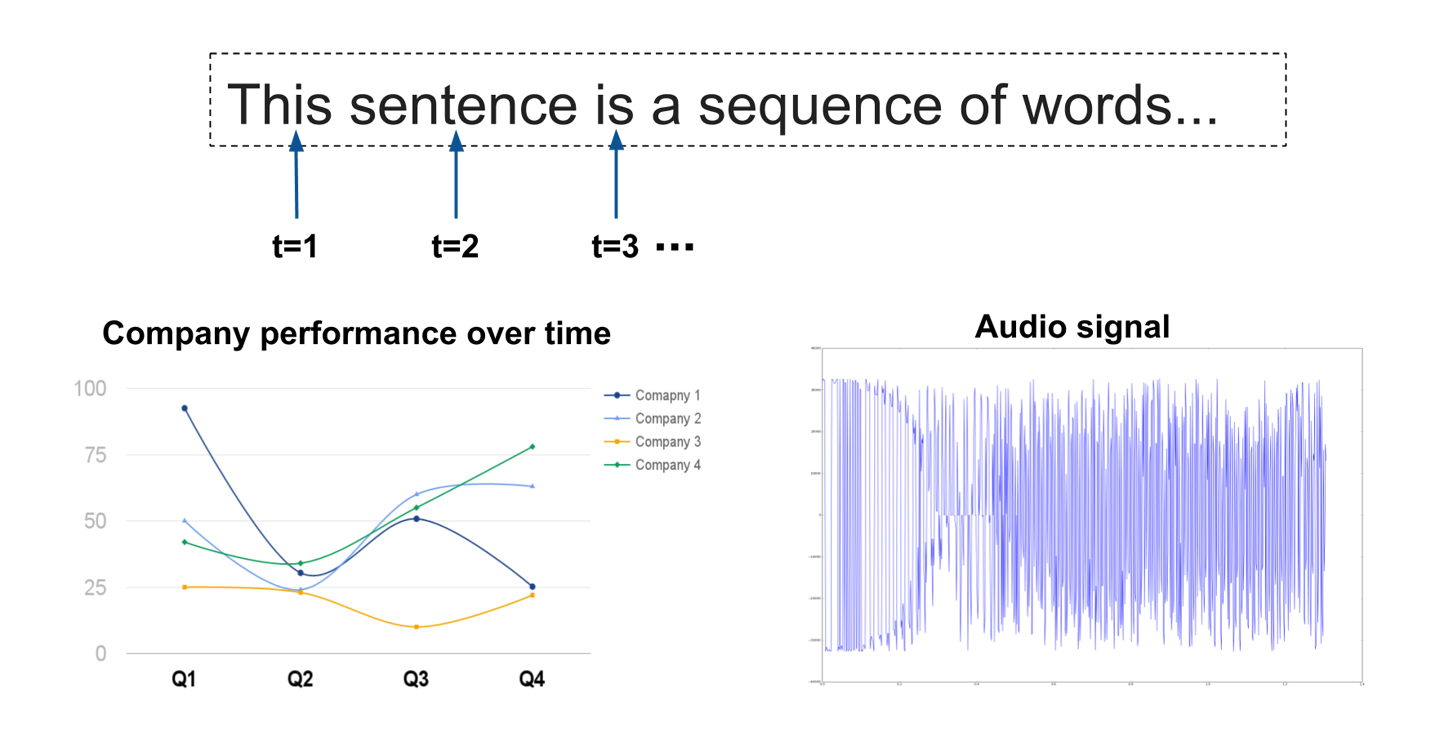
Working with Text and Sequences, and Tensorboard Visualization

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.



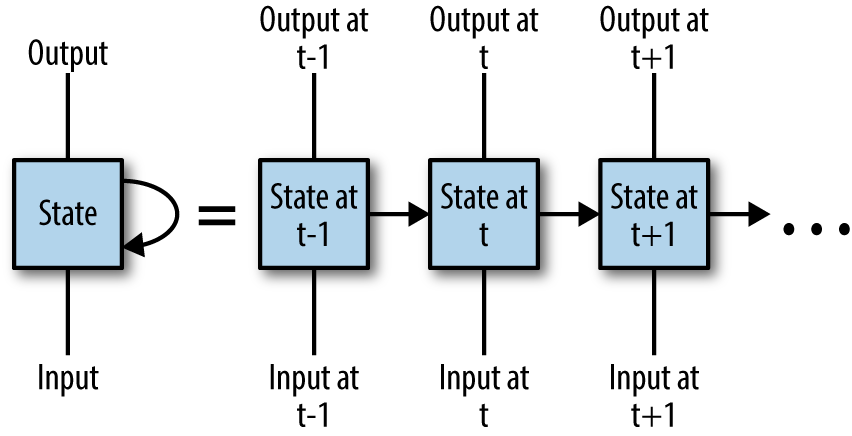
시퀀스 데이터

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

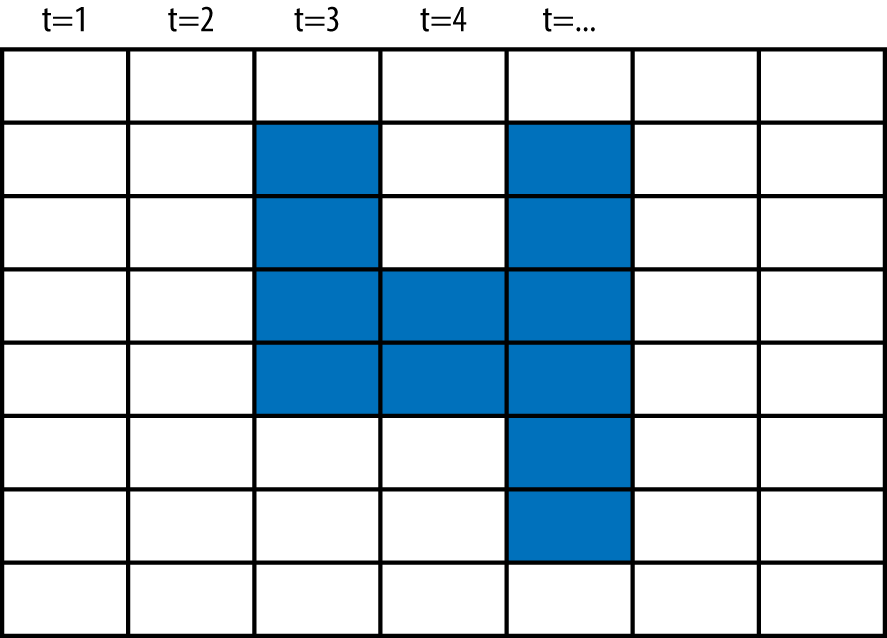
이 절에서는 기초적인 RNN을 바닥부터 구현하고 내부 동작을 탐험하면서 tensorflow를 사용해 시퀀스를 다루는 방법에 대해서 알아본다. 시퀀스 데이터를 다룰 수 있도록 tensorflow가 제공하는 몇 가지 강력한 저수준 도구도 소개한다.

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.

Sequential outputs

RNN에서는 각 시간 단계에 대한 상태 벡터에 가중치를 곱하여 데이터의 새로운 표현인 출력 벡터를 얻는다.

RNN에 들어가는 입력은 연속적이며 출력도 그러하다. 이 시퀀스 분류 예제에서는 연결된 선형 계층에 마지막 상태 벡터를 적용해서 출력 벡터를 얻는다. 기초적인 시퀀스 분류에서 일반적으로 사용되는 방법인데, 여기서 마지막 벡터는 전체 시퀀스를 표현하는 ‘누적된’ 정보를 가지고 있다고 가정한다.

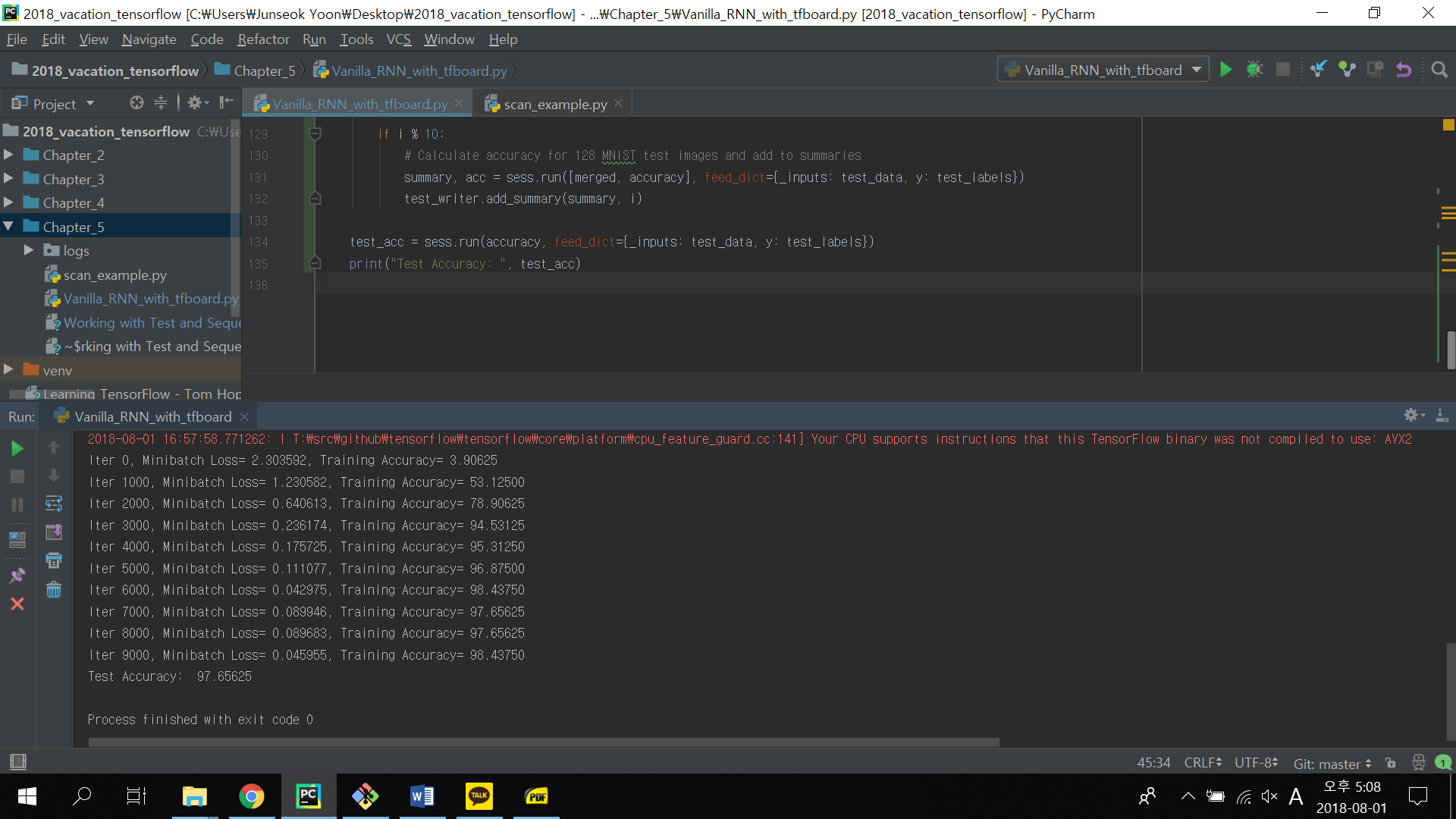
이를 구현하기 위해 먼저 선형 계층의 가중치와 편향값을 정의하고 이 계층에 대한 팩토리 함수를 만든다. 이어서 이 계층을 tf.map\_fn()를 사용해 모든 출력에 적용하는데, 이는 전형적인 맵 함수와 흡사하다. 즉 원소 쌍끼리의 시퀀스/이터러블에 함수를 적용한다.

마지막으로 네거티브 인덱싱을 사용해 각 데이터 인스턴스의 최종 출력을 뽑아낸다. 뒷부분에서는 최종 출력을 뽑는 다른 방법들을 살펴보고, 결과와 상태를 더 깊이 탐구해볼 것이다.

RNN classification

분류기를 학습할 준비가 되었으므로 손실 함수의 계산, 최적화, 예측을 위한 연산을 정의하고, tensorboard에 사용할 요약을 추가한 다음 모든 요약을 하나의 연산으로 병합한다.

몇 개의 표준 하이퍼파라미터(hyperparameter)를 가진 잘 알려진 강력한 경사 하강법 알고리즘을 구현한 RMSPropOptimizer를 사용한다. 학습에 사용하지 않은 MNIST 이미지로 작은 테스트 데이터를 생성하고 텐서보드에서 사용할 로깅을 기록하기 위한 몇 가지 기술적인 연산과 명령을 추가한 후 결과를 확인해보자.



학습 및 테스트 정확도 결과

Visualizing the model with TensorBoard

tensorboard는 학습 과정을 시각화하고 학습된 모델을 탐색할 수 있는 브라우저 기반의 대화형 도구이다. 텐서보드를 사용하려면 명령줄 터미널에서 요약 데이터를 기록한 위치를 지정해서 tensorboard에 알려줘야 한다.

tensorboard --logdir=LOG\_DIR

여기서 LOG\_DIR은 실제 로그가 위치한 디렉터리 이름으로 변경해야 한다. 윈도우에서 실행 중 정상적으로 동작하지 않는다면 터미널이 로그 데이터가 있는 해당 드라이브에서 실행 중인지 확인하고, tensorboard가 경로를 찾을 때 오류가 나는 걸 방지하기 위해서 해당 디렉터리에 이름을 추가하는 것도 방법이다.

tensorboard --logdir=rnn\_demo:LOG\_DIR

tensorboard에서는 콜론을 사용해 드라이브 이름과 경로로 구성된 각각의 로그 디렉터리를 지정할 수 있다. 여러 개의 로그 디렉터리를 사용할 때는 다음과 같이 쉼표로 구분된 로그 디렉터리의 목록을 사용할 수도 있다.

tensorboard --logdir=rnn\_demo1:LOG\_DIR1, rnn\_demo2:LOG\_DIR2

이 예제에서는 단 하나의 로그 디렉터리를 사용한다. tensorboard 명령을 실행하면 브라우저에서 입력할 URL을 다음과 같이 알려준다.

Starting Tensorboard b’39’ on port 6006

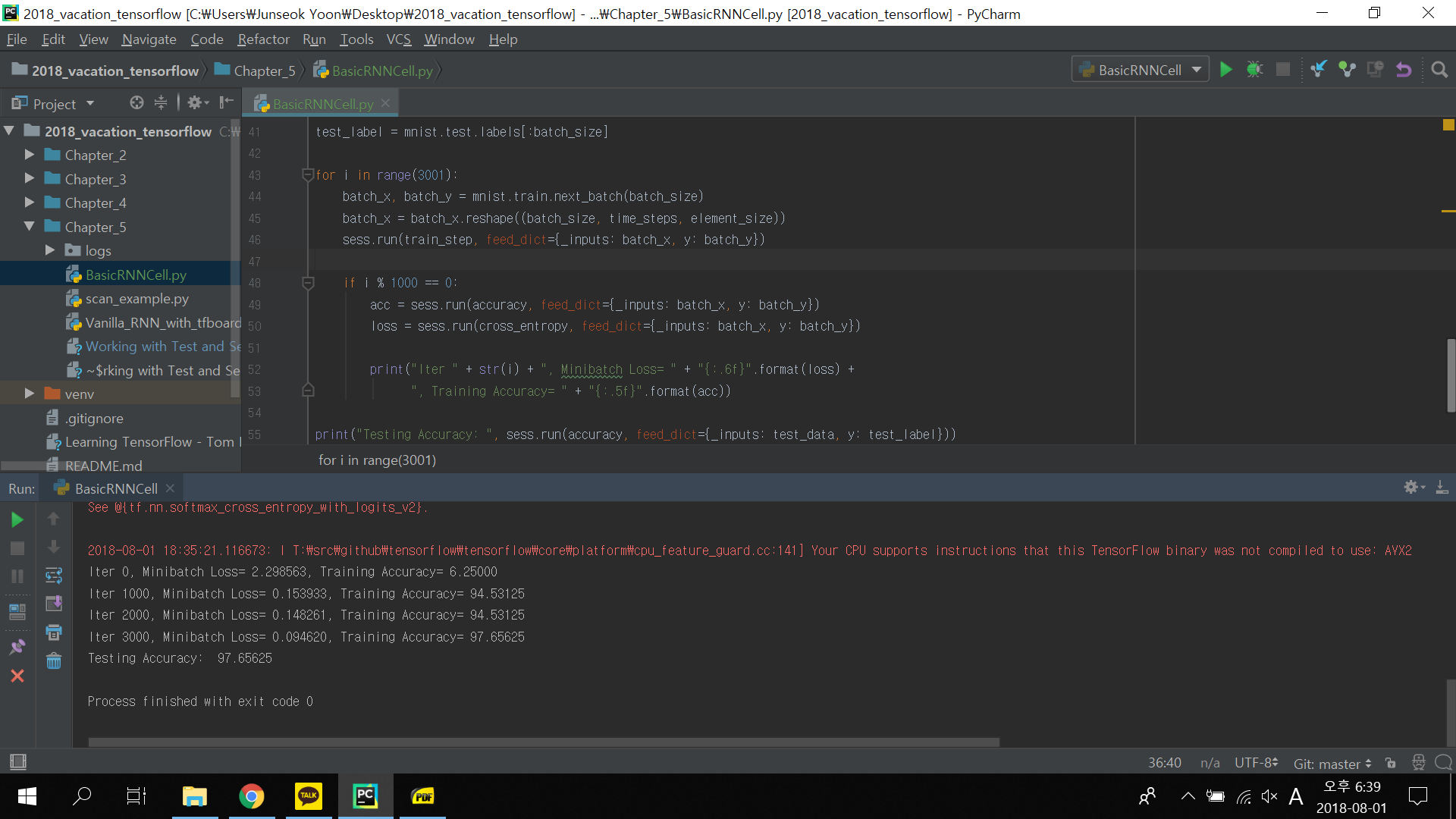
(You can navigate to <http://10.100.102.4:6006>)

tensorboard는 tfevents 로그 데이터를 포함하는 LOG\_DIR의 하위 디렉터리를 찾아 디렉터리 구조를 재귀적으로 탐색한다. 책의 예제를 여러 번 수행할 때 로그 파일을 덮어써버리는 상황을 피하려면 실행할 때마다 LOG\_DIR 디렉터리를 삭제하거나 LOG\_DIR 내 별도의 하위 디렉터리에 로그를 기록해야 한다.

TensorFlow Built-in RNN Functions

앞의 예제를 통해 밑바닥부터 그래프를 구현해보면서 시퀀스를 가지고 작업하는 기본적이고 강력한 방법들을 배웠다. 실무에서는 기본 제공되는 고수준 모듈과 함수를 사용하는 것이 당연히 좋다. 코드를 더 짧고 쉽게 만들 수 있을 뿐 아니라 tensorflow가 구현해서 제공하는 많은 저수준 최적화를 활용할 수 있다.

(BasicRNNCell.py)



학습 및 테스트 정확도 결과

tf.contrib.rnn.BasicRNNCell and tf.nn.dynamic\_rnn()

tensorflow의 RNN 셀은 각 순환 ‘셀’이 수행하는 기본 연산과 이와 연관된 상태를 표현하는 추상화이다. 이들은 rnn\_step() 함수와 필요한 관련 변수를 ‘대체’한다. 물론 이 셀의 많은 변형과 유형이 있으며 각각은 많은 메서드와 특성을 가지고 있다.

rnn\_cell을 생성하고 나면 이를 tf.nn.dynamic\_rnn()에 밀어 넣는다. 이 함수는 기초적인 RNN 구현의 tf.scan()을 대체하고 rnn\_cell에 지정된 RNN을 만든다.

contrib이 의미하는 바는 tensorflow에서 오픈소스 커뮤니티에서 개발에 기여한 코드를 반영했으나 아직 테스트가 필요하다는 뜻이다. BasicRNNCell은 tensorflow 1.0부터 개발이 진행 중인 부분에 해당하는 contrib으로 이동되었다. 버전 1.2에서는 많은 RNN 함수와 클래스가 핵심 네임스페이스로 들어갔는데 역호환성을 유지하기 위해서 contrib을 별칭으로 유지하고 있다.

RNN for Text Sequences

텍스트 데이터에는 이미지 데이터와 다른 몇 가지 특징이 있다. 처음에는 이 특징 때문에 텍스트 데이터를 다루기가 다소 어려울 수 있으며, 이런 특징을 다루기 위한 몇 가지 전처리 과정이 텍스트 데이터 처리에 반드시 들어가야 한다. 여기서는 tensorflow에서의 텍스트 처리 방식을 소개하기 위한 핵심 구성 요소에 집중하고자 바로 적용 가능한 최소한의 작위적인 텍스트 데이터를 사용할 것이다.

Text Sequence

텍스트 시퀀스는 문장을 구성하는 단어들, 문단을 구성하는 문장들, 심지어 단어를 구성하는 문자들, 혹은 하나의 전체 문서 등으로 구성될 수 있다.

예를 들어, 다음의 문장을 검토해보자

Our conpany provides smart agriculture solutions for farms, with advanced AI, deep-learning.

이 문장의 각 단어는 ID로 표현될 수 있다. 예를 들어 ‘agriculture’라는 단어는 정수 3452에, ‘farm’이라는 단어는 12로, ‘AI’라는 단어는 150으로, ‘deep-learning’은 0으로 매핑할 수 있다. 정수 식별자로 이 단어들을 표현하는 것은 이미지 데이터를 픽셀의 벡터로 표현하는 것과 여러 가지로 매우 다르다. 구체적인 설명을 위해 단순한 텍스트 데이터를 만들어보자.

시뮬레이션된 데이터는 두 분류의 아주 짧은 ‘문장’으로 구성되는데 하나는 홀수 숫자로, 다른 하나는 짝수 숫자로 구성되어 있다. 짝수를 나타내는 단어로 구성된 문장과 홀수를 나타내는 단어로 구성된 문장을 만든다.

(LSTM\_supervised\_embeddings.py)

몇 가지 상수를 정의하는 것으로 시작한다. 그 다음 문장을 만든다. 임의로 숫자를 샘플링해서 해당 ‘단어’에 매핑한다. 실제 자연어 데이터가 그러하듯 텍스트 시퀀스는 일반적으로 길이가 정해져 있지 않다. 문장의 길이를 다르게 시뮬레이션 하기 위해 각 문장마다 np.random.choice(range(3, 7))을 사용하여 3과 6 사이의 무작위한 값으로 길이를 지정한다. 이제 입력 문장들은 하나의 tensor에 넣으려면 어떻게든 같은 크기로 맞춰야 한다. 따라서 6보다 작은 길이의 문장은 0으로 채워서 모든 문장의 길이를 맞춘다. 이러한 전처리 단계를 제로 패딩(zero-padding)이라고 부른다.

PAD 단어를 데이터와 digit\_to\_word\_map 딕셔너리에 추가하고, 짝수 및 홀수 문장과 패딩 전의 원래 길이를 별도로 저장한다.

왜 원래 문장의 길이를 저장해야 할까? 제로 패딩을 사용해 기술적인 문제 하나는 해결했지만 다른 문제가 생긴다. 만약 패딩된 문장을 RNN 모델에 그대로 전달하면 RNN모델은 의미 없는 PAD 스트링까지 처리할 것이다. 이 ‘잡음’을 처리하느라 모델의 정확도를 망칠 수 있고 계산 시간도 늘어난다. 이 문제를 해결하기 위해 원래 시퀀스의 길이를 seqlens 배열에 저장하고 tensorflow의 tf.nn.dynamic\_rnn()에 각 문장이 끝나는 위치를 전달한다.

그 후 단어를 키로, 인덱스를 값으로 하는 딕셔너리를 생성하는 방식으로 단어를 인덱스에 매핑한다. 또 역방향의 매핑도 생성한다. 이때 단어 ID와 각 단어를 표현하는 숫자 사이에는 아무런 관련이 없다.

지도 학습을 통한 분류 작업이므로 다른 예제와 마찬가지로 one-hot 포맷의 레이블의 배열, 학습과 테스트 데이터, 데이터 인스턴스의 일괄 작업을 생성하는 함수 및 placeholder가 필요하다.

이어서 문장의 일괄 처리 데이터를 생성하는 함수(get\_sentence\_batch())를 만든다. 일괄 처리 데이터의 각 문장은 단어에 대응하는 정수 ID의 리스트이다. 그 후 데이터에 사용할 placeholder들을 만든다.

Supervised Word Embedding

이제 텍스트 데이터는 단어 ID의 리스트로 인코딩되었고, 각 문장은 단아에 대응하는 정수들의 시퀀스가 되었다. 각 단어가 ID로 표현되는 이런 원자적 표현은 실제 NLU에서 접하게 되는 많은 수의 어휘를 다루는 딥러닝 모델의 학습에 사용할 수 있을 만큼 확장력이 있지는 않다. 단어 ID가 수백만 개에 이를 수 있으며 각각이 one-hot 범주 형식으로 인코딩되면 데이터 희소성과 계산 효율성 면에서 문제가 심각해진다.

이 문제를 해결하는 강력한 접근법을 word embedding을 사용하는 것이다. 임베딩은 간단히 말해 고차원의 one-hot 벡터를 저차원의 고밀도 벡터로 매핑하는 것이다. 예를 들어 어휘의 크기가 10만이라면 one-hot 표현에서 각 단어는 10만의 크기를 가지게 된다 이에 해당하는 단어 벡터, 즉 단어 임베딩의 크기를 예를 들어 300쯤 될 것이다. 고차원의 one-hot 벡터는 훨씬 더 낮은 차원의 연속 벡터 공간으로 ‘embedding’된다.

이 예제의 최종 목표는 텍스트 분류 문제를 해결하는 것으로 이를 위해서 지도 학습 방식으로 단어 벡터를 학습하고 임베딩된 단어 벡터를 튜닝하여 분류 문제를 해결해본다.

단어 임베딩을 기본 해시 테이블이나 룩업 테이블로 간주하고 단어를 고밀고 벡터의 값에 매핑한다고 생각하면 이해하기 쉽다. 이 벡터는 학습 과정에서 최적화된다. 앞에서는 각 단어에 정수 인덱스를 부여했고 문장은 이들 인덱스 값의 시퀀스로 표현되었다. 임베딩에서는 기본 제공되는 tf.nn.embedding\_lookup() 함수를 사용해 주어진 단어 인덱스 시퀀스에 포함된 각 단어의 벡터를 효율적으로 가지고 올 수 있다.

LSTM and Using Sequence Length

가장 대중적인 순환신경망은 LSTM 네트워크이다. 기초적인 RNN과 다른 점은 특별한 기억 메커니즘(memory mechanism)을 가지고 있어 순환 셀이 오랜 시간 동안의 정보를 더 잘 저장할 수 있으며 따라서 일반 RNN보다 장기간에 걸친 종속성을 잘 반영할 수 있다는 점이다.

이러한 기억 메커니즘은 별반 특이할 것이 없는데, RNN이 최적화 문제를 해결하고 정보를 전파할 수 있도록 단순히 각 순환 셀에 몇 개의 매개변수를 추가하는 것뿐이다. 이들 학습 가능한 매개변수는 어떤 정보가 ‘기억’하고 전달할 만한 가치가 있는지, 그리고 어떤 것은 ‘잊어버려야’할지에 대한 필터 역할을 한다. 이들은 네트워크의 다른 매개변수와 동일하게 경사 하강법 알고리즘과 역전파를 사용하여 학습된다.

이 장 처음에 했던 것과 비슷하게 tf.contrib.rnn.BasicLSTMCell()으로 LSTM 셀을 만들고 tf.nn.dynamic\_rnn()에 데이터를 넣는다. 또 앞에서 만든 \_seqlens placeholder를 사용해 dynamic\_RNN()에 예제 데이터의 배치로 처리할 각 시퀀스의 길이를 저장한다. tensorflow는 이 길이를 사용해 시퀀스상의 실제 마지막 항목 이후의 모든 RNN 단계를 중단시킨다. 또 시간의 경과에 따른 모든 출력 벡터를 반환하는데 이때 시퀀스의 실제 마지막 항목 이후는 모두 제로 패딩된다.

Training Embeddings and the LSTM Classifier

Stacking multiple LSTMs

여러 개의 RNN 셀을 하나의 다층 셀로 결합하는 MultiRNNCell()을 사용하면 계층을 추가할 수 있다. 예를 들어 2개의 LSTM계층을 쌓으려 할 때,

num\_LSTM\_layers = 2

with tf.variable\_scope(“lstm”):

lstm\_cell\_list = [tf.contrib.rnn.BasicLSTMCell(hidden\_layer\_size, forget\_bias=1.0) for i in range(num\_LSTM\_layers)]

cell = tf.contrib.rnn.MultiRNNCell(cells=lstm\_cell\_list, state\_is\_tuple=True)

outputs, states = tf.nn.dynamic\_rnn(cell, embed, sequence\_length = \_seqlens, dtype=tf.float32)

먼저 이전처럼 LSTM셀을 정의한 후 tf.contrib.rnn.MultiRNNCell() 래퍼에 밀어 넣는다. 이제 네트워크에는 두 개의 LSTM 계층이 있으므로 최종 상태 벡터를 뽑아내려 할 때 형태의 문제가 발생한다. 두 번째 계층의 최종 상태를 얻으려면 다음과 같이 인덱스를 맞춰주면 된다.

final\_output = tf.matmul(states[num\_LSTM\_layers – 1][1], weights[“linear\_layer”] + biases[“linear\_layer”]