MNIST 고급

설정

MNIST 데이터셋 불러오기

자동으로 MNIST 데이터셋을 불러오는 스크립트가 존재한다.

from tensorflow.examples.tutorials.mnist import input\_data

mnist = input\_data.read\_data\_sets('MNIST\_data', one\_hot=True)

mnist 변소는 훈련(training), 테스트(testing), 검증(validation)데이터를 Numpy배열로 저장하는 클래스이다.

TensorFlow InteractiveSession

TensorFlow(이하 TF)는 계산을 위해 고효율의 C++ 백앤드(Backend)를 사용한다. 백앤드와 TF와의 연결을 위해 세션(Session)을 사용한다. 일반적으로 TF 프로그램은 먼저 그래프를 구성한 뒤 그래프를 세션을 통해 실행하는 방식을 따른다. 여기선 InteractiveSession(TF를 보다 유연하게 작성하게 해주는 클래스)을 사용하며, 이 클래스는 계산그래프(computation graph)를 구성하는 작업과 그 그래프를 실행하는 작업을 분리시킨다.

import tensorflow as tf

sess = tf.InteractiveSession()

계산그래프(computational Graph)

Python에서 효율적인 수치 계산을 하기 위해서 주로 NumPy와 같이 Python 외부에서 다른 언어로 된 고효율의 코드를 통해 행렬 곱셈과 같은 고비용의(expensive)연산을 수행하는 라이브러리를 이용한다. 하지만 이런 **연살결과를 일일히 Python으로 다시 불러들이는 데는 많은 오버헤드가 발생하며, 계산 과정을 gpu에 분산시키는 경우는 데이터를 이동시키는 데 드는 비용이 특히 더 커진다**. TF도 마찬가지로 고비용 연산은 Python 외부에서 실행한다. **하지만 오버헤드 문제를 피하기위해 고비용 연산을 Python에서 독립적으로 실행하는 대신, TF는 상호작용하는 연산을 그래프로 묶어 그 전체를 Python 바깥에서 실행시키는 방법을 사용**한다.

소프트맥스 회귀 모델 구성

플레이스홀더(Placeholder)

계산 그래프를 구성하기 위해 먼저 입력될 이미지와 각각의 출력 클래스에 해당하는 노드를 생성

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

y\_ = tf.placeholder(tf.float32, shape=[None, 10])

x, y\_는 빈공간, 나중에 placeholder가 값을 넣어줄 자리

입력될 이미지들 x는 부동소스점 실수(float)값들의 2d텐서이다. 위 코드에서 shape크기에 해당하는 첫번째 차원 크기의 None은 크기를 여기서 정하지않음(어떤 배치라도 가능함). 출력 클래스인 y\_또한 2dㄹ텐서, 각 uf은 해당하는 MNIST 이미지의 숫자 클래스를 10차원 one-hot백터로 나타냄

tf.placeholder에 shape 매개변수가 필수는 아닙니다. 하지만, 이를 명시해 줌으로써 TensorFlow가 잘못된 텐서 구조(shape)에 따른 오류를 자동으로 잡아낼 수 있게 됨

변수(Variable)

모델에 사용할 가중치(weight) w 와 편항(bias) b를 정의한다. TF에선 이들을 다루기 위해 Variable을 제공하는데, TF의 계산 그래프 안에 있는 값이다. 이들은 계산에 사용될 뿐만 아니라, 계산에 의해 변경될 수도 있다.

W = tf.Variable(tf.zeros([784,10]))

b = tf.Variable(tf.zeros([10]))

tf.Variable을 사용할 때에는 변수 초기값을 지정해줘야 된다. 위의 경우는 w, b 모두 0으로만 구성된 텐서로 초기화 된다. W는 784 x 10행렬(이미지 벡터 크기784, 숫자 클래스 10)이며, b는 10차원 벡터이다. Vriable들은 세션이 시작되기 전에 초기화 되어야한다.

sess.run(tf.global\_variables\_initializer())

이 코드는 모든 Vriable들 각각에 대해 미리 지정된 초기값(0으로 구성된 텐서)를 넣어주는 역할을 한다.

클래스 예측 및 비용 함수(Cost Function)

벡터화된 입력 이미지인 x를 가중치 행렬인 w와 곱하고, 여기에 편향 b를 더한 뒤 각각의 클래스에 대한 소프트맥스 함수의 결과를 계산하면 된다.

y = tf.nn.softmax(tf.matmul(x,W) + b)

모델 훈련 과정에서 최소화될 비용 함수(cost function)또한 간단하게 도입할 수 있다. 여기서 사용될 비용 함수는 실제 클래스와 모델의 예측 결과 간 크로스 엔트로피(cross-entropy)함수이다.

cross\_entropy = tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(y), reduction\_indices=[1]))

여기서 tf.reduce\_sum은 모든 클래스에 대해 결과를 합하는 함수, tf.reduce\_mean은 사용된 이미지들 각각에서 계산된 합의 평균을 구하는 함수이다.

모델 훈련

TF모델을 훈련시켜야 된다. TF에 전체 계산 그래프의 정보가 입력되어 있으므로, 라이브러리가 자동으로 미분을 통해 각각의 변수에 대한 비용 함수를 기울기(gradient)를 계산한다.

train\_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross\_entropy)

위 코드와 같이 학습 속도 0.5의 경사 하강법(STEEPEST GRADIENT DESCENT)알고리즘을 사용하여 크로스 엔트로피를 최소하 한다. 이 코드 한줄에서 실제로 하는 것은 계산 그래프에 기울기를 계산하고 얼마나 매개변수를 변경해야 할지 계산, 그리고 매개변수를 변경하는 새로운 계산들을 추가하는 것이다. 반환된 TRAIN\_STEP은 실행되었을 때 경사 하강법을 통해 각각의 매개변수를 변화시키게 된다. 따라서, 모델을 훈련시키려면 이 TRAIN\_STEP을 반복적으로 실행하면 된다.

for i in range(1000):

batch = mnist.train.next\_batch(50)

train\_step.run(feed\_dict={x: batch[0], y\_: batch[1]})

각 훈련 단계(ITERATION)에서, 50개의 훈련 샘플이 추출된다. 그리고 train\_step을 실행하며, feed\_dict를 통해 PLACEHOLDER 텐서인 X와 Y\_에 훈련 샘플을 넣어준다. 참고로 feed\_dict는 placeholder외에도 계산 그래프 안에 어떤 텐서라도 변경할 수 있다.

모델 평가

먼저 모델이 정확한 레이블을 예측했는지 확인해 볼 것이다. tf.argmax함수는 텐서의 한 차원을 따라 가장 큰 값의 인덱스를 반환한다. Ex) tf.argmax(y,1)은 모델이 입력을 받고 가장 그럴듯하다고 생각한 레이블이고, tf.argmax(y\_,1)은 실제 레이블이다. 이제 tf.equal함수를 사용해 두 레이블이 일치하는지 확인할 수 있다.

correct\_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y\_,1))

이 코드는 불리언으로 이루어진 리스트를 반환한다. 전체에서 얼마나 맞았는지를 확인하려면, 불리언을 부동 소수점 실수로 형변환하여 리스트의 평균을 구하면된다.(ex: [True, False, True, False] 였다면 이는 형변환을 통해 [1, 0, 1, 0]이 되고 평균이 0.5가 된다.)

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

이제 feed\_dict로 mnist.test를 전달하여 테스트 데이터셋에 대한 예측 정확도를 확인할 수있다. 대략 92%정도가 나온다.

print(accuracy.eval(feed\_dict={x: mnist.test.images, y\_: mnist.test.labels}))

다중 계층 합성 신경망

MNIST데이터에서 91~2%의 정확도는 그다지 좋은 겨로가가 아니다. 이번엔 정확도를 높이기 위해 다중 계층 합성 신경망(convolutional neural network)이라는 약간 복잡한 모형을 사용할 것이다. 이를 통해 99.2%정도의 정확도를 얻을 수 있다.

가중치 초기화

합성곱 신경망 모델을 구성하기 위해서는 많은 수의 가중치(weight, w)와 편향(bias, b)를 사용하게 된다. 대칭성을 깨뜨리고 기울기(gradient)가 0이 되는 것을 방지하기 위해 가중치에 약간의 잡음을 주어 초기화한다. 또한 모델에 ReLU 뉴런이 포함되므로, “죽은 뉴런”을 방지하기 위해 편향을 작은 양수(0.1)로 초기화한다. 매번 모델을 만들 때마다 반복하는 대신, 아래 코드와 같이 이러한 일을 해주는 함수 2개를 생성한다.

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)

합성곱(Convolution)과 폴링(Pooling)

TF는 합성곱과 폴링 계층(layer)을 유연하게 다룰 수 있도록 해준다. 경계의 패딩(padding)과 스트라이드(stride)에 대해 다양한 선택을 할 수 있다. 이번 예시는 스트라이드를 1로, 출력 크기가 입력과 같게 되도록 0으로 패딩하도록 설정한다. 풀링은 2x2 크기의 맥스 풀링을 적용한다. 마찬가지로 코드를 간단하게 하기 위해 합성곱과 풀링을 위한 함수를 아래 코드와 같이 생성한다.

def conv2d(x, W):

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

def max\_pool\_2x2(x):

return tf.nn.max\_pool(x, ksize=[1, 2, 2, 1],

strides=[1, 2, 2, 1], padding='SAME')

첫 번째 합성곱 계층

이 층은 합성곱 계층과 맥스 폴링 계층으로 이루어진다. 합성곱 계층에서는 5x5의 윈도우(patch) 크기를 가지는 32개의 필터를 사용하며, 따라서 구조(shape)가 [5,5,1,32]인 가중치 텐서를 정의해야 한다. 처음 두 개의 차원은 윈도우의 크기, 세 번째는 입력 채널의 수, 마지막은 출력 채널의 수(얼마나 많은 특징을 사용할 것인가)를 나타낸다. 또한 각 출력 채널에 대한 편향을 정의해야 한다.

W\_conv1 = weight\_variable([5, 5, 1, 32])

b\_conv1 = bias\_variable([32])

앞서 만든 함수를 사용한다. 이 계층에 이미지를 입력하려면 먼저 x를 4D텐서로 reshape해야한다. 두번째와 세번째 차원은 이미지의 가로와 세로길이, 그로기 마지막 차원은 컬러 채널의 수를 나타낸다.

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

이제 x\_image와 가중치 텐서의 합성곱을 적용하고, 편향을 더한 뒤 ReLU함수를 적용한다. 출력 값을 구하기 위해 마지막으로 맥스 풀링을 적용한다.

h\_conv1 = tf.nn.relu(conv2d(x\_image, W\_conv1) + b\_conv1)

h\_pool1 = max\_pool\_2x2(h\_conv1)

두 번째 합성곱 계층

심층 신경말을 구성하기 위해서 앞서 만든 것과 비슷한 계층을 쌓아올릴 수 있다. 여기서는 두 번째 합성곱 계층이 5x5윈도우에 64개의 필터를 가진다.

W\_conv2 = weight\_variable([5, 5, 32, 64])

b\_conv2 = bias\_variable([64])

h\_conv2 = tf.nn.relu(conv2d(h\_pool1, W\_conv2) + b\_conv2)

h\_pool2 = max\_pool\_2x2(h\_conv2)

완전 연결 계층(Fully-Connected Layer)

두 번째 계층을 거친 뒤 이미지 크기는 7x7로 줄어들었다. 이제 여기에 1024개의 뉴러느로 연결되는 완전 연결 계층을 구성한다. 이를 위해서 7x7이미지 배열을 reshape해야 하며, 완전 연결 계층에 맞는 가중치 행렬과 편향 행렬을 구성한다. 최종적으로 완전 연결 계층의 끝에 ReLU함수를 적용한다.

W\_fc1 = weight\_variable([7 \* 7 \* 64, 1024])

b\_fc1 = bias\_variable([1024])

h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7\*7\*64])

h\_fc1 = tf.nn.relu(tf.matmul(h\_pool2\_flat, W\_fc1) + b\_fc1)

드롭아웃(Dropout)

오버피팅(overfitting) 되는 것을 방지하기 위해, 드롭아웃을 적용한다. 뉴런이 드롭아웃 되지않을 확률을 저장하는 placeholder를 만든다. 이렇게 하면 나중에 드롭아웃이 훈련 과정에는 적용되고, 테스트 과정에는 적용되지 않도록 설정할 수 있다. TF의 tf.nn.dropout 함수는 뉴런의 출력을 자동으로 스케일링(scaling)하므로, 추가로 스케일링 할 필요 없이 그냥 드롭아웃을 적용할 수 있다.

keep\_prob = tf.placeholder(tf.float32)

h\_fc1\_drop = tf.nn.dropout(h\_fc1, keep\_prob)

최종 소프트맥스 계층

마지막으로 위에서 단일 계층 소프트맥스 회귀 모델을 구성할 때와 비슷하게 소프트맥스 계층을 추가한다.

W\_fc2 = weight\_variable([1024, 10])

b\_fc2 = bias\_variable([10])

y\_conv=tf.nn.softmax(tf.matmul(h\_fc1\_drop, W\_fc2) + b\_fc2)

모델의 훈련 및 평가

훈련평가모델은 기존 단일 계층 모델과 거의 같다. 차이가 있다면 경사하강법 알고리즘 대신 더 복잡한 ADAM최적화 알고리즘ㅡㄹ 사용하는 것이다. 또한 드롭아웃 확률을 설정하는 추가 변수인 keep\_prob을 feed\_dict인수를 통해 전달한다. 아래 코드는 훈련 과정에서 100회 반복 시마다 로그를 작성한다.

cross\_entropy = tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(y\_conv), reduction\_indices=[1]))

train\_step = tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy)

correct\_prediction = tf.equal(tf.argmax(y\_conv,1), tf.argmax(y\_,1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

sess.run(tf.global\_variables\_initializer())

for i in range(20000):

batch = mnist.train.next\_batch(50)

if i%100 == 0:

train\_accuracy = accuracy.eval(feed\_dict={

x:batch[0], y\_: batch[1], keep\_prob: 1.0})

print("step %d, training accuracy %g"%(i, train\_accuracy))

train\_step.run(feed\_dict={x: batch[0], y\_: batch[1], keep\_prob: 0.5})

print("test accuracy %g"%accuracy.eval(feed\_dict={

x: mnist.test.images, y\_: mnist.test.labels, keep\_prob: 1.0}))