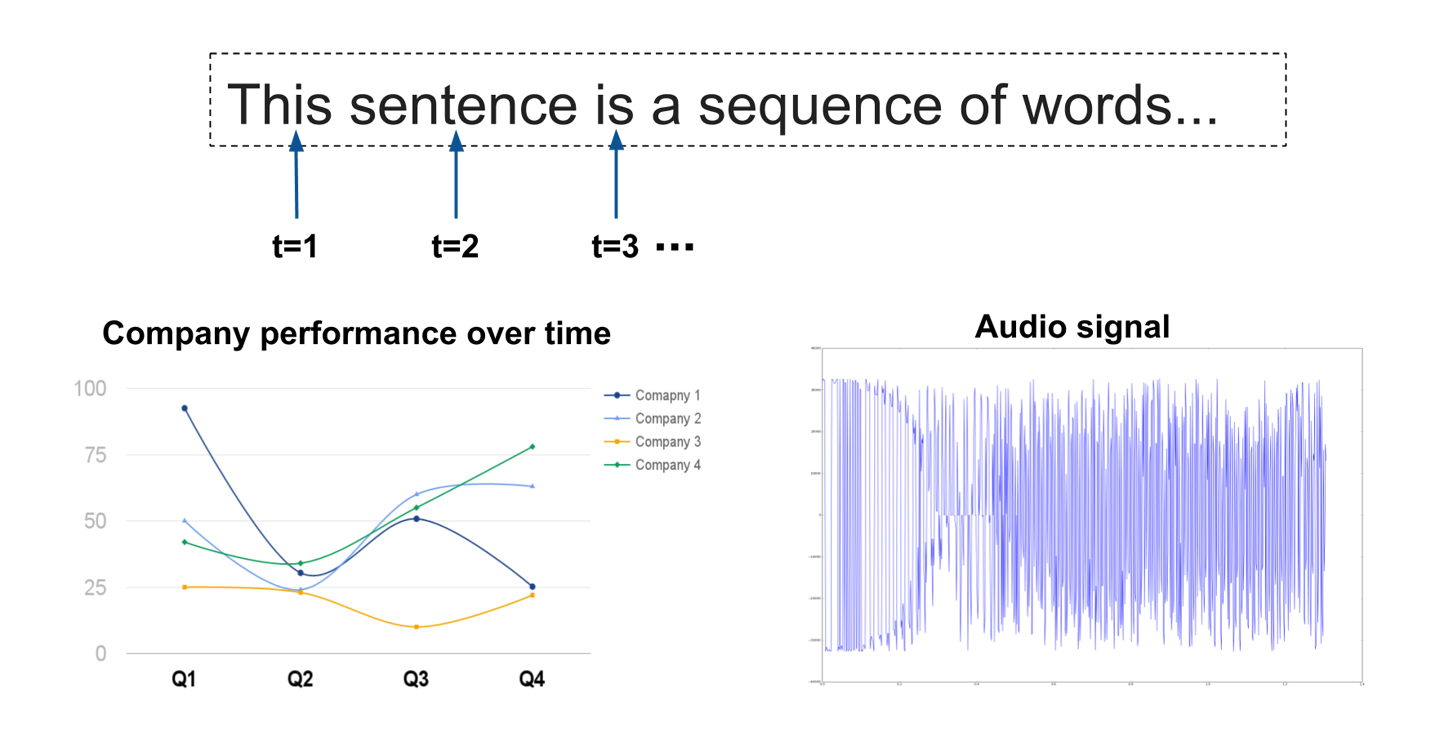
Working with Text and Sequences, and Tensorboard Visualization

텐서플로에서 시퀀스(sequence), 특히 텍스트를 어떻게 다루는지를 보여준다. 우선 RNN(Recurrent Neural Network)를 소개하는데, 이는 NLP(Natural Language Processing)에 특히 유용하고 널리 사용되는 강력한 종류의 딥러닝 알고리즘이다. RNN 모델을 밑바닥부터 구현하는 방법을 살펴보고, 몇 가지 중요한 tensorflow 기능을 도입할 것이며, 대화형 텐서보드로 모델을 시각화해볼 것이다. 이어서 단어 임베딩 학습(Word-embedding training)과 함께, 지도 학습을 사용한 텍스트 분류에 RNN을 사용하는 방법을 탐구한다. 마지막으로 LSTM(Long Short-Term Memory) 네트워크를 적용한 고급 RNN 모델을 만드는 방법과 가변 길이의 시퀀스를 다루는 방법을 다루겠다.

The Importance of Sequence Data

앞에서 이미지의 공간 구조를 사용하여 뛰어난 결과를 보여주는 고급 모델을 만들고 구조를 활용하는 것이 매우 중요하다는 교훈을 얻었다. 하지만 순차형(sequential) 구조 또한 매우 중요하고 유용한 유형의 구조다. 데이터 과학의 측면에서 순차형 구조는 모든 도메인의 많은 데이터에서 찾을 수 있다.



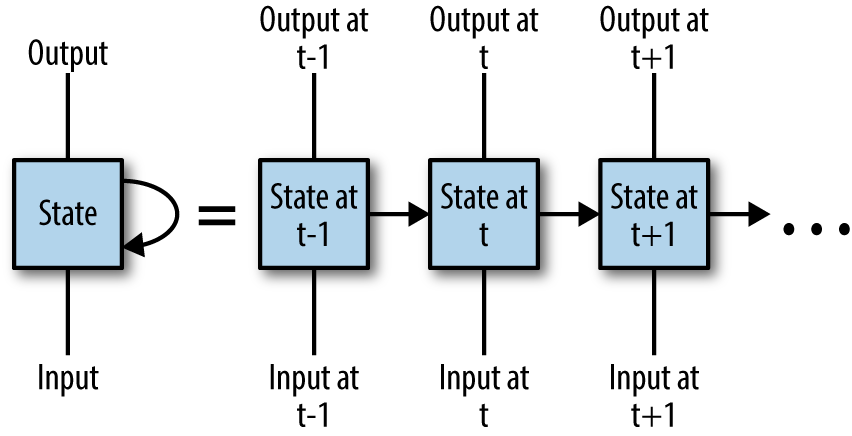
시퀀스 데이터

Introduction to Recurrent Neural Network

순환 신경망(RNN)은 강력하고 널리 사용되는 신경망 구조로 시퀀스 데이터의 모델링에 사용된다. RNN 모델의 바탕에는, 시퀀스에서 현재 이후의 각 데이터는 새로운 정보를 제공하므로, 이 정보로 모델의 현재 상태를 갱신한다는 아이디어가 깔려 있다.

우리가 새로운 정보를 접했을 때는 ‘역사’나 ‘기억’이 사라지는 대신 ‘갱신’된다. 어떤 텍스트에서 문장을 읽을 때 각각의 새로운 단어로 현재 상태의 정보가 갱신되는데 이 상태는 새롭게 등장한 단어 뿐 아니라 이전의 단어에도 종속적이다.

머신러닝을 통해 시퀀스 패턴을 모델링하기 위한 구성 요소로 흔히 사용되는 통계나 확률 기반의 수학적 구조는 마르코프 체인 모델이다. 비유적으로 말하자면 데이터 시퀀스를 ‘체인’으로 볼 수 있는데, 체인의 각 노드는 이전 노드로부터 어떤 식으로든 종속적이므로 ‘과거’는 지워지지 않고 이어진다.



RNN 모델도 체인 구조 개념을 기반으로 하고 있으며 정보를 유지하고 갱신하는 방법에 따라 종류가 다양하다. ‘순환’이라는 이름에서 유추할 수 있듯 RNN은 일종의 ‘루프’로 이루어진다. 위 그림에서 볼 수 있듯 어떤 시점 에서 네트워크는 입력값 (문장 중 하나의 단어)를 관찰하고 ‘상태 벡터’를 이전의 에서 로 갱신한다. 새로운 입력(다음 단어)을 처리하는 과정은 에 의존하는 어떤 방법이 되므로 과거 시퀀스에 종속적이다. 그림에서 보는 것처럼, 순환 구조는 하나의 긴 펼쳐놓은 체인으로 단순화해서 생각할 수 있다. 체인의 각 노드는 이전 노드의 결과로부터 얻은 ‘메시지’에 기반한 동일한 종류의 ‘단계’를 수행한다. 이는 물론 앞에서 언급한 마르코프 체인 모델, 그리고 그 확장인 은닉 마르코프 모델(Hidden Markov Model)과도 관계가 있다.

Vanilla RNN Implementation

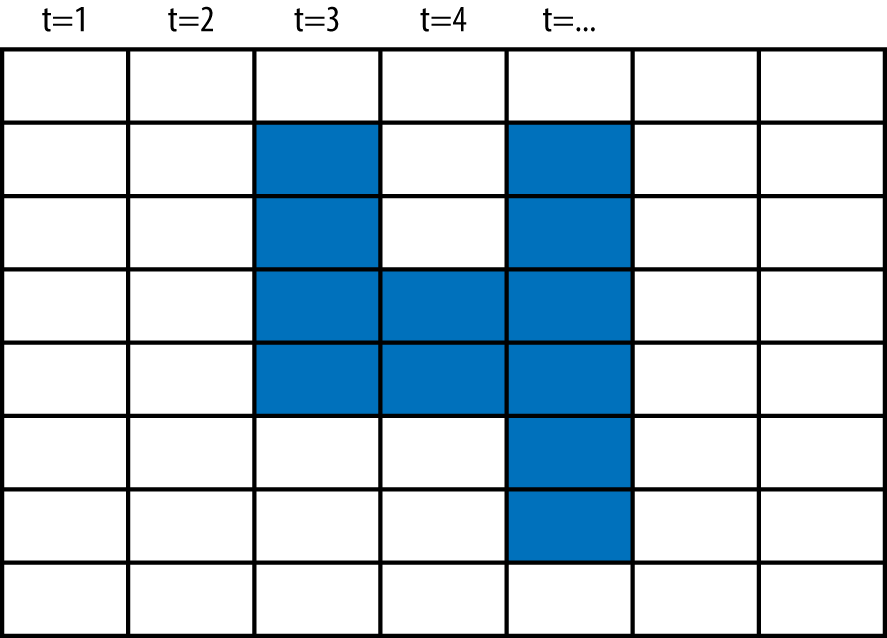
이 절에서는 기초적인 RNN을 바닥부터 구현하고 내부 동작을 탐험하면서 tensorflow를 사용해 시퀀스를 다루는 방법에 대해서 알아본다. 시퀀스 데이터를 다룰 수 있도록 tensorflow가 제공하는 몇 가지 강력한 저수준 도구도 소개한다.

일단 기초 모델을 수학적으로 정의하는 데서 시작한다. 이 작업은 RNN의 갱신 단계를 정의하는 것으로 이루어진다. 간단한 기본적인 RNN의 갱신 단계는 다음과 같다.

여기서 는 학습할 가중치와 편향값의 변수이며, 는 [-1, 1]의 범위를 가지는 쌍곡선 탄젠트 함수로 이전에 사용된 Sigmoid Function과 밀접한 관계가 있다. 와 는 앞에서 설명한 것과 같이 입력과 상태 벡터이다. 마지막으로 숨은 상태 벡터에 가중치의 집합을 곱해서 위 그림에 나온 결과를 얻는다.

MNIST image as sequences

시퀀스 모델의 능력과 일반적인 적용 가능성을 알아보기 위해, MNIST 이미지 분류를 수행하는 첫 번째 RNN을 만들어보자. 여기서는 기초적인 RNN과 시퀀스를 다루는 방법을 소개하기 위해 이미지를 간단한 시퀀스로 처리한다. 데이터의 각 이미지를 행 또는 열의 시퀀스로 본다. MNIST 데이터에서 28\*28픽셀의 각 이미지를 28의 길이를 가진 시퀀스로, 시퀀스의 각 원소는 28개의 픽셀을 가진 하나의 벡터로 볼 수 있다는 뜻이다. 이미지를 위에서 아래로 또는 왼쪽에서 오른쪽으로 스캔하여 RNN의 입력으로 사용한다고 생각할 수 있다.



픽셀 열이라는 시퀀스의 관점으로 본 이미지

(Vanilla\_RNN\_with\_tfboard.py)

element\_size는 시퀀스 벡터 차원으로, 여기서는 행 또는 열의 픽셀 크기인 28이다. time\_steps는 한 시퀀스 내에 들어 있는 이들 원소의 수이다.

이전에서 보았듯 MNIST 내장 데이터 로더를 사용하여 데이터를 읽어 들이면 784픽셀의 풀린 벡터로 저장된다. 학습 과정에서 데이터를 일괄 로딩할 때 각 펼쳐진 벡터를 [batch\_size, time\_steps, element\_size]의 형태로 간단히 변경한다.

hidden\_layer\_size는 임의로 128로 설정하여 앞에서 언급한 숨은 RNN 상태 벡터의 크기를 제어한다. LOG\_DIR은 tensorboard 시각화를 위해 모델의 요약 정보를 저장하는 디렉터리이다.

The RNN Steps

요약을 기록할 variables\_summaries() 함수를 만들고 RNN 단계에서 사용할 가중치와 편향값 변수를 만든다.

Applying the RNN step with tf.scan()

지금까지 만든 변수를 사용해 기초적인 RNN 단계를 구현하는 함수를 만들 차례다.

이 함수를 28 단계의 step에 걸쳐 적용하는데 [batch\_size, time\_steps, element\_size]의 입력값을 [time\_steps, batch\_size, element\_size]의 형태로 바꾼다. tf.transpose()의 perm 인수는 변경할 축을 지적한다. 변경 이후에는 입력 tensor의 첫 번째 축이 시간축을 나타내므로 제공되는 tf.scan() 함수를 사용하여 모든 시간 단계를 반복할 수 있다. 이 함수는 순서대로 모든 원소의 시퀀스에 반복해서 호출 가능한 객체(callable)를 적용한다.

tf.scan() (scan\_example.py)

tf.scan()은 같은 연산을 명시적으로 복제해가면서 루프를 ‘풀어놓지’ 않고 연산 그래프에 루프를 사용할 수 있게 해주는 중요한 함수이다. 기술적으로 접근하자면 이 함수는 리듀스 연산과 유사한 고계함수(higher-order function)이지만, 시간의 흐름에 따른 중간 누적값을 모두 반환한다. 몇 가지 장점이 있는 접근법인데 그중에서도 고정된 계산 속도 향상이나 최적화 방법에 비해 그래프 구성 시 동적으로 반복의 횟수를 정할 수 있다는 점이 최고의 장점이다.