
10 units 
X(26) X(27) 
Figure 14-6. Sequence classifier 

구축 단계는 매우 간단합니다. 언롤된 RNN이 숨겨진 레이어를 대체한다는 점을 제외하고는 10 장에서 작성한 MNIST 분류기와 거의 같습니다. 완전히 연결된 레이어는 RNN의 최종 상태 (즉, 28 번째 출력) 만 포함하는 상태 텐서에 연결됩니다. 또한 y는 대상 클래스의 자리 표시 자입니다.

In [34]: 
28 
n steps = 
n inputs 
= 150 
n neurons 
n outputs — 10 
learning_rate = O. 001 
X = tf.placeholder (tf. float32, [None, n _ steps, n_inputs] ) 
y = tf.p1aceh01der(tf.int32, [None] ) 
basic_ce11 = tf. contrib.rnn.BasicRNNCe11(num 
outputs, states = tf .nn.dynamic rnn (basic_cell, X, dtype=tf . float32) 
logits = tf. layers . dense( states, n_outputs ) 
xentropy = tf .nn.sparse_softmax cross_entropy_with logits ( labels=y, 
10gitS=10gitS ) 
loss = tf. xentropy) 
optimizer = tf 
training_op = optimizer .minimize( loss) 
correct = tf logits, y, 1) 
accuracy = tf . cast (correct, tf. float32) ) 
init = tf. ) 

이제 MNIST 데이터를 로드하고 네트워크에서 예상하는대로 [batch\_size, n\_steps, n\_inputs]에 테스트 데이터를 다시 만들어 봅시다. 우리는 잠시 동안 훈련 데이터를 재구성 할 것입니다.

In [35]: 
from tensorflow.examples . tutorials .mnist import input data 
mnist = input_data.read data sets( " /tmp/data/" ) 
X_test = mnist. test. images. reshape( (—1, n_steps, n_inputs) ) 
y _ test = mnist. test. labels 
Extracting 
Extracting 
Extracting 
Extracting 
/ tmp/ data/ g z 
/ tmp/ data/ g z 
/ / data/ t 1 Ok— images — idx3 —ubyte. g z 
/ tmp/ data/ t 1 —ubyte . g z 

이제 우리는 RNN을 훈련할 준비가 되었습니다. 실행 단계는 10 장의 MNIST 분류기와 완전히 동일하지만, 네트워크에 공급하기 전에 각 교육 배치를 재구성합니다. 출력은 다음과 같아야합니다.

In 
[36]: 
n_epochs 
= 100 
batch size = 150 
with tf.Session() as sess: 
init. run( ) 
for 
est) 
epoch in range (n _ epochs) : 
for iteration in range(mnist.train.num examples // batch_size) : 
X _ batch, y_batch = mnist. train. 
X batch = X batch. reshape( (—1, n steps, n_inputs) ) 
sess.run (training_op, X _ batch, 
acc_train = accuracy. eval ( X _ batch, 
acc test = accuracy. eval (feed X test, y: 
print ( epoch, "Train accuracy: " , 
acc train, "Test 
Y : y _ batch} ) 
Y : y_batch} ) 
y _ test} ) 
acc t 
accuracy: 
1 
2 
3 
4 
5 
Train 
Train 
Train 
Train 
Train 
Tra n 
accuracy : 
accuracy : 
accuracy : 
accuracy : 
accuracy : 
acenracv 
0.946667 Test accuracy: 0.9366 
0.966667 Test accuracy: 0.9488 
0.96 Test accuracy: 0.9614 
0.966667 Test accuracy: 0.9572 
0.946667 Test accuracy: 0.9624 
0.96 Tegt-_ acenracv: O - 9634 

:

95 
96 
97 
98 
99 
Train 
Train 
Train 
Train 
Train 
accuracy : 
accuracy : 
accuracy : 
accuracy : 
accuracy : 
1.0 Test 
1.0 Test 
0.993333 
0.993333 
1.0 Test 
accuracy: 0.9771 
accuracy: O. 9814 
Test accuracy: 0.9673 
Test accuracy: 0.9788 
accuracy: O. 9792 

우리는 98 % 이상의 정확도를 얻었습니다.

또한 하이퍼 매개 변수를 조정하고 초기화 (He 초기화), 더 길게 트레이닝하거나 정당화 (예 : 드롭 아웃)를 추가하여 RNN 가중치를 초기화하면 더 나은 결과를 얻을 수 있습니다.

[Note}

[RNN의 초기화 코드는 가변 스코프 (예 : variable\_scope ( "rnn", initializer = variance\_scaling\_initializer ())를 사용하여 해당 초기화 코드를 사용하여 구성 코드를 래핑하여 지정할 수 있습니다.]

**3.2 Training to Predict Time Series**

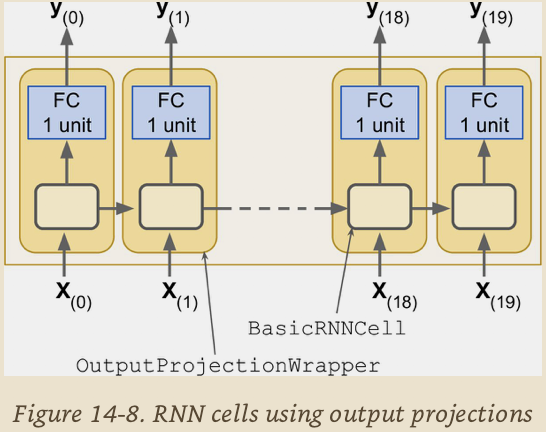
이제 주식 가격, 기온, 뇌파 패턴 등과 같은 time series를 처리하는 방법을 살펴 보겠습니다. 이 섹션에서는 생성된 time series에서 다음 값을 예측하기 위해 RNN을 학습합니다. 각 트레이닝 인스턴스는 time series에서 20 개의 연속 값의 무작위로 선택된 시퀀스이며, 대상 시퀀스는 입력 시퀀스와 동일하지만 한 단계씩 앞으로 이동해야합니다 (그림 14-7 참조).

In [42]: 
t = t_max, int( (t_max — t_min) / resolution) ) 
n steps = 20 
t instance = 
np.linspace( 12.2, 12.2 + resolution * (n _ steps + 1), 
plt. figure( 11, 4) ) 
pit. subplot ( 121 ) 
plt. title( "A time series (generated) " , 
fontsize=14) 
n_steps 
plt.plot(t, time series(t), label=r"$t . \sin(t) / 3 + 2 
. \sin(5t)$") 
pit. plot , : 
training instance " ) 
plt. legend( loc= " lower 
left" , 
fontsize=14) 
30, 
-17, 
13]) 
plt . xlabel( "Time" ) 
plt.ylabel( "Va1ue " ) 
plt. subplot ( 122 ) 
plt.title( "A training 
instance " 
fontsize=14 ) 
O label=" instance" ) 
plt.plot(t instance[l:], time series(t_instance[l: 
label= target " ) 
plt. legend ( loc= " upper 
left " ) 
plt.xlabel( "Time " ) 
save fig( "time series_plot" ) 
plt. show( ) 
Saving figure time series_plot 
A time series (generated) 
linewidth=3 , 
¯1]) , "bon , markersize=l 
] ) , "w* markersize=10, 
A training instance 
10 
5 
-10 
-15 
o 
t. +2. sin(5t) 
A training instance 
6 
4 
2 
—2 
25 
instance 
target 
0 
0 
0 
12.5 
0 
5 
10 
15 
Time 
20 
30 
13.0 
0 
Time 
13.5 
14.0 

먼저 RNN을 만듭니다. 그것은 100 개의 반복적인 뉴런을 포함할 것이고, 각 훈련 인스턴스는 20 개의 입력이 될 것이므로 20 개의 시간 간격으로 그것을 풀 것입니다. 각 입력에는 하나의 해당 feature만 포함됩니다. 목표는 또한 각각 하나의 값을 포함하는 20 개의 입력 시퀀스입니다. 코드는 이전과 거의 같습니다.

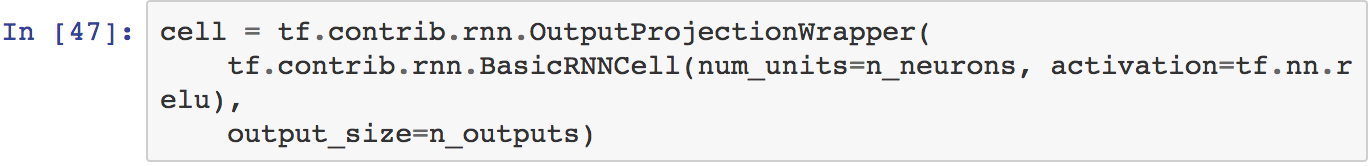
In [45]: 
reset_graph ( ) 
n 
n 
n 
n 
steps 
inputs 
100 
neurons = 
outputs = 1 
tf. placeholder ( tf. float32, [None, n_steps, n _ inputs] ) 
tf. placeholder (tf. float32, [None, n_steps, n_outputs] ) 
cell = tf . contrib.rnn.BasicRNNCe11 (num units=n_neurons, activation=tf. nn.r 
elu) 
outputs, states = X, dtype=tf .f10at32) 

각 시간 단계마다 크기가 100 인 출력 벡터가 생성됩니다. 그러나 실제로 원하는 것은 각 시간 단계에서 단일 출력 값입니다. 가장 간단한 해결책은 OutputProjectionWrapper에 셀을 래핑하는 것입니다. 셀 wrapper는 기본 셀처럼 작동하여 기본 셀에 대한 모든 메서드 호출을 프록싱 하지만 몇 가지 기능을 추가합니다. OutputProjectionWrapper는 각 출력 위에 활성화 기능이없는 선형 뉴런에 완전히 연결된 레이어를 추가합니다. 이 모든 완전히 연결된 레이어는 동일한 (훈련 가능한) 가중치 및 바이어스 조건을 공유합니다. 결과 RNN은 그림 14-8에 나와 있습니다.



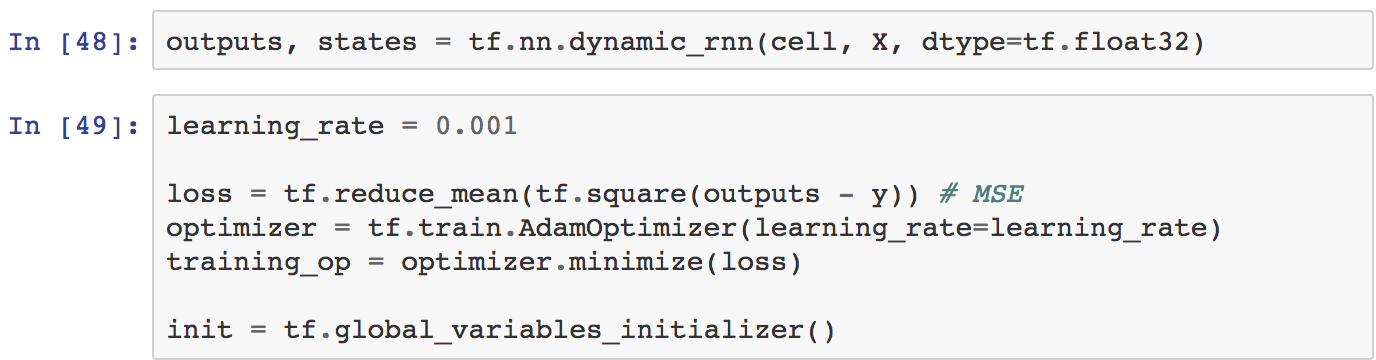
셀을 감싸는 것은 아주 쉽습니다.

BasicRNNCell을 OutputProjectionWrapper에 래핑하여 앞의 코드를 조정 해봅니다.

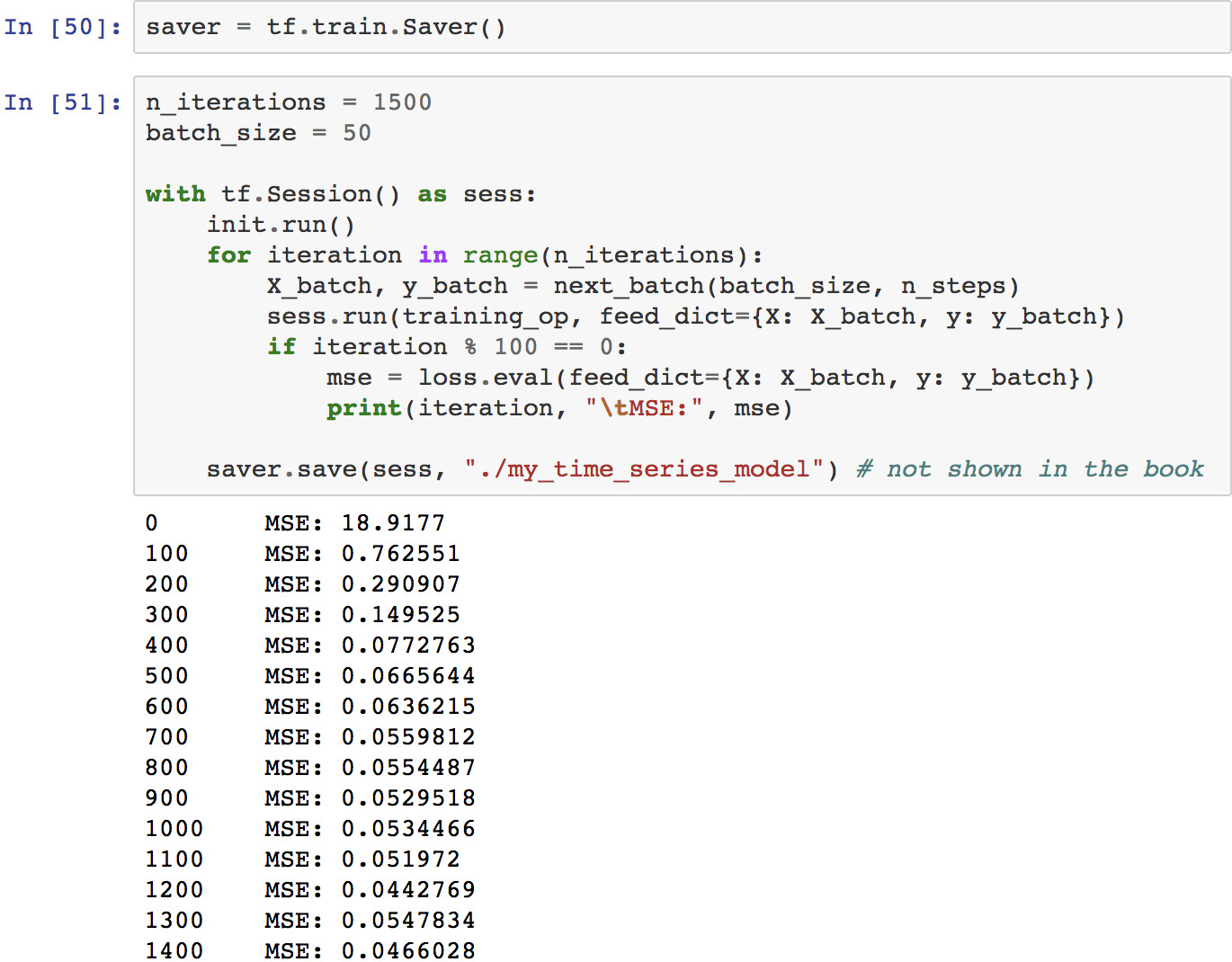


여태까지는 그런대로 잘됐다. 이제 비용 함수를 정의해야합니다. 이전의 회귀 작업에서와 마찬가지로 평균제곱오류(MSE)를 사용합니다.

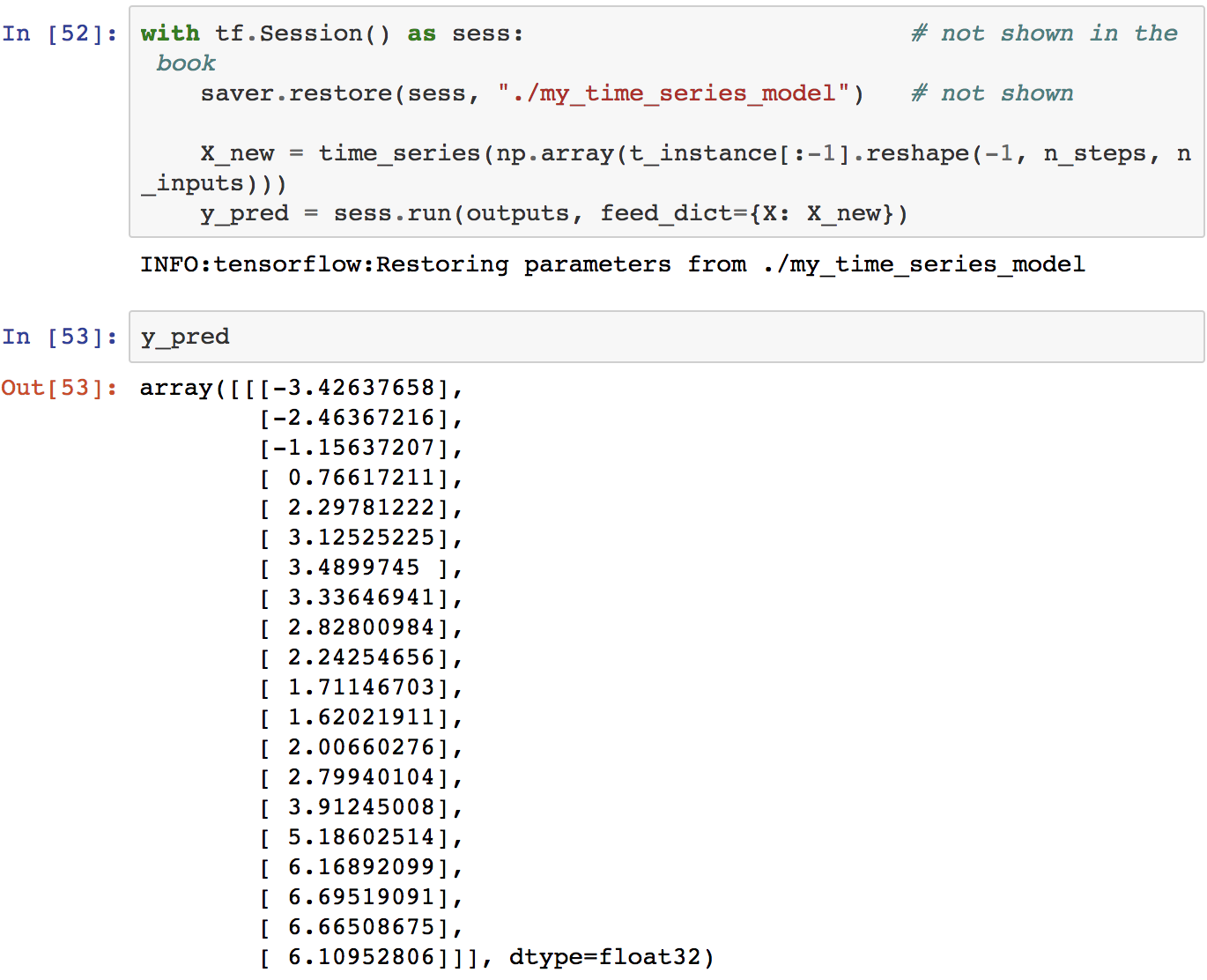
다음으로 평소처럼 Adam 최적화 도구, 교육 op 및 변수 초기화 op를 만듭니다.



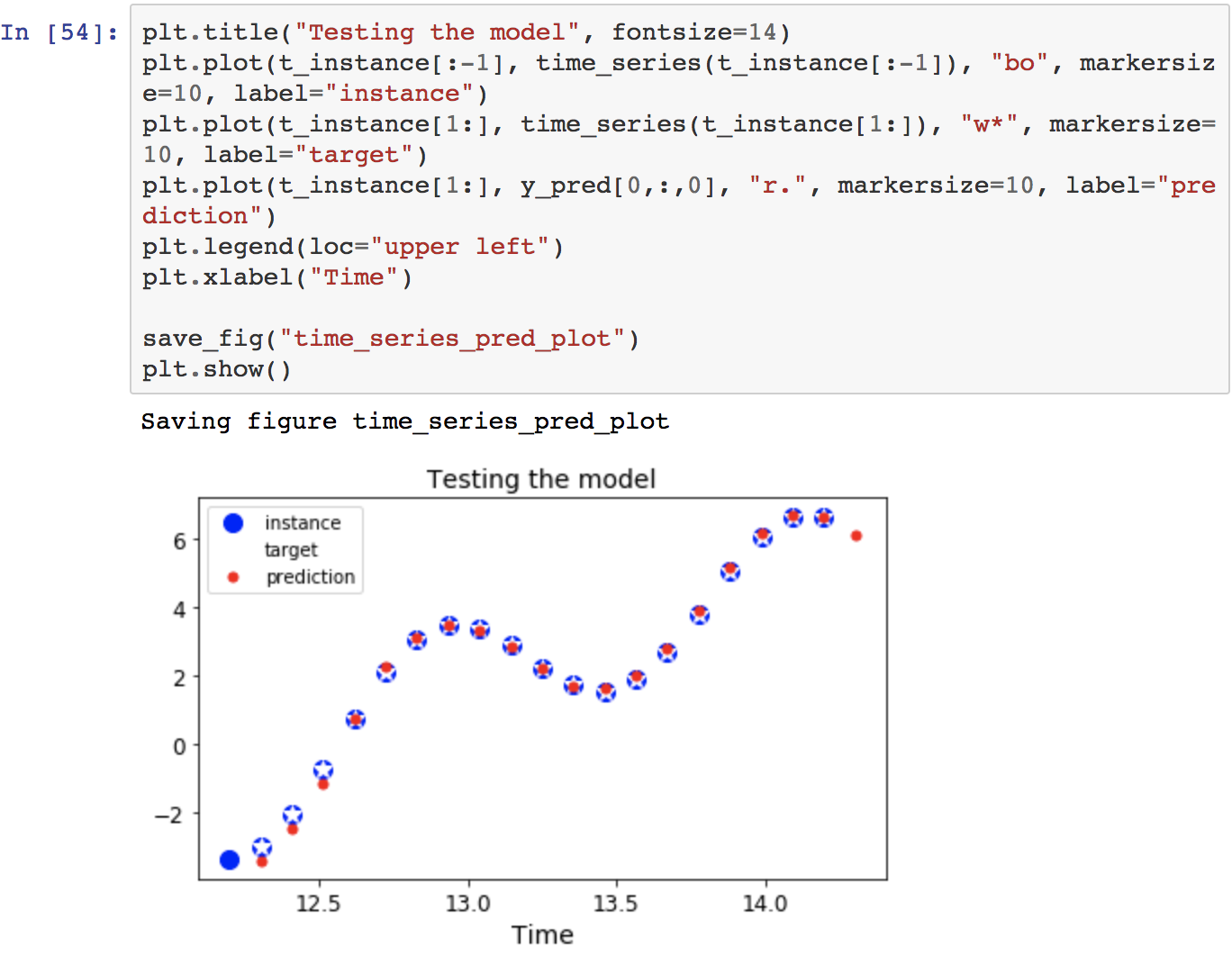
이제 실행 단계로 넘어갑니다.프로그램의 출력은 다음과 같아야합니다.



모델이 훈련되면 예측을 할 수 있습니다.



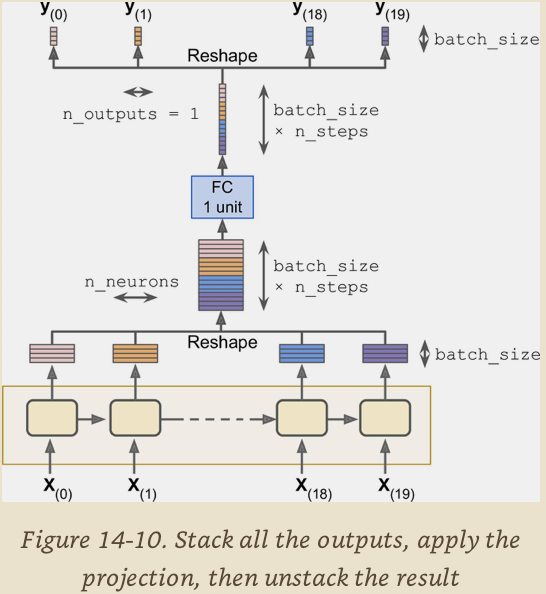
아래 그림 14-9는 단지 1,000 회 반복 한 후에 앞에서 살펴본 인스턴스 (그림 14-7)의 예상 시퀀스를 보여줍니다.



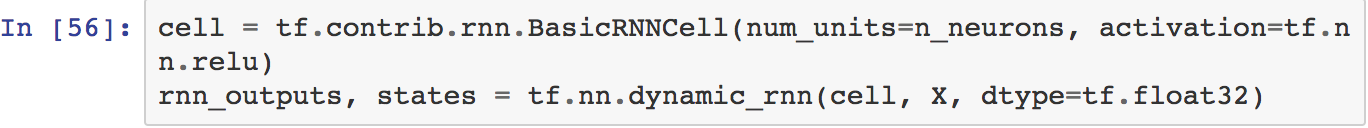
OutputProjectionWrapper를 사용하는 것이 RNN의 출력 시퀀스의 차원을 시간 단계 하나의 값으로 줄이는 가장 간단한 솔루션이지만 가장 효율적이지는 않습니다. 더 까다롭지만 보다 효율적인 솔루션이 있습니다.

[batch\_size, n\_steps, n\_neurons]에서 [batch\_size \* n\_steps, n\_neurons] 까지의 RNN출력을 재구성한 다음 적절한 출력 크기를 가진 단일 연결레이어를 적용할 수 있습니다. 이는 결과 텐서 형태의 [batch\_size \* n\_steps, n\_outputs]가 될 것이고 이 텐서를 [batch\_size, n\_steps, n\_outputs]로 바꿀 것입니다.

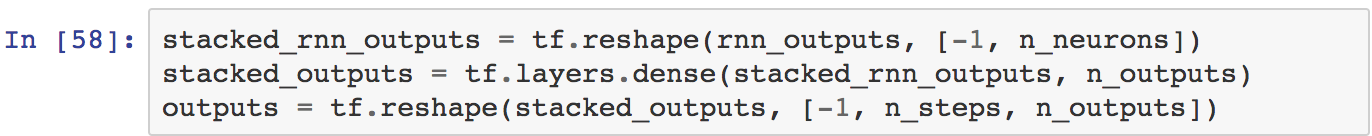
이러한 작업은 그림 14-10에 나와 있습니다.



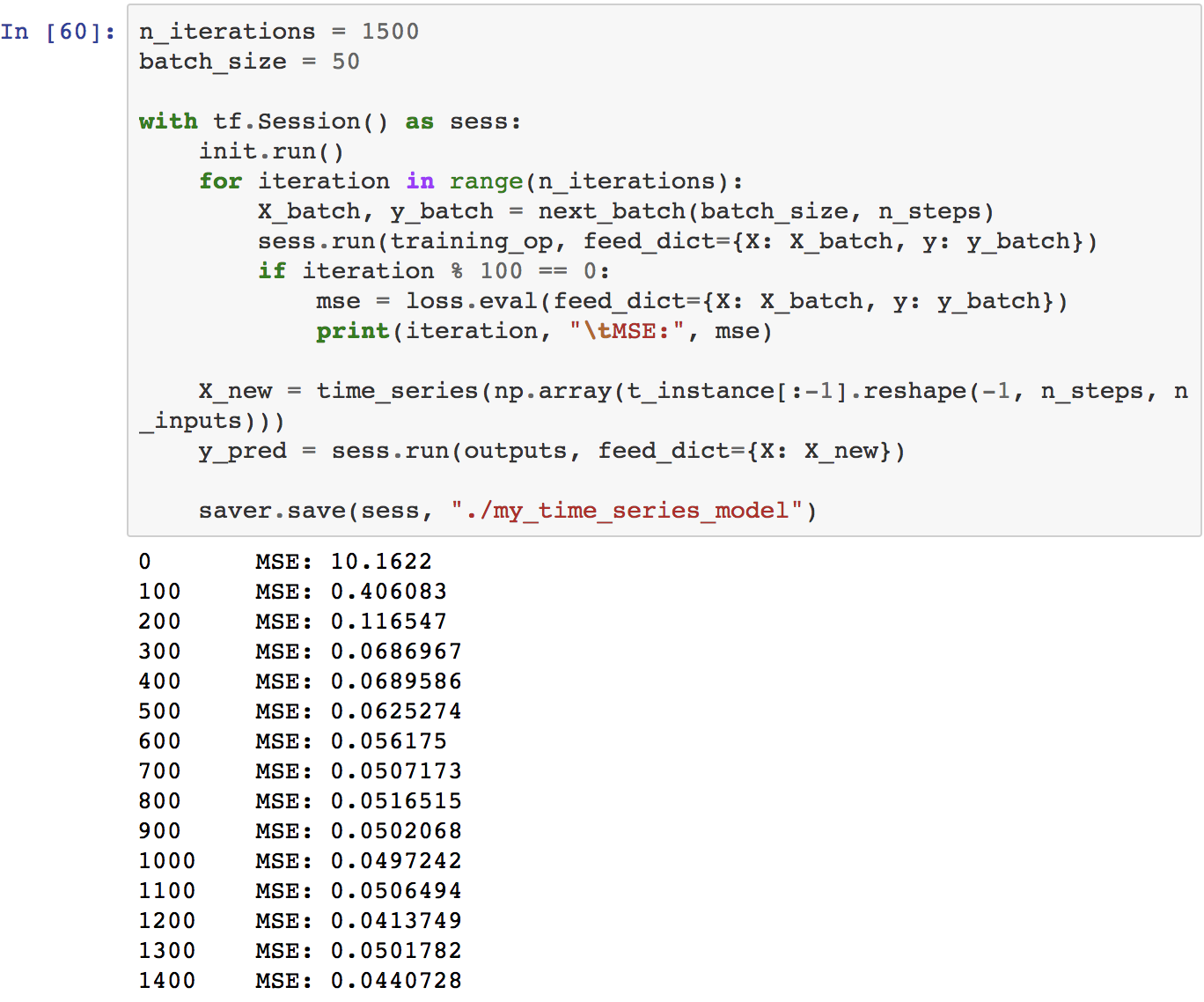
이 솔루션을 구현하려면 먼저 OutputProjectionWrapper없이 기본 셀로 되돌아갑니다.

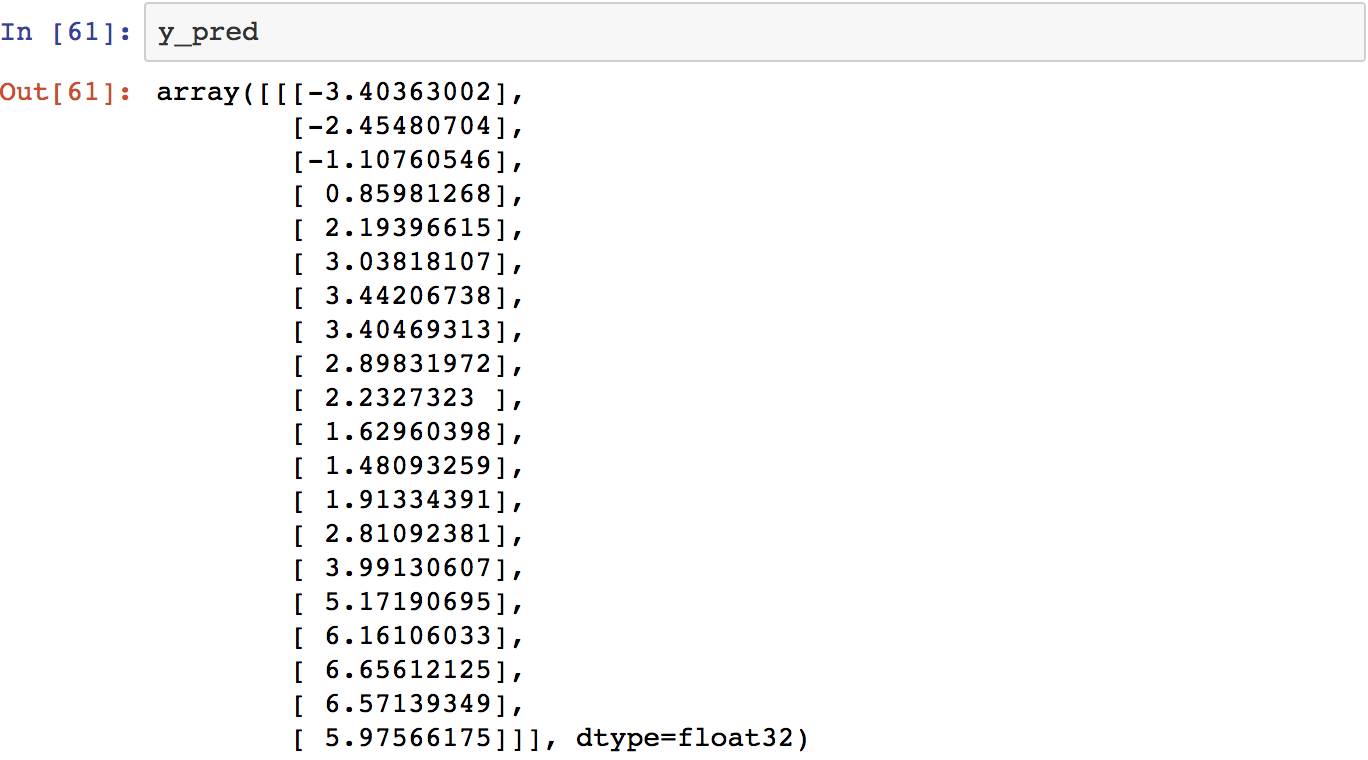


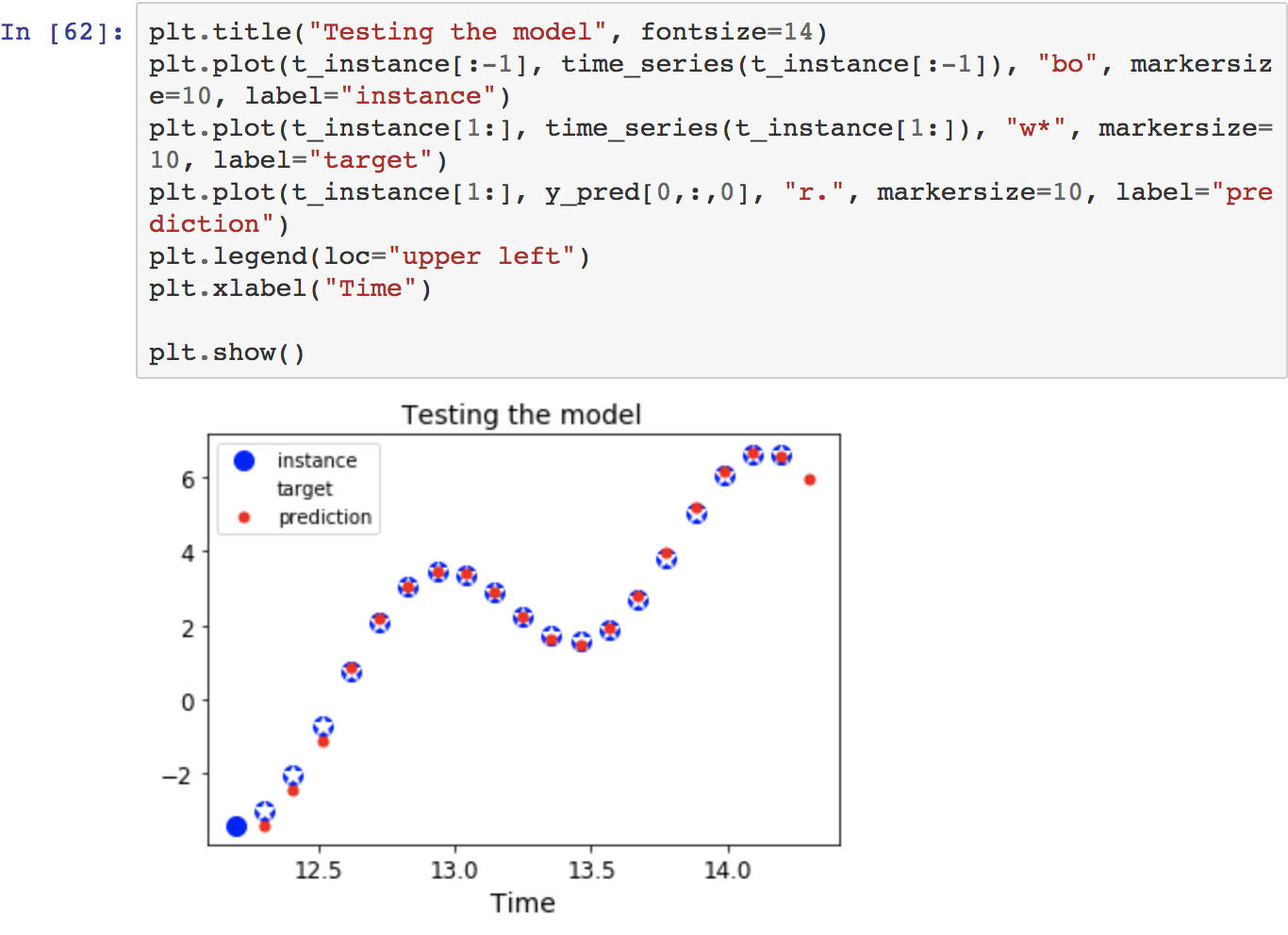
그런 다음 reshape () 연산을 사용하여 모든 출력을 스택에 쌓고, 모든 연결 함수를 사용하지 않고 완전히 연결된 선형 레이어를 적용한 다음 reshape ()를 사용하여 모든 출력을 최종적으로 언 스택합니다.



나머지 코드는 이전과 동일합니다.





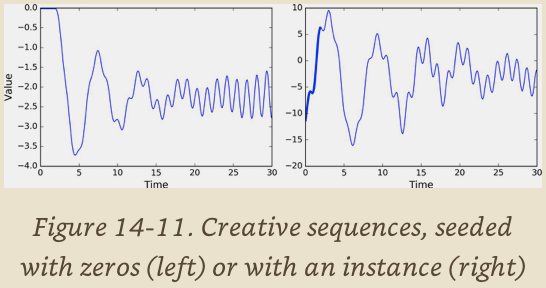


이것은 시간 단계마다 하나가 아닌 완전히 연결된 레이어가 있기 때문에 상당한 속도 향상을 제공 할 수 있습니다.

**3.3 Creative RNN**

 미래를 예측할 수있는 모델을 만들었으므로 이 장의 처음 부분에서 설명한대로 이를 사용하여 창의적인 시퀀스를 생성할 수 있습니다. 우리가 필요로하는 것은 n\_steps 값 (예 : 0으로 채움)을 포함하는 시드 시퀀스를 제공하고, 다음 값을 예측하기 위해 모델을 사용하고, 이 예측 값을 시퀀스에 추가하고, 마지막 n\_steps 값을 모델에 공급하여 다음 값 등등.

이 프로세스는 원래의 time series와 닮은 새로운 시퀀스를 생성합니다 (그림 14-11 참조).

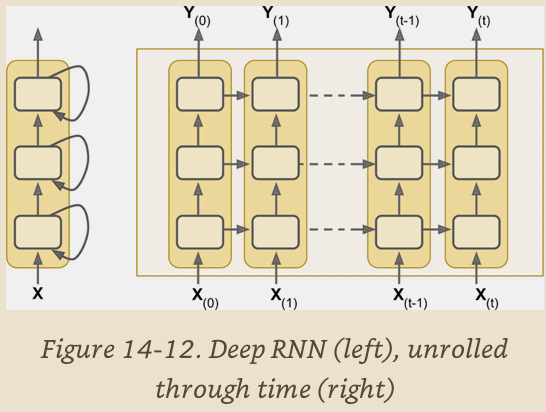


더 강력한 RNN이 필요하며 더 많은 뉴런이 있고 훨씬 더 깊을 수도 있습니다.

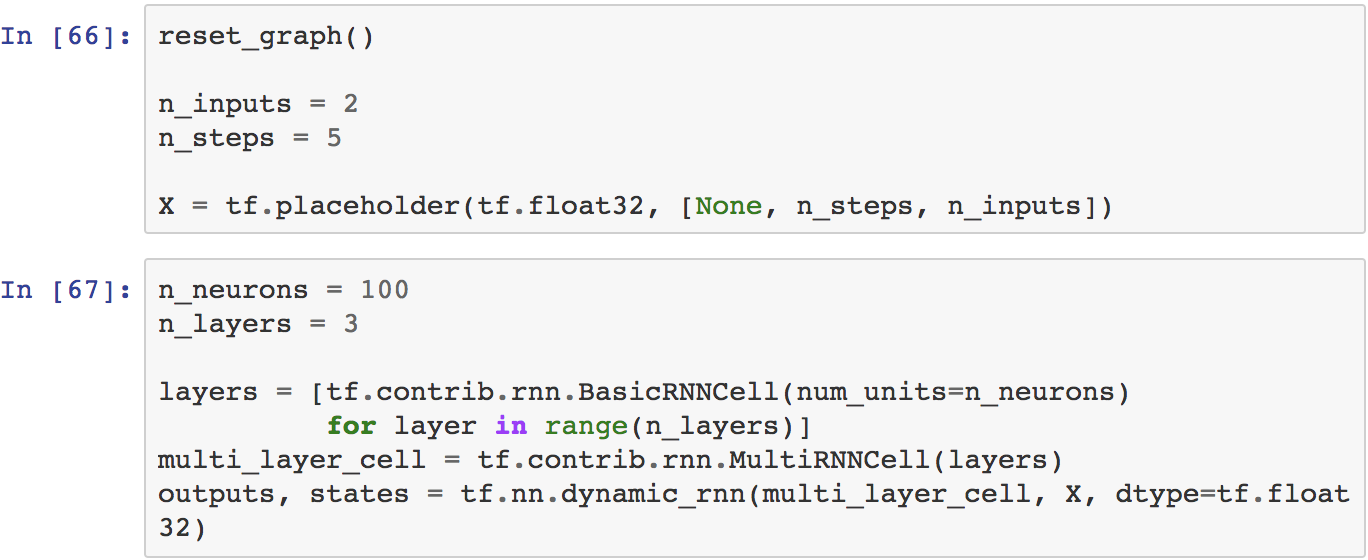
이제 깊은 RNN을 살펴 보겠습니다.

1. **Deep RNNs**

 그림 14-12와 같이 여러 개의 셀 레이어를 쌓는 것이 일반적입니다. 이것은 당신에게 깊은 RNN을 제공합니다. TensorFlow에서 깊은 RNN을 구현하려면 여러 셀을 만들어 MultiRNNCell에 스택으로 묶을 수 있습니다.



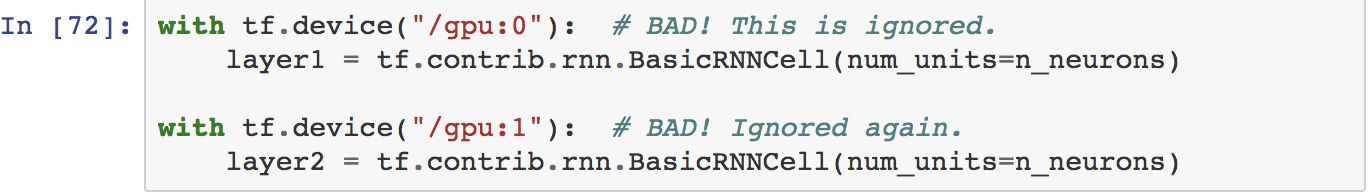
다음 코드에서는 3 개의 동일한 셀을 쌓습니다 (하지만 다른 수의 뉴런을 사용하여 다양한 종류의 셀을 사용할 수 있습니다).



그게 전부입니다! states 변수는 레이어당 하나의 텐서를 포함하는 튜플이며, 각 레이어의 셀의 최종 상태를 나타냅니다 (shape [batch\_size, n\_neurons]). MultiRNNCell을 만들 때 state\_is\_tuple = False로 설정하면 상태는 열 축을 따라 연결된 모든 레이어의 상태를 포함하는 단일 텐서가 됩니다 (즉, 모양은 [batch\_size, n\_layers \* n\_neurons] ).

**4.1 Distributing a Deep RNN Across Multiple GPUs**

 12 장에서는 각 계층을 다른 GPU에 고정하여 여러 GPU에 걸쳐 심층 RNN을 효율적으로 배포 할 수 있다고 지적했습니다 (그림 12-16 참조). 그러나 다른 장치 () 블록에서 각 셀을 만들려고해도 작동하지 않습니다.

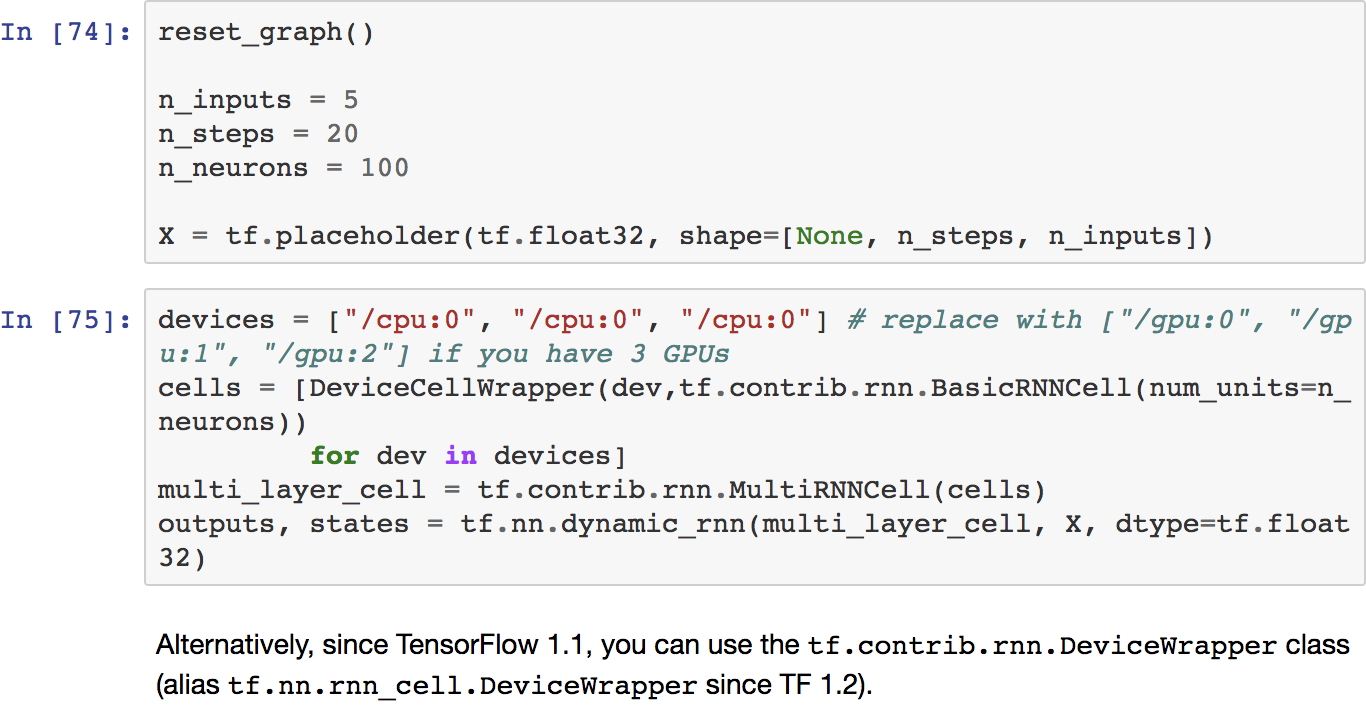


이것은 BasicRNNCell이 셀 자체가 아니기 때문에 실패합니다. 팩토리를 생성할 때 셀이 생성되지 않으므로 아무런 변수도 생성되지 않습니다. 장치 블록은 단순히 무시됩니다. 세포는 실제로 나중에 생성됩니다. dynamic\_rnn ()을 호출하면 MultiRNNCell이 호출되어 각 개별 BasicRNNCell을 호출하여 해당 변수를 포함하는 실제 셀을 만듭니다. 하지만 이 클래스들 중 어떤 것도 변수가 생성된 장치를 제어하는 ​​방법을 제공하지 못합니다. 장치 블록 내에 dynamic\_rnn () 호출을 넣으려고 하면 전체 RNN이 단일 장치에 고정됩니다. 고정 안되게하는 방법은 자신만의 셀 wrapper를 작성하거나 TensorFlow 1.1에 추가 된 tf.contrib.rnn.DeviceWrapper 클래스를 사용하는 것입니다.



이 wrapper는 장치 블록 내에서 \_\_call \_\_ () 함수를 래핑하는 것을 제외하고는 모든 메서드 호출을 다른 셀로 단순히 프록시합니다.

이제 각 레이어를 다른 GPU에 배포 할 수 있습니다.



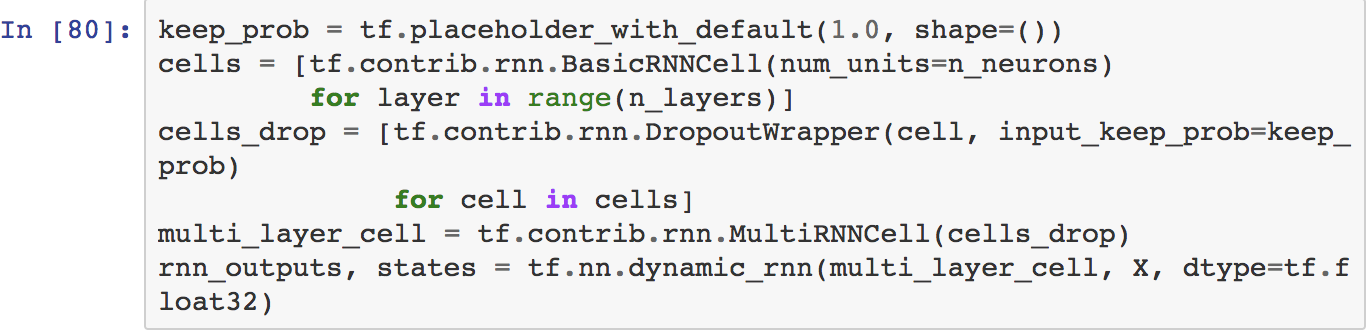
[warning]

[state\_is\_tuple = False로 설정하지 마십시오. 그렇지 않으면 MultiRNNCell은 단일 GPU에서 모든 셀 상태를 단일 텐서로 연결합니다.]

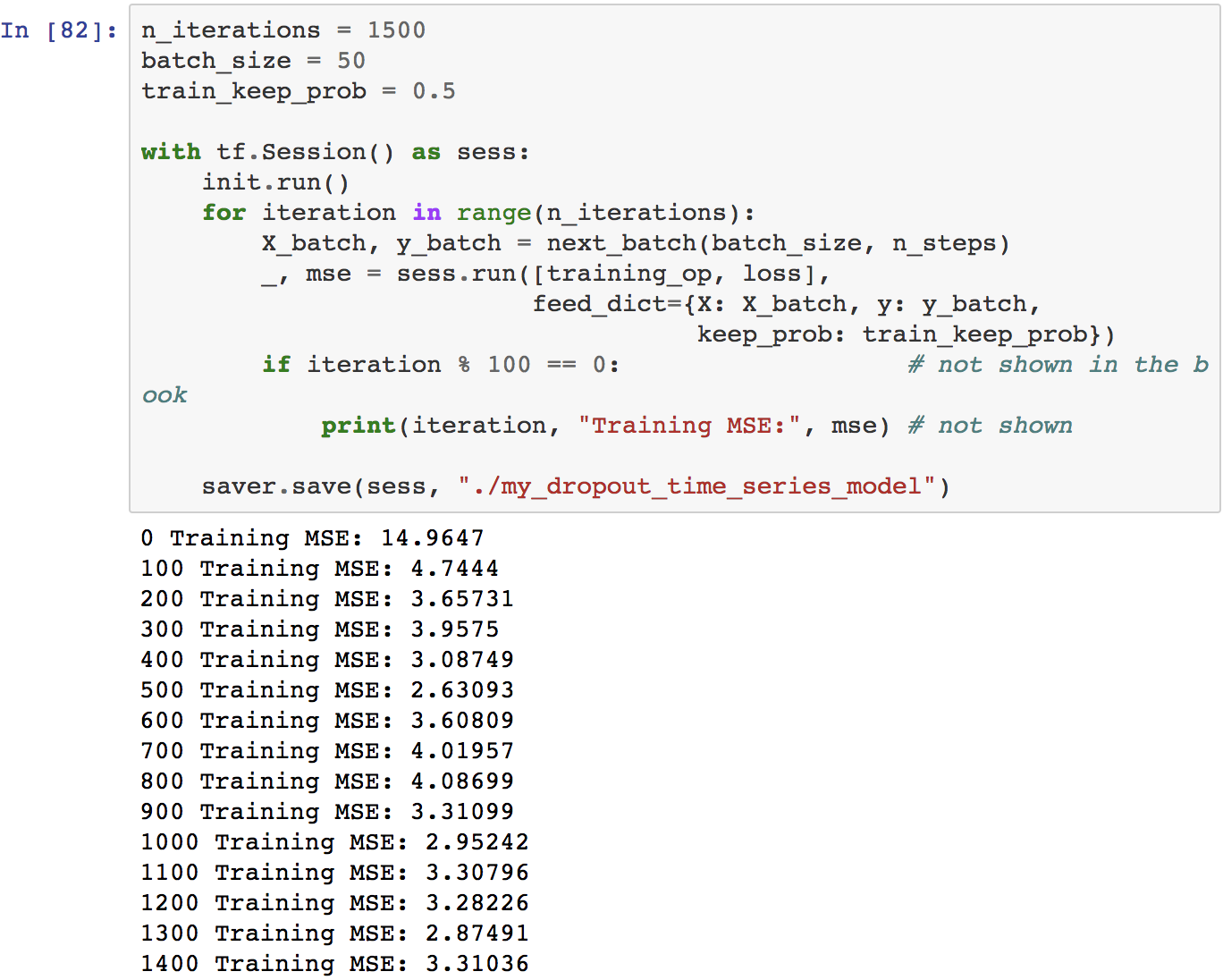
**4.2 Applying Dropout**

 매우 깊은 RNN을 구축하면 교육 세트에 지나치게 적합할 수 있습니다. 이를 막기 위해 일반적인 기술은 dropout을 적용하는 것입니다 (11 장에서 소개 함). 평소처럼 RNN 앞뒤에 드롭 아웃 레이어를 간단하게 추가 할 수 있지만 RNN 레이어간에 드롭 아웃을 적용하려는 경우 DropoutWrapper를 사용해야합니다.

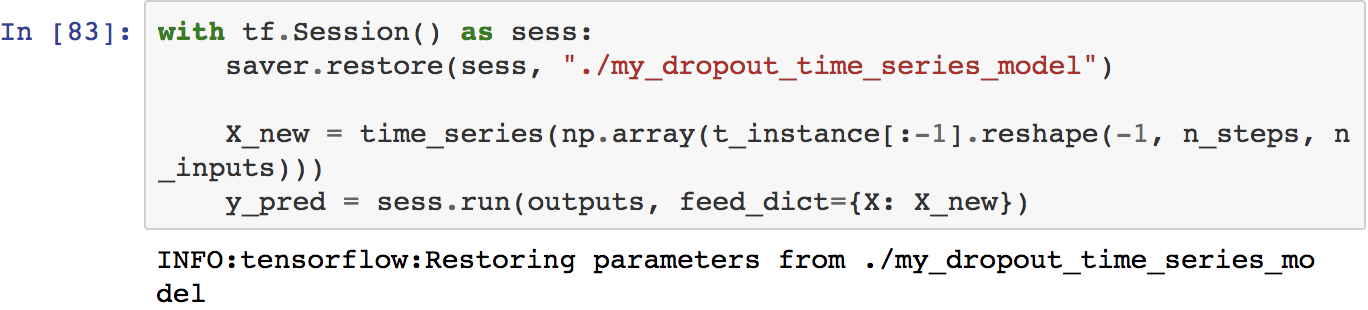
다음 코드는 RNN의 각 레이어 입력에 드롭 아웃을 적용합니다.

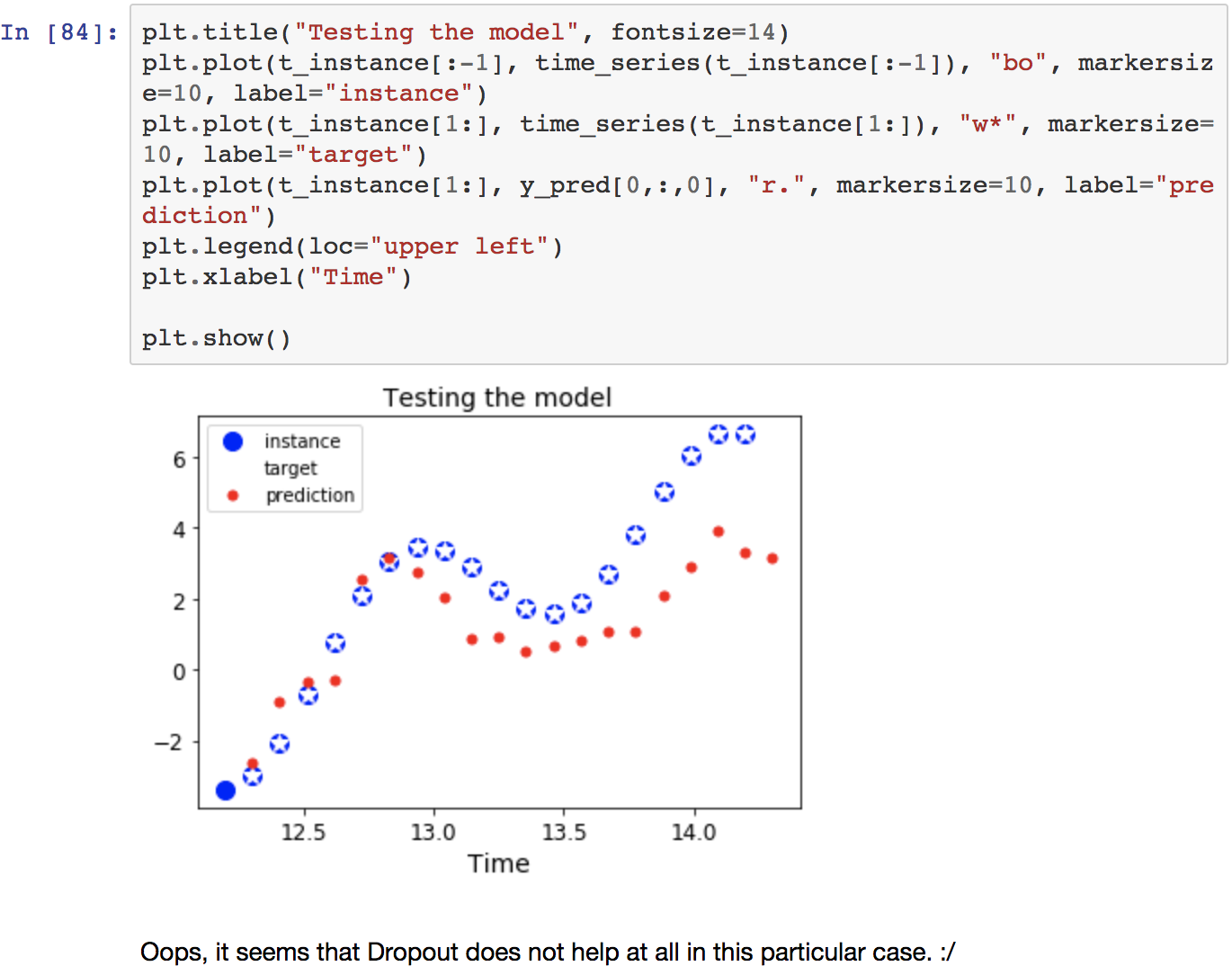


교육을하는 동안 원하는 값을 keep\_prob 자리 표시 자 (일반적으로 0.5)에 제공 할 수 있습니다.



테스트하는 동안 keep\_prob의 기본값을 1.0으로 설정하여 드롭 아웃을 효과적으로 해제해야합니다 (교육 중에 만 활성 상태 여야 함을 기억하십시오).





output\_keep\_prob을 설정하여 출력에 드롭아웃을 적용할 수도 있고 TensorFlow 1.1 이후 state\_keep\_prob를 사용하여 셀 상태에 드롭아웃을 적용할 수도 있습니다. 그걸로 모든 종류의 RNN을 훈련 할 수 있어야합니다! 하지만 긴 시퀀스에 대해 RNN을 훈련 시키려한다면, 상황이 조금 더 어려워 질 것입니다. 왜 그리고 어떻게 할 수 있는지 보도록하겠습니다.

**4.3 The Difficulty of Training over Many Time Steps**

긴 시퀀스에서 RNN을 훈련 시키려면 많은 시간 간격으로 RNN을 실행해야합니다. 이렇게 하면 펼쳐진 RNN이 매우 깊은 네트워크가 됩니다. 어떤 DNN과 마찬가지로 그것은 vanishing/ exploding 그라디언트 문제로 고생하고 훈련에 영원히 오래 걸릴 수 있습니다.

이 문제를 해결하기 위해 우리가 논의한 많은 트릭은 매개변수 초기화, ReLU, batch정규화, 그라디언트 클리핑 및 더 빠른 최적화와 같이 전개되지 않은 RNN에도 사용할 수 있습니다. 그러나, RNN이 적당히 긴 시퀀스(예를 들어, 100 개의 입력)를 처리할 필요가 있다면, 트레이닝은 여전히 ​​매우 느릴 것이다.

이 문제에 대한 가장 간단하고 가장 일반적인 해결책은 트레이닝 중에 제한된 수의 시간 단계에만 RNN을 언롤링하는 것입니다. 이를 시간 경과에 따른 잘린 backpropagation이라고합니다. TensorFlow에서는 입력 시퀀스를 잘라내어 간단하게 구현할 수 있습니다. 예를 들어, time series 예측 문제에서 훈련 중 n\_steps를 줄이면됩니다. 문제는 모델이 장기적인 패턴을 배울 수 없다는 것입니다.

한 가지 해결 방법은 단축된 시퀀스에 이전 데이터와 최근 데이터가 모두 포함되어 있는지 확인하여 모델에서 두 데이터를 모두 사용할 수 있도록 하는 것입니다.(예 : 시퀀스에는 지난 5 개월 동안의 월별 데이터, 지난 5 주간의 주간 데이터, 마지막 5 일 간의 일일 데이터가 포함될 수 있음).

하지만이 해결 방법에는 한계가 있습니다. 작년의 세밀한 데이터가 실제로 유용하다면 어떨까요?

몇 년 후에도 절대적으로 고려해야만하는 짧지만 중요한 사건이 있었다면 어떨까요? (예 : 선거 결과)

오랜 트레이닝 시간 외에도 장기 실행 RNN이 직면한 또 다른 문제는 첫 번째 입력의 메모리가 점차 사라지게 된다는 사실입니다.

실제로 RNN을 통과할 때 데이터가 통과하는 변환으로인해 시간 간격마다 일부 정보가 손실됩니다.

잠시 후, RNN의 상태는 사실상 첫 번째 입력의 흔적을 포함하지 않습니다.

예를 들어 "I loved this movie." 라는 네 단어로 시작하는 긴 검토에 대해 정서 분석을 수행하려고 한다고 가정 해보십시오.리뷰의 나머지 부분은 영화를 더욱 좋게 만들 수 있는 많은 것들을 나열합니다. RNN이 처음 네 단어를 점차 잊어 버리면 리뷰가 완전히 오판됩니다.

이 문제를 해결하기 위해 장기간 메모리를 사용하는 다양한 유형의 셀이 도입되었습니다. 그들은 너무 성공적이어서 기본 세포가 더 이상 사용되지 않는다는 것을 증명했습니다.

먼저 이러한 긴 메모리 셀 (LSTM 셀) 중에서 가장 많이 사용되는 셀을 살펴 보겠습니다.

1. **LSTM Cell**

 LSTM (Long Short-Term Memory) 셀은 1997 년 Sepp Hochreiter와 Jürgen Schmidhuber에 의해 제안되었으며, Alex Graves, Haşim Sak, Wojciech Zaremba 등의 연구자들에 의해 수년 동안 점진적으로 개선되었습니다. LSTM 셀을 블랙 박스로 생각하면 훨씬 더 우수한 성능을 낼 수 있다는 점을 제외하고는 기본 셀과 매우 비슷하게 사용할 수 있습니다. 트레이닝은 더 빨리 수렴되며 데이터의 장기 의존성을 감지합니다.

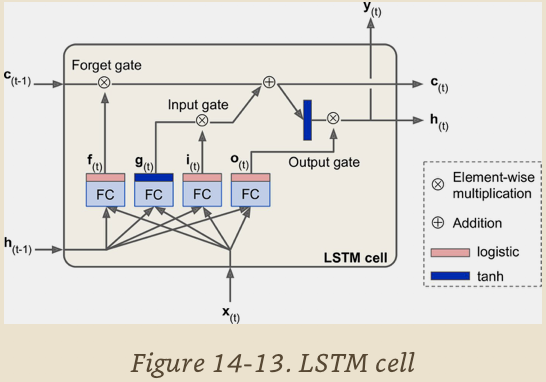
TensorFlow에서는 BasicRNNCell 대신 BasicLSTMCell을 사용할 수 있습니다.



그렇다면 LSTM 세포는 어떻게 작동합니까? 기본 LSTM 셀의 아키텍처는 그림 14-13에 나와 있습니다.

LSTM 셀은 두 개의 상태 벡터를 관리하며, 성능상의 이유 때문에 기본적으로 분리되어 있습니다.

상자 안의 내용을 보지 않으면 LSTM 셀은 h (t)와 c (t)의 두 벡터로 나뉘어져 있다는 점을 제외하고는 일반 셀과 똑같이 보입니다 ( "c"는 "cell" ). h (t)를 단기 상태로, c (t)를 장기 상태로 생각할 수 있습니다.



이제 상자를 살펴보겠습니다. 핵심 아이디어는 네트워크가 저장기간에 무엇을 저장할 것인지, 무엇을 버릴 것인지, 무엇을 읽을 것인지를 학습할 수 있다는 것입니다.

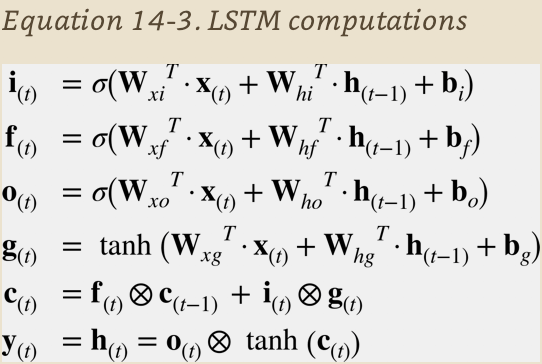
* + 장기 상태 c (t-1)가 왼쪽에서 오른쪽으로 네트워크를 가로 지르면 forget 게이트를 지나가고 메모리를 버린 다음 작업을 통해 새로운 메모리를 추가한다는 것을 알 수 있습니다. 입력 게이트에 의해 선택된 메모리를 추가 합니다.
  + 결과 c (t)는 더 이상의 변환없이 곧바로 전송됩니다. 따라서 각 시간 단계마다 일부 메모리가 사라지고 추가됩니다.
  + 또한 가산 연산 후, 장기 상태가 복사되어 tanh 함수를 통과하고, 그 결과가 출력 게이트에 의해 필터링된다. 이것은 단기 상태 h (t)를 생성한다(이 시간 단계 y (t)에 대한 셀의 출력과 동일하다).

이제 새로운 메모리의 출처와 게이트가 어떻게 작동하는지 살펴 보겠습니다.

* + 먼저, 현재의 입력 벡터 x (t)와 이전의 단기 상태 h (t-1)는 완전히 다른 4 개의 레이어에 공급된다. 그들은 모두 다른 목적을 가지고 있습니다 :
  + 메인 레이어는 g (t)를 출력하는 레이어 입니다. 현재 입력 x (t)와 이전 (단기) 상태 h (t-1)를 분석하는 일반적인 역할을 합니다. 기본 셀에는 이것 이외의 것이 없으며 출력은 y (t) 및 h (t)로 곧바로 이동합니다. 대조적으로, LSTM 셀에서 이 레이어의 출력은 곧바로 진행되지 않지만 대신 부분적으로 장기 상태로 저장됩니다.
  + 세 개의 다른 레이어는 게이트 컨트롤러입니다. 로지스틱 액티비티 기능을 사용하기 때문에 이들의 출력 범위는 0에서 1까지입니다. 이들의 출력은 요소 단위의 곱셈 연산에 공급되므로 0을 출력하면 게이트를 닫고 1을 출력하면 열게 됩니다. 구체적으로 :  
    - forget 게이트(f (t)에 의해 제어 됨)는 장기 상태의 어느 부분이 지워 져야 하는지를 제어합니다.  
    - 입력 게이트 (i (t)에 의해 제어 됨)는 g (t)의 어떤 부분을 장기 상태에 추가해야 하는지를 제어합니다 (이것이 우리가 단지 부분적으로 저장되었다고 말한 이유입니다).   
    - 출력 게이트 (o (t)에 의해 제어됨)는 장기 상태의 어느 부분을 이 시간 단계 (h (t)와 y (t) 모두)에서 읽고 출력해야 하는지를 제어합니다.

즉, LSTM 셀은 중요한 입력을 인식하여 장기 상태로 저장하고, 필요한만큼 오랫동안 보존하는 법과 필요할 때마다 추출하는 법을 배웁니다. 이것은 그들이 time series, long texts, audio recordings 등에서 장기적인 패턴을 알아내는데 왜 놀랍도록 성공적이었는지 설명합니다.

방정식 14-3은 단일 인스턴스의 각 시간 단계에서 셀의 장기 상태, 단기 상태 및 출력을 계산하는 방법을 요약합니다 (전체 미니 배치의 방정식은 매우 유사합니다).

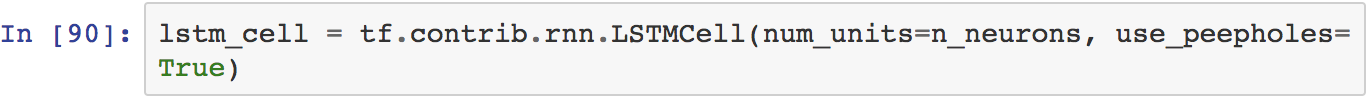


* + Wxi, Wxf, Wxo, Wxg는 입력 벡터 x (t)에 연결하기위한 4 개의 레이어 각각의 가중치 행렬입니다
  + Whi, Whf, Who 및 Whg는 이전 단기 상태 h (t-1)에 대한 연결에 대한 4개의 계층 각각의 가중치 행렬입니다.
  + bi, bf, bo 및 bg는 4 개의 레이어 각각에 대한 바이어스 항이다. TensorFlow는 bf를 0 대신 1로 가득 찬 벡터로 초기화합니다. 이렇게하면 훈련 시작시 모든 것을 잊어 버리는 것을 방지 할 수 있습니다.

**5.1 Peephole Connections**

 기본 LSTM 셀에서 게이트 컨트롤러는 입력 x (t)와 이전 단기 상태 h (t-1)만 볼 수 있습니다. 2000 년에 Felix Gers와 Jürgen Schmidhuber에 의해 아이디어가 제안되었다. 그들은 peephole 연결이라고 불리는 여분의 연결을 가진 LSTM 변형을 제안했다. 이전의 장기 상태 c (t-1)는 forget게이트의 컨트롤러에 입력으로 추가되고 출력 게이트의 컨트롤러에 현재 장기 상태 c (t)가 입력으로 추가됩니다.

TensorFlow에서 훔쳐보기 연결을 구현하려면 BasicLSTMCell 대신 LSTMCell을 사용하고 use\_peepholes = True를 설정해야합니다.



LSTM 셀의 다른 많은 변형이 있습니다.

특히 많이 사용되는 변종은 GRU cell입니다.

1. **GRU Cell**

Gated Recurrent Unit (GRU) 셀 (그림 14-14 참조)은 조경현 (Kyunghyun Cho) 등에 의해 제안되었다. 앞서 언급 한 인코더 - 디코더 네트워크도 소개 한 2014 년 7 월에 발표되었습니다.

GRU 셀은 LSTM 셀의 간소화 된 버전이며 잘 작동하는 것처럼 보입니다 (증가하는 인기도를 설명합니다). 주요 단순화는 다음과 같습니다.

* + 두 상태 벡터가 단일 벡터 h (t)로 병합됩니다.
  + 단일 게이트 컨트롤러는 잊어 버린 게이트와 입력 게이트를 모두 제어합니다. 게이트 컨트롤러가 1을 출력하면, 잊어 버린 게이트는 열리고 입력 게이트는 닫힙니다. 0을 출력하면 반대가 발생합니다. 즉, 메모리를 반드시 저장해야 할 때마다 저장 위치가 먼저 지워집니다. 이것은 사실 LSTM 셀 자체의 빈번한 변형입니다.
  + 출력 게이트가 없습니다. 전체 상태 벡터는 매 시간 단계마다 출력됩니다. 그러나 이전 상태의 어느 부분이 메인 레이어에 표시 될지를 제어하는 ​​새로운 게이트 컨트롤러가 있습니다.

식 14-4는 단일 인스턴스의 각 시간 단계에서 셀의 상태를 계산하는 방법을 요약합니다.

LSTM 또는 GRU 세포는 최근 자연 언어 처리 (NLP) 응용 프로그램에 사용되는 RNN의 성공을 뒷받침하는 주된 이유 중 하나입니다.

1. **Natural Language Processing**

 기계 번역, 자동 요약, 구문 분석, 정서 분석 등과 같은 최첨단 NLP 응용 프로그램의 대부분은 RNN을 기반으로 합니다. 이 마지막 절에서는 기계번역 모델의 모양을 간략하게 살펴 보겠습니다.

이 항목은 TensorFlow의 Word2Vec 및 Seq2Seq 튜토리얼에서 다루어지므로 확실히 확인해야 합니다.

**7.1 Word Embeddings**

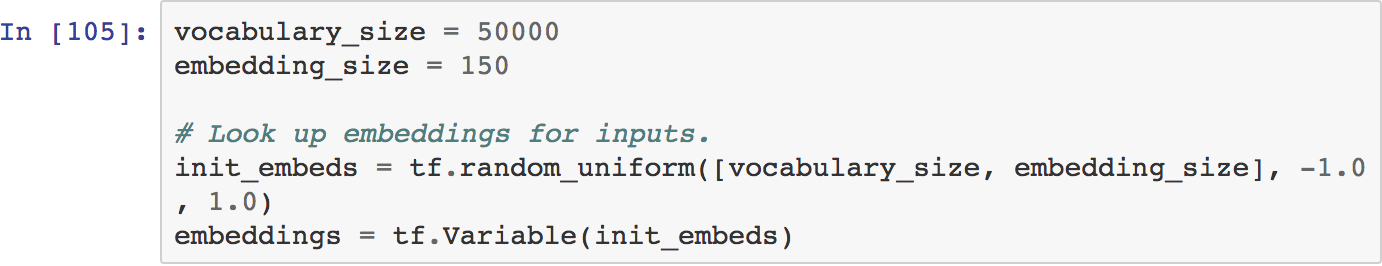
 시작하기 전에 단어 표현을 선택해야 합니다. 하나의 옵션은 one-hot 벡터를 사용하여 각 단어를 표현하는 것일 수 있습니다. 어휘에 50,000 단어가 포함된 경우 n 번째 단어는 50,000 차원 벡터로 표시되며 n 번째 위치의 1을 제외하고는 0으로 채웁니다. 그러나, 그러한 많은 양의 어휘로 이 희소한 표현은 전혀 효과적이지 않을 것입니다.

이상적인 것은 유사 단어가 비슷한 표현을 가지기 원하기 때문에 모델이 비슷한 단어에 대해 배운 것을 일반화하는 것이 쉽습니다. 예를 들어 모델에 "나는 우유를 마시라"라는 문구가 유효한 문장이며, "우유"가 "물"에 가까우면서 "신발"에서 멀리 떨어져 있다는 것을 알고 있다면 "나는 물을 마신다" 아마도 유효한 문장일 것입니다. 반면에 "나는 신발을 마신다"는 아마 그렇지 않을 것입니다. 그러나 어떻게 이런 의미있는 표상을 생각해 낼 수 있을까?

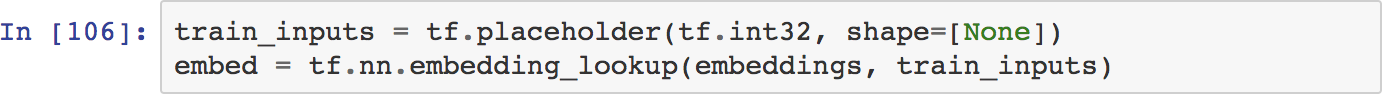
가장 보편적인 해결책은 임베딩 (embedding)이라고 불리는 상당히 작고 밀집된 벡터를 사용하여 어휘의 각 단어를 표현하고 트레이닝 중에 각 단어에 대해 신경망을 잘 배울 수 있도록하는 것입니다. 트레이닝이 시작될 때 임베딩은 무작위로 선택되지만, 트레이닝 중에는 역전파가 신경망이 작업을 수행하는데 도움되는 방식으로 임베딩을 자동으로 이동시킵니다.

일반적으로 이것은 비슷한 단어가 점차 서로 가깝게 클러스터되고 심지어 의미있는 방식으로 구성되는 것을 의미합니다. 예를 들어, 임베딩은 성별, 단수 / 복수, 형용사 / 명사 등을 나타내는 다양한 축을 따라 배치 될 수 있습니다. 결과는 정말 놀랍습니다.

TensorFlow에서 어휘의 모든 단어에 대한 포함을 나타내는 변수를 먼저 생성해야합니다.



이제 당신이 신경망에 "나는 우유를 마시라"라는 문장을 먹이고 싶다고 가정해 봅시다. 먼저 문장을 사전 처리하여 알려진 단어 목록으로 분리해야합니다. 예를 들어 불필요한 문자를 제거하고 "[UNK]"와 같은 미리 정의된 토큰 단어로 알 수없는 단어를 대체하거나 숫자 값을 "[NUM]"으로 대체하거나 URL을 "[URL]"로 바꿀 수 있습니다. 일단 알려진 단어의 목록이 있으면 사전에서 각 단어의 정수 식별자 (0에서 49999까지)를 찾을 수 있습니다 (예 : [72, 3335, 288]). 이 시점에서 placeholder를 사용하여 TensorFlow에 이러한 단어 식별자를 제공하고 embedding\_lookup () 함수를 적용하여 해당 포함을 가져올 수 있습니다.



모델이 좋은 단어 삽입을 학습하면 NLP 응용 프로그램에서 실제로 효율적으로 재사용 할 수 있습니다. 응용 프로그램이 무엇이든 "우유"는 여전히 "물"에 가까우며 "신발"에서 멀리 떨어져 있습니다. 실제로, 자신의 단어 삽입을 훈련하는 대신 미리 짜여진 단어 삽입을 다운로드 할 수 있습니다. 사전 학습된 레이어 (11 장 참조)를 재사용 할 때와 마찬가지로 미리 짜넣은 embedding을 고정하거나 (예 : trainable = False를 사용하여 embeddings 변수를 만들거나) 응용 프로그램에서 backpropagation을 조정할 수 있습니다. 첫 번째 옵션을 사용하면 교육 속도가 빨라지지만 두 번째 옵션을 사용하면 약간 높은 성능을 얻을 수 있습니다.

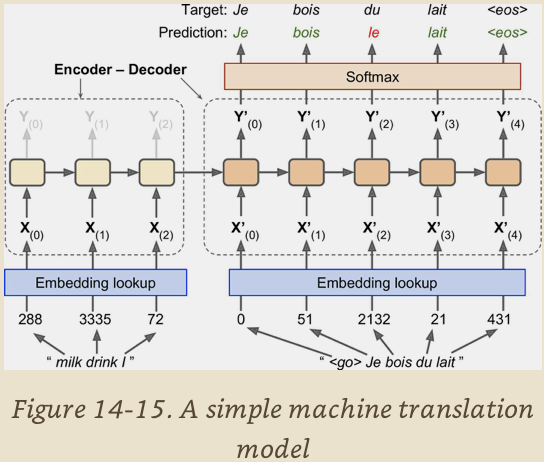
[Tip]

[임베딩은 특히 값 사이에 복잡한 유사성이있는 경우 많은 수의 다른 값을 취할 수있는 범주 속성을 나타내는 데 유용합니다. 예를 들어 직업, 취미, 요리, 종, 브랜드 등을 생각해보십시오.]

이제는 기계 번역 시스템을 구현하는 데 필요한 거의 모든 도구가 있습니다. 이제 이것을 보도록 하겠습니다.

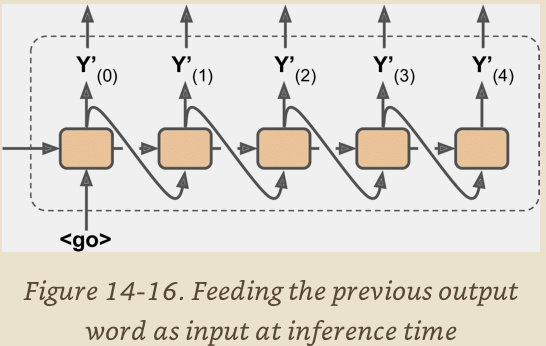
**7.2 An Encoder– Decoder Network for Machine Translation**

영어 문장을 불어로 번역할 간단한 기계 번역 모델을 살펴 보겠습니다 (그림 14-15 참조).



* + 영어 문장은 인코더에 공급되고 디코더는 프랑스어 번역을 출력합니다. 프랑스어 번역은 디코더의 입력으로도 사용되지만 한 단계씩 뒤로 밀려나는 점에 유의하십시오. 즉, 디코더는 실제 출력 내용에 관계없이 이전 단계에서 출력해야하는 단어를 입력으로 제공합니다.
  + 첫 번째 단어의 경우 문장의 시작을 나타내는 토큰("<go>")이 제공됩니다. 디코더는 EOS (End of Sequence) 토큰("<eos>")으로 문장을 끝내야합니다.
  + 영어 문장은 인코더에 공급되기 전에 되돌려집니다. 예를 들어, "I drink milk"는 "milk drink I"로 바뀝니다. 이렇게하면 영어 문장의 시작 부분이 인코더에 마지막으로 공급됩니다. 이것은 일반적으로 디코더가 번역해야하는 첫 번째 항목이기 때문에 유용합니다.
  + 각 단어는 처음에 간단한 정수 식별자로 표시됩니다 (단어 "milk"의 경우 288).
  + 다음으로, embedding lookup은 word embedding을 리턴한다 (앞에서 설명한 바와 같이, 이는 밀도가 높고 상당히 낮은 차원의 벡터이다). 이러한 단어 삽입은 실제로 인코더와 디코더에 제공됩니다.
  + 각 단계에서, 디코더는 출력 어휘 (즉, 프랑스어)에서 각 단어에 대한 점수를 출력하고, Softmax 층은 이들 점수를 확률로 전환시킨다. 예를 들어, 첫 번째 단계에서 "Je"라는 단어의 확률은 20 %이고 "Tu"의 확률은 1 %입니다. 확률이 가장 높은 단어가 출력됩니다. 이것은 일반적인 분류 작업과 매우 비슷하므로 softmax\_cross\_entropy\_with\_logits () 함수를 사용하여 모델을 학습 할 수 있습니다.

트레이닝 후 추론 시간에서는 디코더에 공급할 대상 문장이 없습니다. 대신 그림 14-16에 표시된 것처럼 이전 단계에서 출력 한 단어를 디코더에 공급하면됩니다.



당신은 큰 그림을 보았습니다. 그러나 TensorFlow의 sequence-to-sequence 튜토리얼을 통해 rnn / translate / seq2seq\_model.py 의 코드를 살펴보면 몇 가지 중요한 차이점을 알 수 있습니다.

* + 먼저, 모든 입력 시퀀스가 일정한 길이라고 가정했습니다. 그러나 분명히 문장 길이가 다를 수 있습니다. 앞에서 다룬 방법과는 다른 방법이 이 튜토리얼에서 사용됩니다. 문장은 유사한 길이의 버킷으로 그룹화된다 (예를 들어 1-6 문장의 버킷, 7-12 문장의 문장 등등), 짧은 문장은 특수 패딩 토큰 ("<pad>")을 사용하여 패딩됩니다.   
    예를 들어, "나는 우유를 마신다"는 "<pad> <pad> <pos> milk drink I"가되며, "Je bois du lait <eos> <pad>"가됩니다. 물론, 우리는 EOS 토큰의 지나간 출력을 무시하고 싶습니다. 이를 위해 튜토리얼의 구현에서는 target\_weights 벡터를 사용합니다.   
    예를 들어, "Je bois du lait <eos> <pad>"대상 문장의 경우, 가중치는 [1.0, 1.0, 1.0, 1.0, 1.0, 0.0] (대상 문장의 패딩 토큰의 가중치 0.0을 확인). 단순히 손실을 목표 가중치로 곱하면 EOS 토큰의 지나간 단어에 해당하는 손실은 제로가됩니다.
  + 둘째, 출력 어휘가 큰 경우 가능한 모든 단어에 대해 확률을 출력하는 것은 대단히 느릴 것입니다. 목표 어휘가 50,000 개의 프랑스어 단어를 포함하면, 디코더는 50,000 차원의 벡터를 출력하고, 그러한 큰 벡터에 대해 softmax 함수를 통해 softmax 함수를 계산하는 것은 매우 계산 집약적일 수 있습니다.   
    이를 피하기 위해서는 해독기가 1,000 차원 벡터와 같은 훨씬 더 작은 벡터를 출력하도록 한 다음 표적 어휘의 모든 단어에 대해 계산하지 않고 샘플링 기술을 사용하여 손실을 추정해야 합니다. 이 Sampled Softmax 기술은 2015 년 Sébastien Jean 등이 도입했습니다. TensorFlow에서 sampled\_softmax\_loss () 함수를 사용할 수 있습니다.
  + 셋째, 튜토리얼의 구현은 디코더가 입력 시퀀스를 들여다 볼 수 있게 해주는 attention 메커니즘을 사용합니다. 주의력이 강화된 RNN은 이 책의 범위를 벗어나지만, 관심이 있다면 기계 번역, 기계 판독, 및 이미지 캡션에 대한 유용한 문서가 있습니다.
  + 마지막으로 튜토리얼의 구현은 tf.nn.legacy\_seq2seq 모듈을 사용합니다. 이 모듈은 다양한 Encoder-Decoder 모델을 쉽게 구축 할 수있는 도구를 제공합니다.   
    예를 들어, embedding\_rnn\_seq2seq () 함수는 그림 14-15에 표시된 것과 같이 자동으로 단어 임베딩을 처리하는 간단한 인코더 - 디코더 모델을 만듭니다. 이 코드는 새로운 tf.nn.seq2seq 모듈을 사용하도록 빠르게 업데이트 될 것입니다.

이제 sequence-to-sequence 튜토리얼의 구현을 이해하는데 필요한 모든 도구가 제공됩니다. 그것을 확인하고 자신의 영어 - 프랑스어 번역기를 만들어 보십시오!