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.

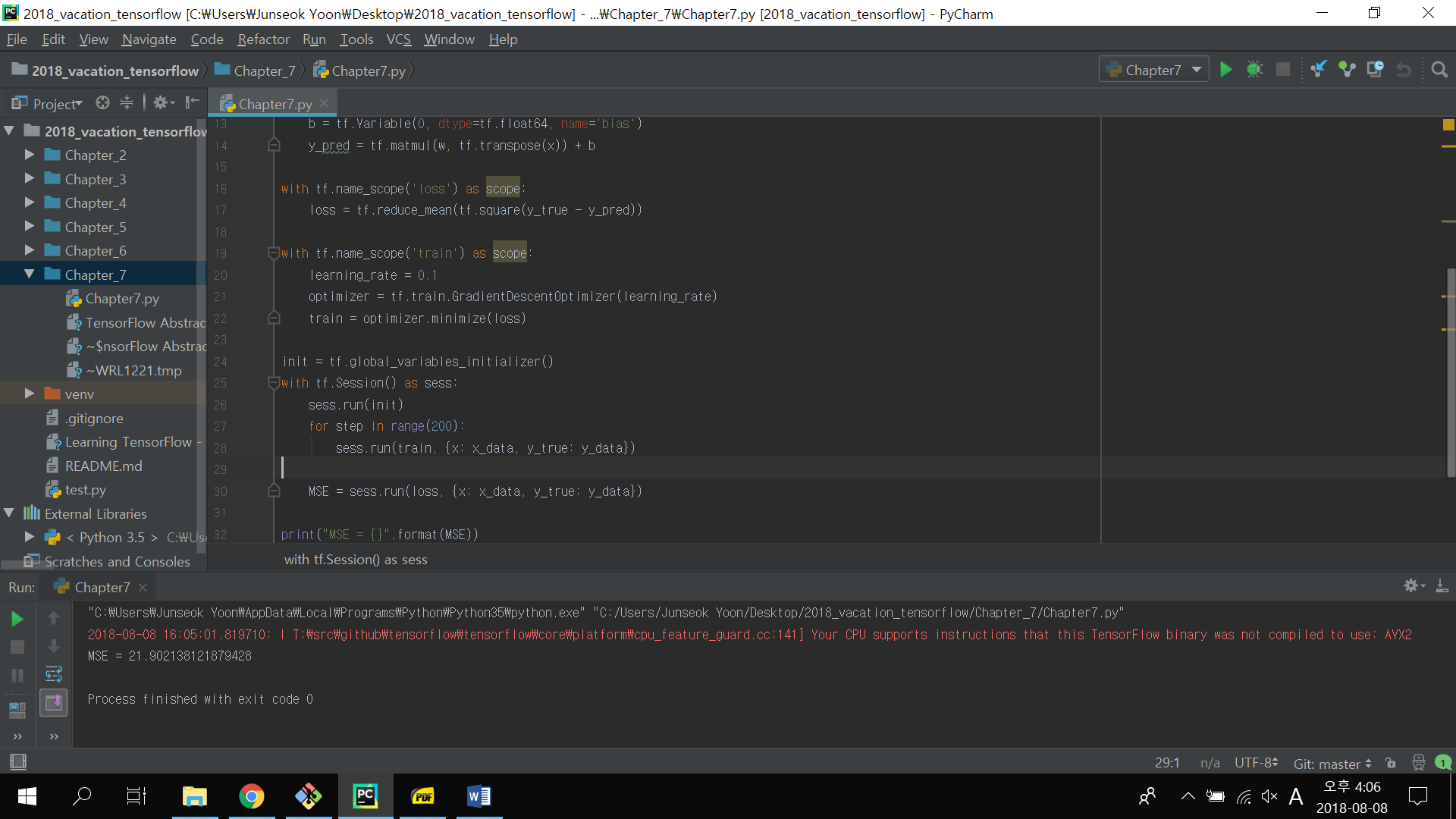
이 예제에서는 사이킷런에서 제공하는 보스턴 하우징 데이터셋을 사용할 것이다(http://bit.ly/2sXIfrX). 보스턴 하우징 데이터는 비교적 작은 데이터로 메사추세츠 주 보스턴 지역의 주택 정보가 담겨 있다. 이 데이터에는 예측변수가 13개 있다.

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.



학습 데이터에 대해 계산된 MSE

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.

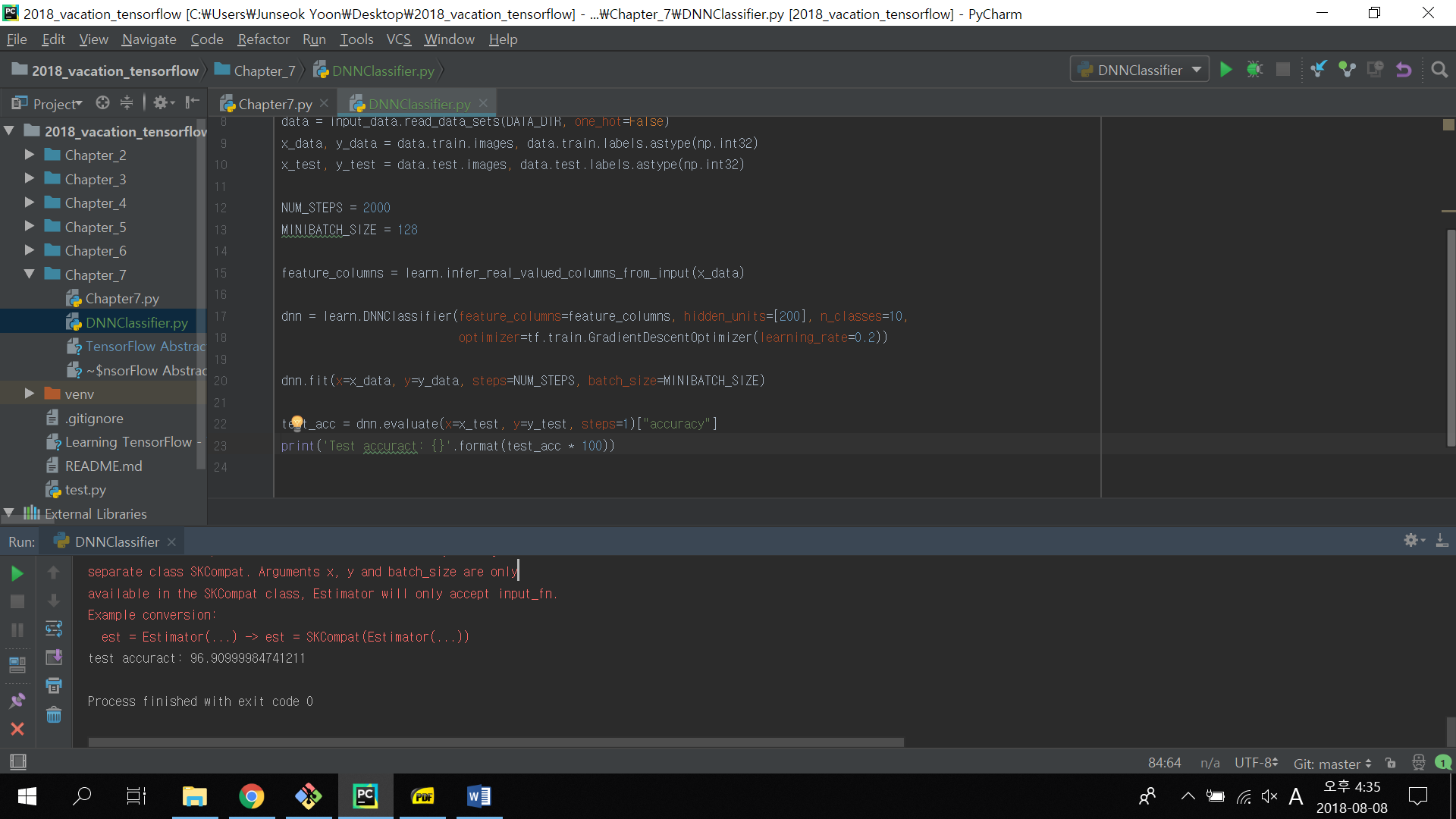
MSE = regressor.evaluate(x\_data, boston.target, steps=1)

DNN Classifier

앞의 회귀에서 사용해본 것과 같이 contrib.learn으로 즉시 사용 가능한 분류기를 구성할 수 있다. 2장에서 MNIST 데이터를 적용한 간단한 소프트맥스 분류기를 만든 적이 있다. DNNClassifier 추정자로 비슷한 일을 상당히 짧은 코드로 수행할 수 있다. (DNNClassifier.py)

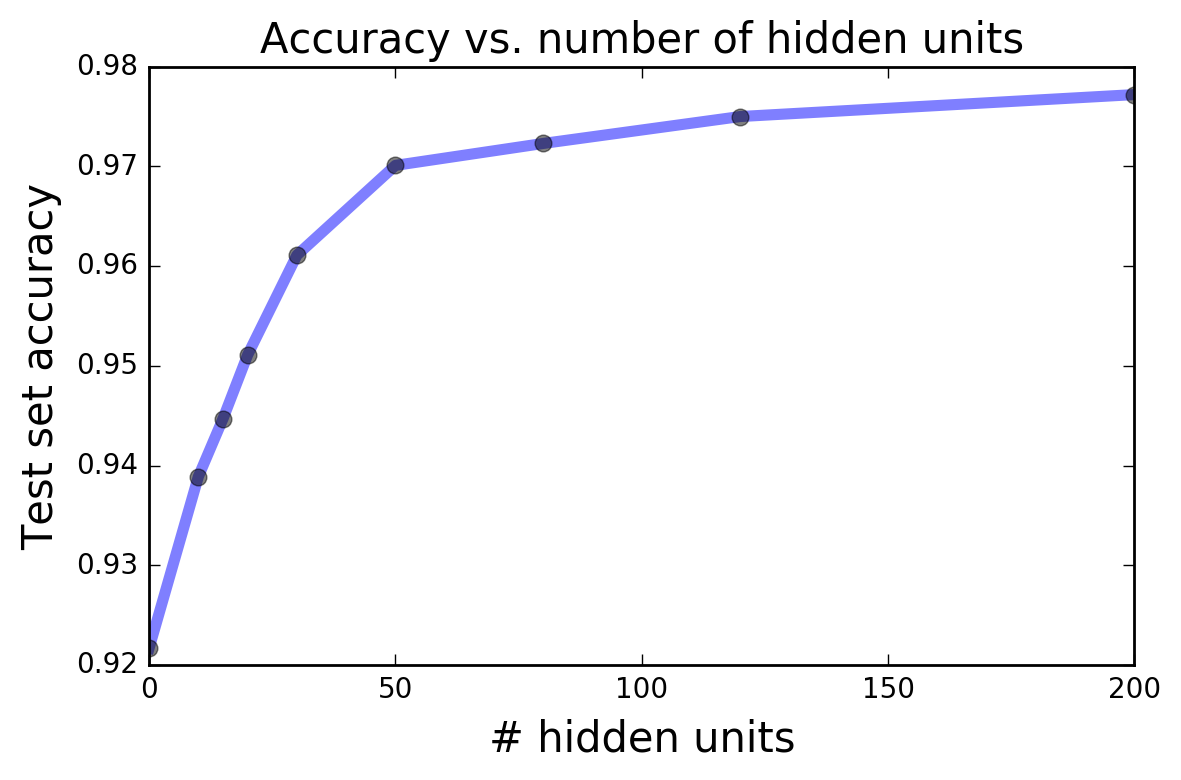
여기서는 추정자의 형식에 맞추어야 하므로 타깃을 클래스의 레이블 형식으로 전달한다는 점에 주의하자. one\_hot=False로 설정했으므로 정답 클래스에 대응되는 인덱스의 값만 1로 나오는 one-hot 형식 대신 정답 클래스에 대응되는 하나의 정수를 반환한다.

다음 단계는 이전 예제에서 했던 것과 비슷한데 모델을 정의할 때 분류의 수와 정해진 수의 유닛을 가진 은닉 계층들의 리스트를 전달한다는 점만 다르다.



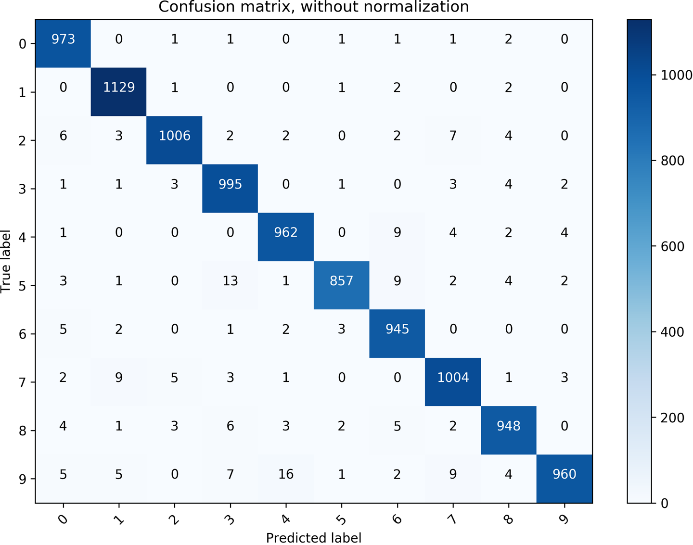
예측 결과

단 하나의 계층을 추가했던 단순한 softmax 예제(약 92%)에 비해 훨씬 테스트 정확도가 개선되었다(약 97%). 밑의 그림은 은닉 계층의 유닛의 수의 증가에 따른 모델의 정확도의 변화를 보여준다.



하나의 은닉 계층에 추가된 유닛의 수에 따른 MNIST 분류 테스트 정확도

<Estimator>.predict() 메서드를 사용하면 새로운 샘플의 클래스를 예측할 수 있다. 예측을 사용하여 어떤 클래스가 가장 잘 식별하였으며 전형적인 오류의 유형이 무엇인지 들 모델의 성능을 분석하는 방법을 설명하고자 한다. 사이킷런 라이브러리에서 혼동 행렬(confusion matrix)를 만드는 코드를 임포트한다.



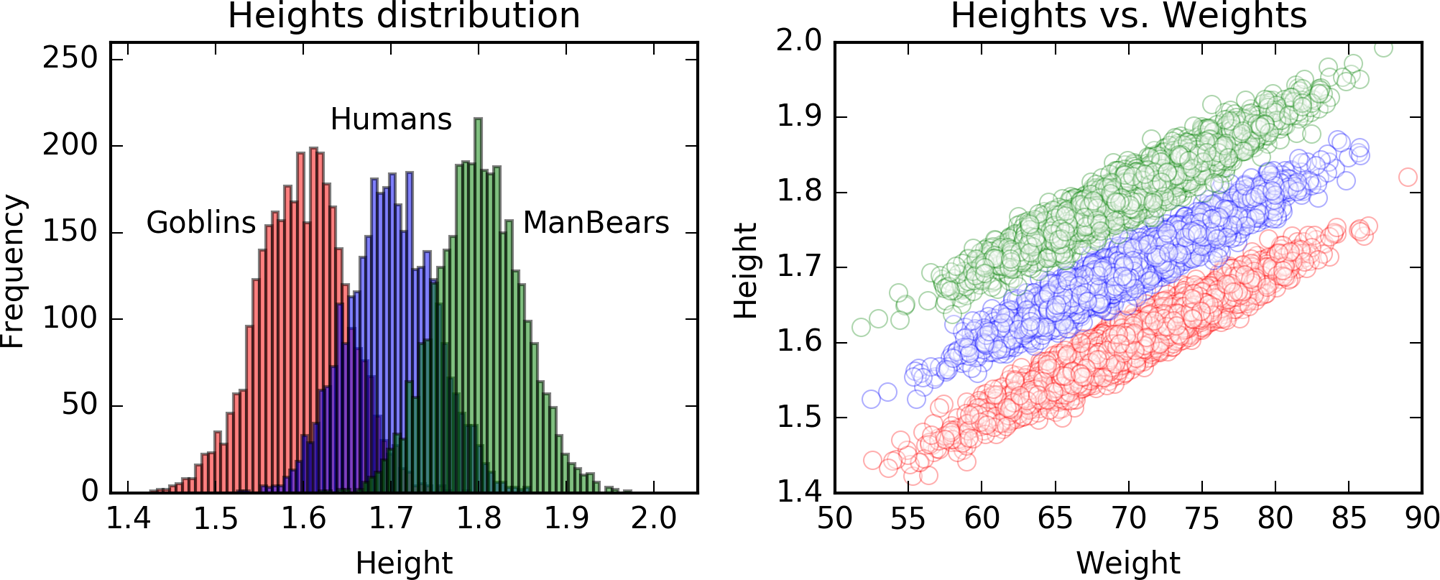
옆의 그림에서 혼동 행렬을 볼 수 있다. 행은 실제 숫자에 대응되며 열은 예측한 숫자에 대응된다. 예를 들어서 이 모델이 종종 5를 3으로, 9를 4나 7로 잘못 분류하고 있다는 사실을 알 수 있다.

FeatureColumn

contrib.learn에서 제공하는 멋진 기능 중 하나는 다른 유형의 특징들을 처리하는 것이다. 경우에 따라 까다로울 수 있는 이 작업을 쉽게 만들기 위해 contrib.learn은 Feature.Column이라는 추상화를 제공한다.

FeatureColumn을 사용하여 데이터 내의 단일 특징의 표현을 유지하면서 FeatureColumn에 정의된 다양한 변환을 수행할 수 있다. FeatureColumn은 원본 열 또는 변환의 결과로 추가된 새로운 열 중 하나일 수 있다. 여기서 제공하는 변환에는 범주형 데이터를 희소 벡터로 인코딩하여 범주형 데이터에 대한 적절하고 효율적인 표현을 생성하거나, 특징의 접점을 찾도록 특징을 결합하거나, 혹은 버킷화하는 등이 포함된다. 이 모든 작업은 특징을 하나의 의미적 단위로 조작하면서 수행될 수 있다.

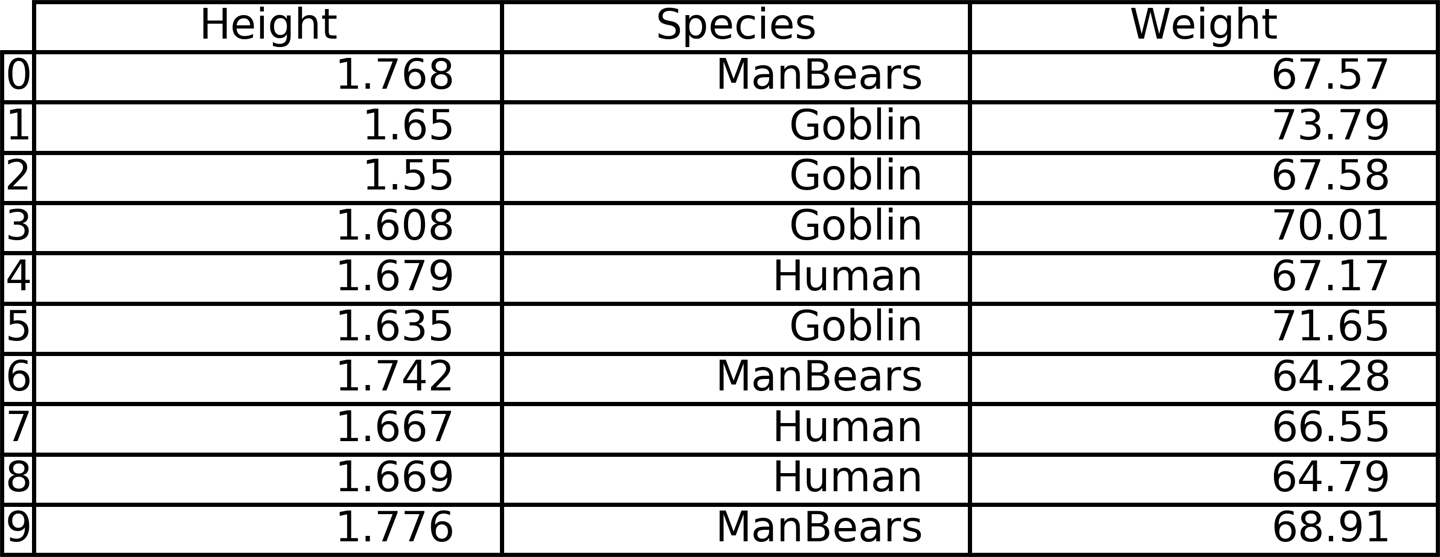
FeatureColumn 추상화를 사용하여 입력 데이터 각각의 특징의 모양과 구조를 지정한다. 예를 들어 무게와 종족 두 개의 특징으로 목표변수인 키를 예측한다고 가정해보자. 각 무게를 100으로 나누고 거기에 종에 따른 상수를 추가해 키를 생성하는 자체 합성 데이터를 만든다. 이어서 각 데이터 인스턴스에 정규분포를 따르는 노이즈를 추가한다.



키의 히스토그램과 키와 몸무게의 산포도

목표변수는 키의 숫자 넘파이 배열(height)이며, 공변량은 몸무게의 숫자 넘파이 배열(weight)와 각 종족의 이름을 나타내는 스트링 리스트(spec)이다.

pandas 라이브러리를 사용하면 데이터를 데이터프레임(table)으로 만들어 각 열의 값에 간편하게 접근할 수 있다.



키-종족-몸무게 데이터프레임의 첫 10행

먼저 각 특징의 특성을 지정한다. 몸무게에 대해서는 FeatureColumn 명령을 사용하여 이것이 연속형 변수임을 알려준다. 함수에 전달된 이름은 매우 중요한데 FeatureColumn 표현을 실제 데이터와 연결하는 데 사용된다.

Species는 범주형 변수인데 이 값의 순서는 의미가 없으므로 모델 내에서 단일 변수로 표시될 수 없다. 대신 이 변수는 범주의 수에 따른 여러 개의 변수들로 확장되어 인코딩되어야 한다. FeatureColumn이 이 작업을 할 수 있도록 다음과 같이 이것이 범주형 특징이며 각 범주의 이름을 지정하고 있음을 명시해야 한다. 그 후 추정자 클래스를 인스턴스화하고 FeatureColumn 리스트를 입력한다.

지금까지 데이터가 모델에서 표현되는 방법을 정의했다. 이어지는 모델 적합 단계에서는 실제 학습 데이터를 제공해야 한다. 보스턴 하우징 예제에서 특징은 모두 숫자이므로 이 특징들을 x\_data와 목표 데이터로 바로 입력할 수 있었다.

이 예제의 contrib.learn에는 캡슐화된 입력 함수를 추가해 사용해야 한다. 이 함수는 예측을 위한 특징 데이터와 목표 데이터 모두를 자체 형식으로 받아 tensor의 딕셔너리를 반환한다. 이들 딕셔너리에서 각 키는 FeatureColumn의 이름이며 값은 이에 대응되는 데이터가 들어 있는 tensor여야 한다. 즉 이 함수에서는 값을 tensorflow에서 사용 가능한 tensor로 바꾸어야 한다.

이 예제에서 함수는 데이터 프레임을 받아 feature\_cols 딕셔너리를 생성한 후 데이터 프레임의 각 열의 값을 해당 키의 tensor로 저장한다. 그런 다음 딕셔너리와 tensor 형태로 된 목표변수를 반환한다. 키는 FeatureColumn을 정의할 때 사용했던 이름과 일치해야 한다.

Species의 값은 해당 FeatureColumn에서 지정한 바에 따라 희소 tensor 형식으로 인코딩해야 한다. 이를 위해서 tf.SparseTensor()를 사용하는데 여기서 각 인덱스 i는 0이 아닌 값에 대응된다. 예를 들어

SparseTensor(indices=[[0, 0], [2, 1], [2, 2]], values=[2, 5, 7], dense\_shape=[3, 3])

위의 희소 텐서는 다음과 같은 텐서이다.

[[2, 0, 0]

[0, 0, 0]

[0, 5, 7]]

이 텐서를 다음과 같이 .fit() 메서드에 전달한다.

reg.fit(input\_fn=lambda: input\_fn(df), steps=50000)

여기서 input\_fn()은 앞에서 만든 함수이고 df는 데이터가 들어 있는 데이터 프레임이다. 또한 반복 횟수도 정의한다.

.fit() 메서드는 함수 객체를 받는 메서드이므로 함수의 출력이 아닌 lambda 함수의 형태로 함수를 전달한다. lambda를 사용하면 입력 인수를 전달하고 이를 객체 형식으로 유지할 수 있다. 다른 방법으로 같은 결과를 얻을 수 있지만 lambda만큼 효과적이지 않다.

이제 추정된 가중치를 통해 모델이 얼마나 잘 동작하는가를 확인하자. .get\_variable\_value() 메서드를 사용해 변수의 값을 얻을 수 있다.

Weight와 Species 모두에 대한 가중치를 구했다. Species는 범주형 변수이므로 3개의 가중치가 서로 다른 편향값으로 사용된다. 모델이 실제 가중치를 잘 추정하고 있음을 알 수 있다.

|  |  |
| --- | --- |
| 연산 | 설명 |
| layers.sparse\_column\_sith\_keys() | 범주형 값의 변환을 처리 |
| layers.sparse\_column\_with\_hash\_bucket() | 나타날 수 있는 값을 모르는 범주형 특징의 변환을 처리 |
| layers.crossed\_column() | 특징 결합을 설정 |
| layers.bucketized\_column() | 연속형 열을 범주형 열로 변환 |

Homemade CNN with contrib.learn

다음은 contrib.learn을 사용하여 자신만의 추정자를 만들어볼 차례다. 이를 위해서 먼저 네트워크를 구현하는 모델 함수와 학습 설정을 포함한 객체를 만들어야 한다. 다음 예제에서는 사용자 정의 CNN 추정자를 만들고 이를 MNIST의 데이터의 분류에 사용해본다. 우선 추정자에 사용할 함수를 만드는데, 데이터를 담고있는 입력, 연산 모드, 모델의 매개변수가 필요하다.

먼저, MNIST 데이터에서 픽셀은 벡터 형태로 연결되어 있으므로 이를 재구성해야 한다.