TensorFlow Abstractions and Simplification

Chapter Overview

프로그래밍의 맥락에서 추상화(abstraction)라는 용어는 원래 코드를 특정한 목적으로 일반화하여 기존 코드의 ‘위에 올라가는’ 코드의 계층을 이르는 말이다. 관련 있는 몇몇 고차원 기능을 묶는 방식의 재구성을 통해 코드를 묶고 감싸서 추상화한다. 다음 코드를 보자.

def weight\_variable(shape):

initial = tf.truncated\_normal(shape, stddev=0.1)

return tf.Variable(initial)

def bias\_variable(shape):

initial = tf.constant(0.1, shape=shape)

return tf.Variable(initial)

def conv2d(x, W):

return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def conv\_layer(input, shape):

W = weight\_variable(shape)

b = bias\_variable([shape[3]])

h = tf.nn.relu(conv2d(input, W) + b)

hp = max\_pool\_2x2(h)

return hp

def max\_pool\_2x2(x):

return tf.nn.max\_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

x = tf.placeholder(tf.float32, shape=[None, 784])

x\_image = tf.reshape(x, [-1, 28, 28, 1])

h1 = conv\_layer(x\_image, shape=[5, 5, 1, 32])

h2 = conv\_layer(h1, shape=[5, 5, 32, 64])

h3 = conv\_layer(h2, shape=[5, 5, 64, 32])

기본 tensorflow에서 합성곱 계층을 생성하려면 입력과 원하는 출력의 형태에 따라 가중치 및 편향값을 정의하고 초기화한 후 정의된 strider와 padding을 사용해 합성곱 연산을 적용하고 마지막으로 활성화 함수 연산을 추가해야 한다. 이러한 코드는 실수를 범하기 쉽고 이런 프로세스를 여러 번 반복하는 것은 지루한 일이 될 수도 있고 좀더 효율적으로 했으면 하는 생각이 들 것이다.

위의 코드는 중복 작업을 일부 제거하는 함수를 사용해 추상화를 시도해 본 결과물이다. 다음 코드와 가독성을 비교해보자.

x = tf.placeholder(tf.float32, shape=[None, 784])

x\_image = tf.reshape(x, [-1, 28, 28, 1])

W1 = tf.truncated\_normal([5, 5, 1, 32], stddev=0.1)

b1 = tf.constant(0.1, shape=[32])

h1 = tf.nn.relu(tf.nn.conv2d(x\_image, W1, strides=[1, 1, 1, 1], padding='SAME') + b1)

hp1 = tf.nn.max\_pool(h1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

W2 = tf.truncated\_normal([5, 5, 32, 64], stddev=0.1)

b2 = tf.constant(0.1, shape=[64])

h2 = tf.nn.relu(tf.nn.conv2d(hp1, W2, strides=[1, 1, 1, 1], padding='SAME') + b2)

hp2 = tf.nn.max\_pool(h2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

W3 = tf.truncated\_normal([5, 5, 64, 32], stddev=0.1)

b3 = tf.constant(0.1, shape=[32])

h3 = h1 = tf.nn.relu(tf.nn.conv2d(hp2, W3, strides=[1, 1, 1, 1], padding='SAME') + b3)

hp3 = tf.nn.max\_pool(h3, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

고작 3개의 계층밖에 없는데도 불구하고 만들어진 코드는 보기에 꽤 복잡하고 혼란스럽다. 전형적인 중간 크기의 코드를 넘어서는 더 길고 복잡한 코드는 보통 추상화 라이브러리로 ‘감싸서’ 제공하는 것이 좋다. tensorflow에서 사용 가능한 추상화 라이브러리 중 contrib.learn을 사용하면 선형회귀모형을 정의하고 학습하는 핵심 기능을 다음과 같이 단 두 줄로 구현할 수 있다.

regressor = learn.LinearRegressor(feature\_columns=feature\_columns, optimizer=optimizer)

regressor.fit(X, Y, steps=200, batch\_size=506)

High-Level Survey

텐서플로에서 사용할 수 있는 몇몇 훌륭한 오픈소스 중 인기있는 것은 다음과 같다.

* tf.contrib.learn
* TFLearn
* TF-Slim
* Keras

TFLearn은 설치가 필요하지만 contrib.learn과 TF-Slim은 tensorflow와 병합되어서 따로 설치가 필요 없다. 2017년 keras는 구글의 공식 후원을 받아 tf.contrib 안으로 들어왔다. contrib이라는 말은 이들 라이브러리가 기부된(contributed) 것이며 테스트가 더 필요하다는 사실을 나타낸다.

contrib.learn은 ‘짧은 코드’로 머신러닝을 수행할 수 있는 사이킷런(sklearn)의 세계에서 옮겨오려는 사람들이 tensorflow에서도 복잡한 네트워크를 쉽게 생성할 수 있도록 만든 tensorflow의 독립 인터페이스 사이킷플로(Scikit Flow)에서 시작되었다.

다른 라이브러리와 마찬가지로 contrib.learn의 주요 목표는 학습 모델을 구성하고, 학습시키고 평가하기 쉽게 만드는 것이다 매우 간단한 모델은 단 몇 줄의 코드만으로 학습이 가능하도록 곧장 구현 가능하다. contrib.learn의 또 다른 장점은 데이터의 특징을 매우 간편하게 다룰 수 있다는 점이다.

contrib.learn은 내부 구조로 들어가기 쉽고 저수준인 반면, 다른 세 확장 라이브러리는 좀 더 깔끔하고 더 추상적이며, 저마다의 특색과 이점이 있다.

contrib.learn

contrib.learn은 tensorflow와 병합되어 있으므로 따로 설치를 할 필요가 없다.

import tensorflow as tf

from tensorflow.contrib import learn

처음 설명할 것은 contrib.learn의 특별한 추정자(estimator)인데, 이를 이용하면 빠르고 효율적으로 학습이 가능하다. 미리 정의된 추정자로 간단한 선형/로지스틱 회귀모형, 간단한 선형 분류기, 기초적인 심층신경망 등이 제공된다.

|  |  |
| --- | --- |
| 추정자 | 설명 |
| LinearRegressor() | 주어진 특징 값들의 관측치에 대해 레이블을 예측하는 선형회귀모형 |
| LogisticRegressor() | 이진 분류를 위한 로지스틱 회귀 추정자 |
| LinearClassifier() | 데이터 인스턴스를 여러 개의 가능한 클래스 중 하나로 분류하는 선형모형 |
| DNNRegressor() | tensorflow DNN 모델의 추정자 |
| DNNClassifier() | tensorflow DNN 모델의 분류기 |

물론 개선된 사용자 정의 모델을 사용해야 할 때도 있고, 이를 위해서 contrib.learn은 사용자가 직접 만든 추정자를 편리하게 감쌀 수 있도록 해준다. 우리가 직접 만들었건 혹은 누군가가 다른 사람을 위해 만든 것이건 간에 배포 가능한 추정자가 있다면 사용 과정은 동일하다.

1. 모델을 생성하려면 추정자클래스를 인스턴스화한다.

model = learn.<some\_Estimator>()

1. 이어서 학습 데이터를 사용해 모델을 적합시킨다.

model.fit()

1. 주어진 데이터에서 모델이 얼마나 잘 동작하는지를 평가한다.

model.evaluate()

1. 마지막으로 새로운 데이터에 대해 학습된 모델을 사용해 결과를 예측한다.

model.predict()

Linear Regression

모델이 특징의 가중합의 함수로 정의될 때 그 모델이 선형(linear)이라고 한다. 형식적으로 말하자면, 가 어떤 종류의 함수이건 간에 로 정의될 때 이 모델이 선형이라고 한다. 비록 표현력의 제약이 있지만 선형모형은 명확한 해석 가능성, 최적화 속도, 단순성 등 많은 장점을 가지고 있다.

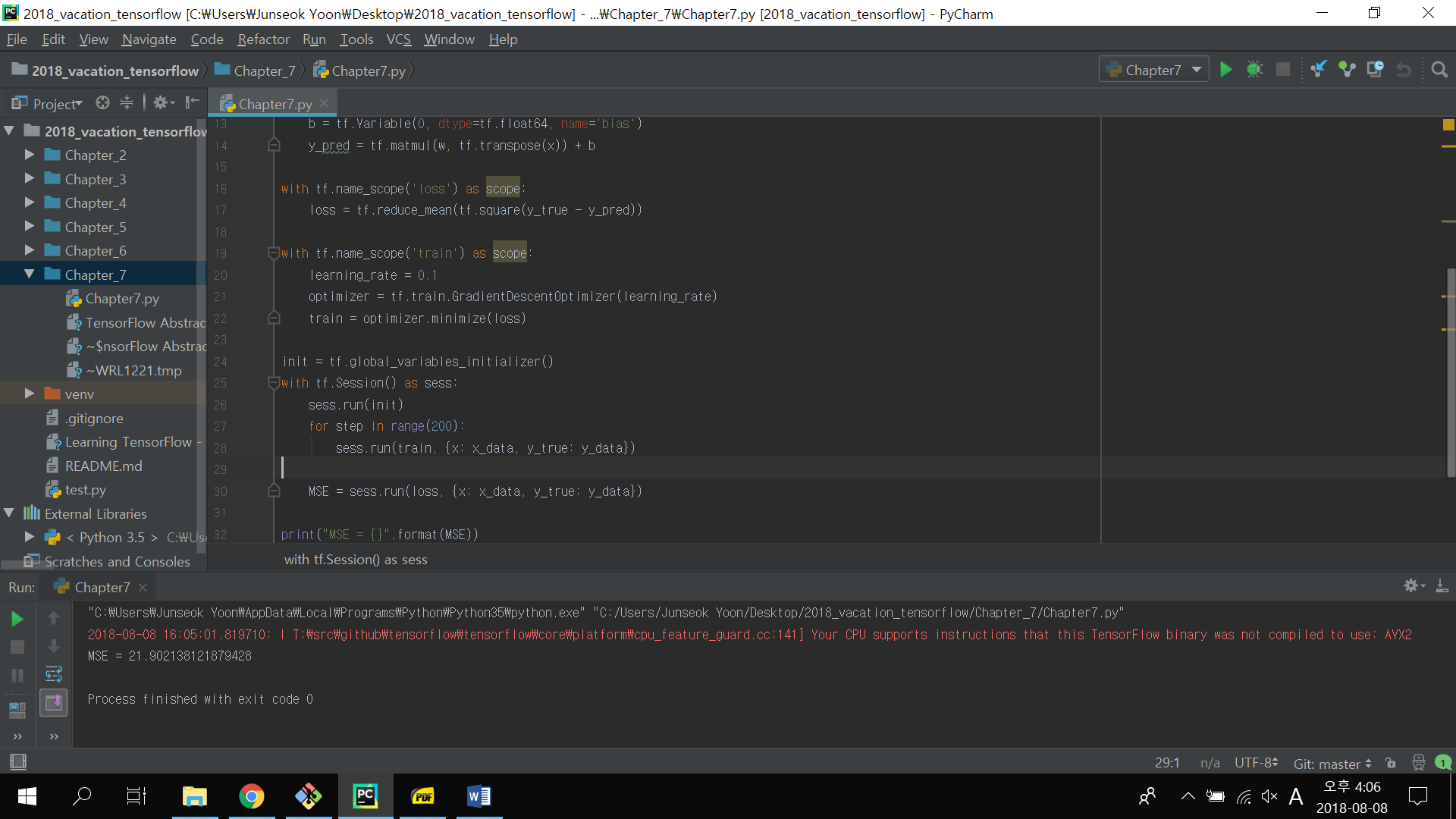
이 예제에서는 사이킷런에서 제공하는 보스턴 하우징 데이터셋을 사용할 것이다(http://bit.ly/2sXIfrX). 보스턴 하우징 데이터는 비교적 작은 데이터로 메사추세츠 주 보스턴 지역의 주택 정보가 담겨 있다. 이 데이터에는 예측변수가 13개 있다.

1. CRIM : 마을의 1인당 범죄율
2. ZN : 2만 5천 평방 피트 이상으로 구획된 택지의 비율
3. INDUS : 마을당 비상업 업무 지구의 비율
4. CHAS : 찰스강 더미 변수 (지역이 강과 접하면 1, 아닌 경우 0)
5. NOX : 질소산화물 농도 (천만 분의 일 단위)
6. RM : 주택당 평균 방의 수
7. AGE : 1940년 이전에 지어진 소유주가 사용 중인 건물의 비율
8. DIS : 5개의 보스턴 고용 센터까지의 가중 거리
9. RAD : 방사형 고속도로까지의 접근성 지수
10. TAX : 1만 달러당 부가가치세
11. PTRATIO : 마을별 학생 대 교사 비율
12. B : 1000(는 마을의 아프리카계 미국인의 비율
13. LSTAT : 하위 계층의 비율(%)

목표변수는 소유주가 거주 중인 주택 가격의 중간값이다. 이 예제에서는 13가지 특징의 선형 조합을 사용하여 목표변수를 예측하고자 한다. (Chapter7.py)

먼저 데이터를 가져온 후 3장에서와 같은 선형회귀모형을 사용한다. ‘손실’을 추적할 수 있으므로 평균제곱오차(MSE)를 측정할 수 있다. 앞에서도 봤지만 MSE는 실제 관측값과 모델 예측값 사이의 차의 제곱의 평균이다. 이 척도를 모델이 얼마나 잘 동작하고 있는지를 알려주는 지표로 사용한다.

200회 반복 후 학습 데이터에 대해 계산된 MSE는 다음과 같다.



이제 선형회귀에 contrib.learn의 추정자를 사용해 똑같은 과정을 수행해본다. 모델을 정의하고 적합시키고 평가하는 전체 과정은 단지 몇 줄로 줄어든다.

1. 선형회귀모형은 데이터의 표현과 최적호 함수의 유형에 관한 내용을 인수로 하는 learn.LinearRegressor()를 사용해 인스턴스화한다,

reg = learn.LinearRegressor(feature\_columns=feature\_columns,

optimizer=tf.train.GradientDescentOptimizer(

learning\_rate=0.1))

1. regressor 객체는 .fit()을 사용해 학습된다. 공변량(covariate)과 목표변수를 전달하고 단계의 수와 배치 크기를 설정한다.

reg.fit(x\_data, boston.target, steps=NUM\_STEPS, batch\_size=MINIBATCH\_SIZE)

1. .evaluate()는 MSE 손실값을 반환한다.

MSE = regressor.evaluate(x\_data, boston.target, steps=1)