MNIST 고급 정리

# TensorFlow

* 큰 규모의 수치 계산에 적합한 강력한 라이브러리
  + 가장 강력한 힘을 발휘하는 작업 중 하나는 심층 신경망을 구성하고 학습시키는 것

# TUTORIAL

* MNIST 데이터를 분류하는 심층 합성곱(convolutional) 신경망을 구성하면서, TensorFlow에서 신경망 모델을 구성하는 기본 블록에 대해서 알아볼 것

# MNIST 데이터셋 불러오기

* 자동으로 MNIST 데이터셋을 다운받은 뒤 불러오는 스크립트
* from tensorflow.examples.tutorials.mnist import input\_data  
  mnist = input\_data.read\_data\_sets('MNIST\_data', one\_hot=True)
* 해당 스크립트를 import하여 실행 시, 현재 디렉토리 하위에 ‘MNIST\_data’ 폴더를 생성하여 자동으로 데이터 파일을 저장할 것
* 위에서 mnist는 훈련(Training), 테스트(Testing), 그리고 검증(validation) 데이터를 NumPy 배열로 저장하는 클래스

# TensorFlow InteractiveSession 시작하기

* TensorFlow는 계산을 위해 고효율의 C++ 백엔드(backend)를 사용
* 이 백엔드와의 연결을 위해 TensorFlow는 세션(session)을 사용
* 일반적으로 TensorFlow 프로그램은 먼저 그래프를 구성하고, 그 이후 그래프를 세션을 통해 실행하는 방식

## InteractiveSession Class

* 계산 그래프(computation graph)를 구성하는 작업과 그 그래프를 실행하는 작업을 분리
* 즉, 이 클래스를 쓰지 않는다면,  세션을 시작하여 그래프를 실행하기 전에 이미 전체 계산 그래프가 구성되어 있어야 함.
* import tensorflow as tf  
  sess = tf.InteractiveSession()

### 계산 그래프 (Computational Graph)

* Python에서 효율적인 수치 계산을 하기 위해서, 주로 NumPy와 같이 Python 외부에서 다른 언어로 된 고효율의 코드를 통해 행렬 곱셈과 같은 고 비용의 연산을 수행하는 라이브러리 이용
  + 이렇게 하면 연산 결과를 일일이 Python으로 다시 불러들이는 데 많은 오버헤드 발생
  + 특히 계산 과정을 여러 GPU에 분산시키는 경우, 데이터를 이동시키는 데 드는 비용 매우 커짐.
* TensorFlow도 고비용의 연산은 Python 외부에서 실행
  + 위와 같은 오버헤드 문제를 피하기 위해 각각의 고 비용 연산을 Python에서 독립적으로 실행하는 대신, TensorFlow는 상호작용하는 연산을 그래프로 묶어 그 전체를 Python 바깥에서 실행시키는 방법 사용
  + Theano나 Torch와 같은 라이브러리에서 활용되는 방법과 비슷
* Python에서 작성하는 코드의 역할은 외부의 계산 그래프를 구성하고, 이 계산 그래프의 어떤 부분이 실행되어야 하는지 지시하는 것.

# 소프트맥스 회귀 모델 구성 (softmax regression model)

## 플레이스홀더 (Placeholder)

* 계산 그래프를 구성하기 위해, 먼저 입력될 이미지와 각각의 출력 클래스에 해당하는 노드 생성
* x = tf.placeholder(tf.float32, shape=[None, 784])  
  y\_ = tf.placeholder(tf.float32, shape=[None, 10])
* x와 y\_는 특정한 값이 부여된 것이 아니라 나중에 TensorFlow가 계산을 실행할 때 값을 넣어 줄 자리인 placeholder임
* 입력될 이미지들 x는 부동 소수점 실수(float)값들의 2D 텐서
  + shape에 [None, 784]
    - 784는 28\*28의 크기를 가지는 MNIST 이미지를 한 줄로 펼친 크기에 해당
    - 배치(batch)의 크기에 해당하는 첫 번째 차원 크기의 None은 크기를 여기서 정하지 않는다(어떤 배치 크기라도 가능하다)는 것 의미
* 출력 클래스인 y\_ 또한 2D 텐서
  + 각 열은 해당하는 MNIST 이미지의 숫자 클래스를 10차원 one-shot 벡터로 나타냄
* tf.placeholder에 shape 매개변수 필수는 아님
  + but, 이를 명시해 줌으로써 TensorFlow가 잘못된 텐서 구조(shape)에 따른 오류를 자동으로 잡아낼 수 있게 됨.

## 변수 (Variable)

* 모델에 사용할 가중치(weight)  w와 편향(bias) b를 정의
* 추가적인 입력으로 대할 수도 있겠지만, TensorFlow는 이러한 변수들을 다루기 위해 Variable 제공
* Variable
  + TensorFlow의 계산 그래프 안에 있는 값
  + 이들은 계산에 사용될 수 있을 뿐만 아니라, 계산에 의해 변경될 수도 있음
  + 따라서 머신 러닝에 활용되는 모델 매개변수는 주로 Variable들로 구성
* W = tf.Variable(tf.zeros([784,10]))  
  b = tf.Variable(tf.zeros([10]))
* tf.Variable을 사용할 때에는 변수의 초기 값을 지정해 주어야 함
* 위 코드의 경우, w와 b 모두 0으로만 구성된 텐서로 초기화됨
  + w는 784\*10 행렬(입력 이미지 벡터의 크기가 784, 출력 숫자 클래스가 10개)
  + b는 10차원 벡터
* Variable들은 세션이 시작되기 전에 초기화되어야 함
* sess.run(tf.global\_variables\_initializer())
* 위 코드는 모든 Variable들 각각에 미리 지정된 초기값(위에서 지정된 0으로만 구성된 텐서)을 넣어 주는 역할을 함.

## 클래스 예측 및 비용 함수 (Cost Function)

* 이제 회귀 모델을 도입할 수 있음
* y = tf.nn.softmax(tf.matmul(x,W) + b)
* 벡터화된 입력 이미지인 x를 가중치 행렬인 w와 곱하고, 여기에 편향 b를 더한 뒤, 각각의 클래스에 대한 소프트맥스 함수의 결과를 계산
* cross\_entropy = tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(y), reduction\_indices=[1]))
* 모델 훈련 과정에서 최소화될 비용 함수(cost function) 또한 간단하게 도입 가능
  + 비용 함수는 실제 클래스와 모델의 예측 결과 간 크로스 엔트로피(cross-entropy) 함수
* tf.reduce\_sum은 모든 클래스에 대해 결과를 합하는 함수,
* tf.reduce\_mean은 사용될 이미지들 각각에서 계산된 합의 평균을 구하는 함수

# 모델 훈련시키기

* TensorFlow에 전체 계산 그래프의 정보가 입력되어 있음
* 라이브러리가 자동으로 미분을 통해 각각의 변수에 대한 비용 함수의 기울기(gradient)를 계산
* TensorFlow는 다양한 내장된 최적화 알고리즘 소유
* 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 엔트로피 최소화할 것
* TensorFlow가 위 코드에서 실제로 하는 것은 계산 그래프의 기울기를 계산, 매개변수를 변경해야 할 지 계산, 매개변수를 변경하는 새로운 계산들 추가
* 반환된 train\_step은 실행되었을 때 경사 하강법을 통해 각각의 매개변수를 변화시킴
* 따라서, 모델을 훈련시키려면 이 train\_step을 반복해서 실행하면 됨!
* for i in range(1000):  
    batch = mnist.train.next\_batch(50)  
    train\_step.run(feed\_dict={x: batch[0], y\_: batch[1]})
* 각각의 훈련 단계(iteration)에서, 50개의 훈련 샘플이 추출됨
* 그리고 train\_step을 실행하며 feed\_dict를 통해 placeholder 텐서인 x와 y\_에 훈련 샘플을 넣어줌
  + feed\_dict는 placeholder 외에도 계산 그래프 안의 어떤 텐서든 변경 가능

## 모델 평가하기

* correct\_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y\_,1))
* tf.argmax 함수는 텐서의 한 차원을 따라 가장 큰 값의 인덱스를 반환
* EX) tf.argmax(y,1)은 모델이 입력을 받고 가장 그럴듯하다고 생각한 레이블, if.argmax(y\_, 1)은 실제 레이블
  + 이제 tf.equal 함수를 사용해 두 레이블이 일치하는지 다음과 같이 확인 가능
  + correct\_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y\_,1))
  + 불리언으로 이루어진 리스트를 반환
  + 전체에서 얼마나 맞았는지를 확인하려면 불리언을 부동 소수점 실수로 형변환하여 리스트의 평균을 구하면 됨
  + EX) 결과가 [True, False, True, True]였다면 이는 형변환을 통해 [1, 0, 1, 1]이 되고, 평균인 0.75가 예측 결과의 정확도가 됨
  + accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))
  + 이제 아래와 같이 feed\_dict로 mnist.test를 전달하여 테스트 데이터셋에 대한 예측 정확도를 확인할 수 있음
  + print(accuracy.eval(feed\_dict={x: mnist.test.images, y\_: mnist.test.labels}))
  + 대략 92%정도의 정확도 얻어짐

# 다중 계층 합성곱 신경망

## 가중치 초기화

* 합성곱 신경망 모델을 구성하기 위해서는 많은 수의 가중치와 편향을 사용하게 됨
* 대칭성을 깨뜨리고 기울기가 0이 되는 것을 방지하기 위해, 가중치에 약간의 잡음을 주어 초기화함.
* 또한, 모델에 ReLU 뉴런이 포함되므로, “죽은 뉴런”을 방지하기 위해 편향을 작은 양수(0.1)로 초기화함
* 매번 모델을 만들 때마다 반복하는 대신, 아래 코드와 같이 이러한 일을 해주는 함수 두 개를 생성
* 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 풀링 (Pooling)

* 경계의 패딩(padding)과 스트라이드(stride)에 대해 다양한 선택을 할 수 있음
* EX) stride = 1로, 출력 크기가 입력과 같게 되도록 0으로 패딩하도록 설정

풀링은 2\*2크기의 맥스 풀링을 적용

* 합성곱과 풀링을 위한 함수
* 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

* 첫 번째 계층은 합성곱 계층과 맥스 풀링 계층으로 구성됨
* 합성곱 계층 에서는 5\*5의 윈도우(patch) 크기를 가지는 32개의 필터를 사용
  + 구조(shape)가 [5, 5, 1, 32]인 가중치 텐서를 정의해야 함
  + 처음 두 개의 차원은 윈도우의 크기, 세 번째는 입력 채널의 수, 마지막은 출력 채널의 수(즉, 얼마나 많은 특징을 사용할 것인가)를 나타냄
  + 또한, 각각의 출력 채널에 대한 편향을 정의
  + W\_conv1 = weight\_variable([5, 5, 1, 32])  
    b\_conv1 = bias\_variable([32])
  + 이 계층에 이미지를 입력하려면 먼저 x를 4D텐서로 reshape해야 함
  + 두 번째와 세 번째 차원은 이미지의 가로와 세로 길이, 그리고 마지막 차원은 컬러 채널의 수를 나타냄
  + x\_image = tf.reshape(x, [-1,28,28,1])
  + x\_image와 가중치 텐서에 합성곱을 적용하고, 편향을 더한 뒤 ReLU 함수를 적용
  + 출력 값을 구하기 위해 마지막으로 맥스 풀링 적용
  + h\_conv1 = tf.nn.relu(conv2d(x\_image, W\_conv1) + b\_conv1)  
    h\_pool1 = max\_pool\_2x2(h\_conv1)

## 두 번째 합성곱 계층

* 심층 신경망을 구성하기 위해서, 앞에서 만든 것과 비슷한 계층을 쌓아올릴 수 있음
* 여기서는 두 번째 합성곱 계층이 5\*5 윈도우에 64개의 필터를 가짐
* 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 7\*7로 줄어듬
* 이제 여기에 1024개의 뉴런으로 연결되는 완전 연결 계층을 구성
* 이를 위해 7\*7 이미지의 배열을 reshape해야 하며, 완전 연결 계층에 맞는 가중치 행렬과 편향 행렬을 구성함
* 최종적으로 완전 연결 계층의 끝에 ReLU 함수를 적용
* 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)

### 드롭 아웃 (Dropout)

* 오버피팅(overfitting) 되는 것을 방지하기 위해 드롭아웃을 적용
* 뉴럽이 드롭아웃되지 않을 확률을 저장하는 placeholder를 만듬
  + 이렇게 하면 나중에 드롭아웃이 훈련 과정에는 적용되고, 테스트 과정에서는 적용되지 않도록 설정할 수 있음
* TensorFlow의 tf.nn.dropout 함수는 뉴런의 출력을 자동으로 스케일링(scaling)하므로 추가로 스케일링 할 필요 없이 그냥 드롭아웃을 적용할 수 있음
* keep\_prob = tf.placeholder(tf.float32)  
  h\_fc1\_drop = tf.nn.dropout(h\_fc1, keep\_prob)

## 최종 소프트 맥스 계층

* 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

* 이렇게 훈련된 모델은 훈련 및 평가 또한 위의 단일 계층 모델과 거의 같음
* 차이가 있다면 이번에는 경사 하강법 알고리즘 대신 더 복잡한 ADAM 최적화 알고리즘을 사용
* 또한, 드롭아웃 확률을 설정하는 추가 변수인 keep\_prob을 feed\_dict 인수를 통해 전달
* 아래 코드는 훈련 과정에서 100회 반복 시마다 로그를 작성
* 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}))
* 코드를 실행시켜서 얻은 최종 정확도는 약 99.2%가 됨