MNIST 초급 정리

**MNIST**

* 프로그래밍을 어떻게 하는지 배울 때, 언제나 “Hello World.”를 가장 먼저 출력해보듯이 머신러닝에선 MNIST를 가장 먼저 다룸.
* 간단한 컴퓨터 비전 데이터셋, 손으로 쓰여진 이미지로 구성됨.
* 또한, 이것은 어떤 정보인지 알려주는 각 이미지에 관한 라벨을 포함

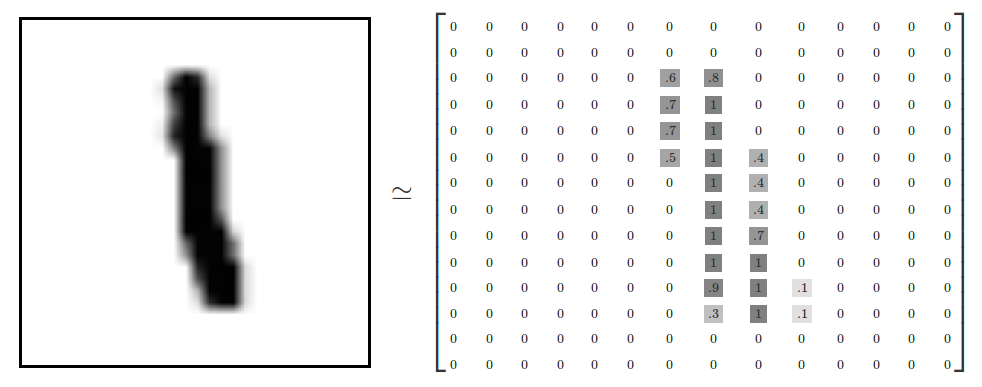
**Tutorial**

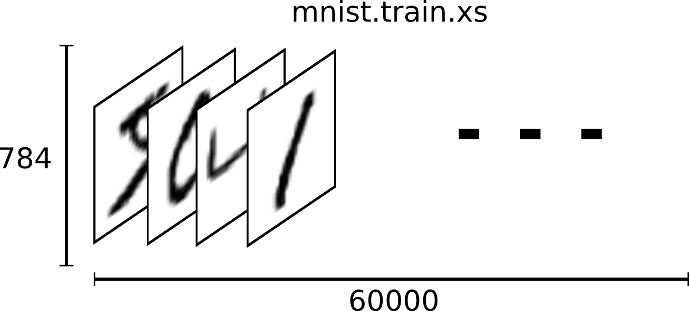
* 모델이 이미지를 보고 어떤 숫자인지 예측하는 모델을 훈련
* 목적 - 가장 최신의 성능을 발휘하는 정교한 시스템을 만드는 것이 소프트맥스 회귀라고 불리는 아주 간단한 모델로 시작

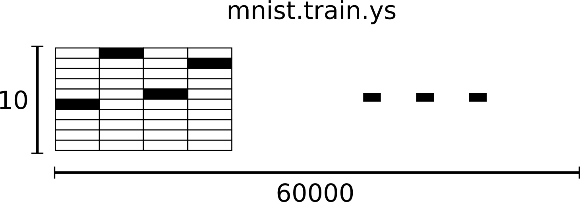
**MNIST DATA SET ( MNIST 데이터 셋 )**

* MNIST데이터셋은 파이썬에서 코드로써 다운받고 설치하는 것을 가능케 한다. 그 코드는,
* from tensorflow.examples.tutorials.mnist import input\_data  
  mnist = input\_data.read\_data\_sets("MNIST\_data/", one\_hot=True)
* 다운로드된 데이터는 55000개의 학습 데이터, 10000개의 테스트 데이터, 그리고 5000개의 검증 데이터 이렇게 세 부분으로 나뉘는데 데이터가 이렇게 나뉜다는 것은 매우 중요함
  + 우리가 학습시키지 않는 데이터를 통해, 우리가 학습한 것이 정말로 일반화되었다고 확신할 수 있음.
* MINIST 데이터셋은 두 부분으로 나뉨.
  + 손으로 쓴 숫자와 그에 따른 라벨
* 우리는 이미지를 “xs”라고 부르고, 라벨은  “ys”라고 부름
* 학습 데이터셋과 테스트 데이터셋은 둘다 xs와 ys를 가짐
* EX) 학습이미지 : mnist.train.images

      학습 라벨 : mnist.train.labels

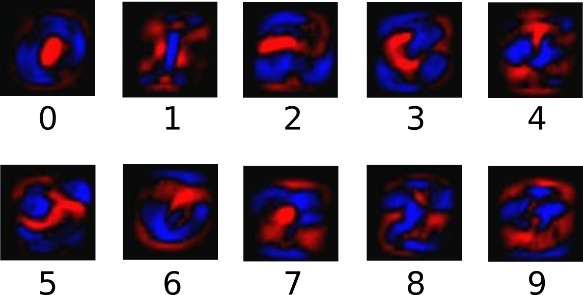
* 각 이미지는 28\*28 픽셀. 이를 숫자의 큰 배열로 해석.
* 이 배열을 펼쳐서 28\*28 = 784 개의 벡터로 만들 수 있음
* 이미지들 간에 일괄적으로 처리하기만 한다면, 배열을 어떻게 펼치던지 상관X
* 이러한 관점에서, MNIST 이미지는 매우 호화스러운 구조를 가진, 단지 784차원 벡터 공간에 있는 여러 개의 데이터일 뿐.
* 데이터를 펼친 결과로 mnist.train.images는 [55000, 784]의 형태를 가진 텐서(n차원 배열)가 됨
* 첫 번째 차원은 이미지를 가리키며, 두 번째 차원은 각 이미지의 픽셀을 가리킴
* 텐서의 모든 성분은 특성 이미지의 특정 픽셀을 특정하는 0과 1사이의 픽셀 강도임.

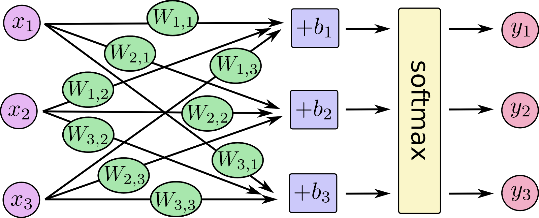
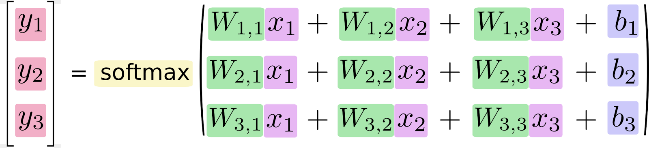
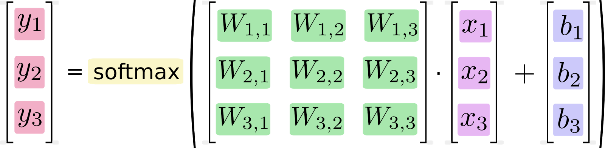


* MNIST에서 각각에 대응하는 라벨을 0과 9사이의 숫자, 각 이미지가 어떤 숫자인지를 말해줌.
* one-hot 벡터는 단 하나의 차원에서만 1이고, 나머지 차원에서는 0인 벡터.
  + 이 경우, n번째 숫자는 n번째 차원이 1인 벡터로 표현
  + EX) 3은 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] 임.
  + 결과적으로 mnist.train.labels는 [55000, 10]의 모양을 같은 실수 배열이 됨

**소프트맥스 회귀 (softmax regression)**

* 우리는 MNIST의 각 이미지가 0부터 9 사이의 손으로 쓴 숫자라는 것을 알고 있음.
* 따라서 각 이미지는 10가지의 경우의 수 중 하나에 해당
* 소프트맥스 회귀를 사용하기에 아주 적절한 상황예시
  + 만드는 모델이 9가 쓰여져 있는 이미지를 보고 이 이미지가 80%의 확률로 9라고 추측하지만, 8일 확률도 5% 있다고 계산할 수도 있음. 또한 그 외의 다른 숫자일 확률도 조금씩 있을 수 있음.
  + 만약 어떤 것이 서로 다른 여러 항목 중 하나일 확률을 계산하고자 할 때, 적합
    - 소프트맥스는 각 값이 0과 1 사이의 값으로 이루어지고, 각 값을 모두 합하면 1이 되는 목록을 제공하기 때문
    - 게다가 나중에 더 복잡한 모델을 트레이닝 할 때에도, 마지막 단계는 소프트맥스 레이어가 될 것
* 소프트맥스 회귀는 두 단계로 이루어짐.
  + 입력한 데이터가 각 클래스에 속한다는 증거(evidence)를 수치적으로 계산하고, 그 뒤엔 계산한 값을 확률로 변환
* 각 이미지가 특정 클래스에 속하는지 계산하기 위해서는 각 픽셀의 어두운 정도(intensity)를 가중치 합(서로 다른 계수를 곱해 합하는 계산, weighted sum)을 함
  + 가중치는 해당 픽셀이 진하다는 것이 특정 클래스에 속한다는 것에 반하는 내용이라면 음(-)의 값을, 특정 클래스에 속한다는 것을 의미한다면 양(+)의 값을 가지게 됨.



* 모델이 각 클래스에 대해 학습한 가중치를 나타내는 그림
* 빨간 부분은 음의 가중치를, 파란 부분은 양의 가중치를 나타냄
* 여기서 바이어스(bias)라는 추가적인 항을 더하게 됨
* 결과값의 일부는 입력된 데이터와는 독릭적일 수 있다는 것을 고려하기 위함
* 이를 수식으로 표현하면, 입력값 x가 주어졌을 때 클래스 i에 대한 증거값은 https://lh4.googleusercontent.com/j-BNRn9E_hvB5NeHH605nSlNIGBaEyL2HvOXZjkb9EJEf7vkHvbbHa4dcZ8hkhjrKAH6INsmkXIRF2ckd5i1Qye3OhLnjRzcX4EUcpLREixX1_GTD5wwaXR3uwuBXMaISQKAsIP8
  + Wi : 가중치
  + bi : 클래스 i에 대한 바이어스
  + j : 입력 데이터로 사용한 이미지 x의 픽셀 값을 합하기 위한 인덱스
* **y = softmax(evidence)**
  + 각 클래스에 대해 계산한 증거값들을 “소프트맥스”함수를 활용해 예측 확률y로 변환
  + 소프트맥스는 우리가 계산한 선형 함수를 우리가 원하는 형태 - 이경우에서는 10가지 경우에 대한 확률 분포-로 변환하는데 사용하는 “활성화” 또는 “링크” 함수의 역할을 함.
  + 이번 예에서는 계산한 증거값들을 입력된 데이터 값이 각 클래스에 속할 확률로 변환하는 것이라고 생각해도 됨.
  + https://lh6.googleusercontent.com/6Nl2RJK67R958vQutm-80FGObF6j-Jwoa9N1lAIB4KWxIOTUp5YAmehFE6XxNHGLE_NZkT2GslwbjHbbPgJhwv-cInzLPL0HyOz8j6dtxDy02Qu3jcy_xL7qSApf9cGidZZJv6sQ과 같이 위 과정 정의
  + 전개한 식 → https://lh4.googleusercontent.com/vHuzBUiv7_q71vUlam5hFLEWwzOGHCk22ZNK3s-yOKjftu5wyOZJnvbg1KxgxpyyGNibe7ci1otLoNHUvfRzcU2q8dThrJNCPkE6ige7vCsB-RTYkzNXx_-fb-1t2DKY_2eTUEBD
    - 많은 경우, 일단 소프트맥스의 입력값을 지수화한 뒤 정규화 하는 과정이라고 생각
    - 지수화란
      * 증거값을 하나 더 추가하면 어떤 가설에 대해 주어진 가중치를 곱으로 증가시키는 것을 의미
      * 또한 반대로, 증거값의 개수가 하나 줄어든다는 것은 가설의 가중치가 기존 가중치의 분수비로 줄어들게 된다는 뜻
      * 어떤 가설도 0 또는 음의 가중치를 가질 수 없음
    - 가중치를 정규화한 후 모두 합하면 1이 되는 확률 분포로 만듬
    - ▲ 각각의 출력값에 대해, 가중치합을 계산하고 바이어스를 더한 뒤 소프트 맥스를 적용
    - 수식으로 표현 시
    - 이 과정을 행렬곱과 벡터합으로 변경하여 “벡터화”할 수 있음
      * 벡터화는 계산의 효율화에 도움이 됨
      * 다음과 같이 표현

**회귀 구현하기https://lh3.googleusercontent.com/orApASXNLWy99x_BwNJtYJ6_kZ-yp2pxTpg7uYh9xZwsWi5uAGcu5tBZuu97Qr1wmPjnVs2IBc_PGwNIYDMtIZZ6UUU6WH-ePimLQukRn1WVSVkg4hI4IJuP3LTvihHBfKpP6gqf**

* 파이썬에서 효율적인 수치 연산을 하기 위해, 다른 언어로 구현된 보다 효율이 높은 코드를 사용하여 행렬곱 같은 무거운 연산을 수행하는 NumPy 등의 라이브러리를 자주 사용
  + 매 연산마다 파있너으로 다시 돌아오는 과정에서 많은 오버헤드 발생할 수 있음
    - GPU에서 연산을 하거나 분산 처리 환경같은, 데이터 전송에 큰 비용이 발생할 수 있는 상황에서 특히 문제가 될 수도 있음
* 텐서플로우 역시 파이썬 외부에서 무거운 작업들을 수행, 텐서플로우는 이런 오버헤드를 피하기 위해 한 단계 더 나아간 방식 활용
  + 파이썬에서 하나의 무거운 작업을 족ㄼ적으로 실행하는 대신, 텐서플로우는 서로 상호작용하는 연ㅇ산간의 글프를 유저가 기술하도록 하고, 그 연산 모두가 파이썬 밖에서 동작
  + 텐서플로우 임포트
    - import tensorflow as tf
  + 심볼릭 변수를 활용해 상호작용하는 연산들 기술
    - x = tf.placeholder(tf.float32, [None, 784])
      * x에 특정한 값이 주어지는 것이 아님. 이는 ‘placeholder’로, 텐서플로우에서 연산을 실행할 때 값을 입력할 자리
      * 784차원의 벡터로 변형된 MNIST 이미지의 데이터를 넣으려 함
      * [None, 784]의 형태를 갖고 부동소수점에서 이루어진 2차원 텐서로 표현
      * (None은 해당 차원의 길이가 어떤 길이든지 될 수 있음을 의미)
  + 모델에는 가중치와 바이어스 역시 필요, Variable이라고 불리는 방법 사용
    - 서로 상호작용하는 연산으로 이루어진 텐서플로우 그래프 안에 존재하는, 수정 가능한 텐서
    - 연산에 사용되기도 하고, 연산을 통해 수정되기도 함
    - 머신러닝에 이를 사용할 때에는 주로 모델의 변수를 Variable들로 사용
      * W = tf.Variable(tf.zeros([784, 10]))  
        b = tf.Variable(tf.zeros([10]))
        + tf.Variable에 Variable의 초기값을 넘겨줌으로써 ,이 Variable들 생성
        + w와 b 둘 다 0으로 이루어진 텐서로 초기화
        + w가 [784, 10]의 형태를 갖는다

w에 784차원의 이미지 벡터를 곱해서 각 클래스에 대한 증거값을 나타내는 10차원 벡터를 얻고자 함

b은 그 10차원 벡터에 더하기 위해 [10]의 형태를 갖는 것

* + 모델 구현
    - y = tf.nn.softmax(tf.matmul(x, W) + b)
      * tf.matmul(x, W)로 x와 W를 곱
      * 위에서 본 수식에서 곱했던 순서인 https://lh5.googleusercontent.com/VYQk7ylKeCecWOryJWs0TqNSYzwXKaVFLwyeewB-I5zjd7ikQpkA3cwQX1KsPX4i3tXEFg9ggAFmJYvbG0qnEbY_iwUzZrV72x7Mwe---aPKw1_j5F9T93CtGzhw1J_s9tqQNkO6와 반대 (행렬이므로 순서가 중요)
      * x가 여러 입력값을 갖는 2차원 텐서 경우에도 대응하기 위한 작은 트릭
      * 그 다음엔 b를 더하고, 마지막으로 tf.nn.softmax을 적용
  + 한 번 작성한 모델을 여러 기기에서 실행할 수 있음

**학습**

* 모델을 학습시키기 위해서는 우선 모델이 좋다는 것은 어떤 것인지를 정의
* 머신러닝에서는 모델이 안 좋다는 것이 어떤 의미인지를 주로 정의
  + 이를 주로 비용(cost) 또는 손실(loss)이라고 부르며, 이것들은 모델이 원하는 결과에서 얼마나 떨어져 있는지를 보여주는 값
  + 그 격차를 줄이기 위해 노력하며, 그 격차가 적으면 적을수록 모델은 좋다고 말함
* 모델의 손실을 정의하기 위해 자주 사용되는 좋은 함수 중 하나로 “크로스 엔트로피” 있음
* 크로스 엔트로피는 정보 이론 분야에서 정보를 압축하는 방법으로써 고안된 것, 현재는 도박에서 머신러닝에 이르기까지 여러 분야에서 중요한 아이디어로 사용
* 크로스 엔트로피 정의 → https://lh3.googleusercontent.com/3NsPhxtMR_pHMFk9v1jSjMIWZq0ZxrRlMleIM37CWVwODcM7O-N6p8mh7FW87AxgO7EQvTAvn-x1Oi0EdIFrJ00itKpFkpLZmBCFNLsJnwoD3-psrTUhqIdkQtWSREfMedy_qjcD
  + y은 예측한 확률 분포, y’는 실제 분포( 입력하는 one-hot 벡터)
  + 크로스 엔트로피는 예측이 실제 값을 설명하기에 얼마나 비효율적인지를 측정하는 것
* 크로스 엔트로피를 구현하기 위해서는 올바른 답을 넣기 위한 새로운 placeholder를 추가하는 것부터 시작
  + y\_ = tf.placeholder(tf.float32, [None, 10])
* 크로스 엔트로피 https://lh3.googleusercontent.com/-nsLTz2UhsN8NRPSvXXyHqEHoz3MqwumdrkV0wItIhgR8Y5KpHjx7x95z7jKxGXt1gyEHbjl9ChIsYQAvu9WJ3taWmyzKpSmBbOeGOvG-w3U7_w97XB-ydrklR8icltDLLWHL2fJ를 구현
  + cross\_entropy = tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(y), reduction\_indices=[1]))
    - tf.log는 y의 각 원소의 로그 값을 계산, y\_의 각 원소를 tf.log(y)의 해당하는 원소들과 곱
    - tf.reduce\_sum으로 y의 2번째 차원(reduction\_indices=[1]이라는 파라미터가 주어졌으므로)의 원소들을 합함
    - tf.reduce\_mean으로 배치(batch)의 모든 예시에 대한 평균을 계산
    - (수학적으로 불안정한 계산이기 때문에, 소스 코드에서는 이 연산을 사용하지 않고 있는 것에 주의하길. 대신, 정규화 되지 않은 로짓(logit)에 대해 tf.nn.softmax\_cross\_entropy\_with\_logits을 적용  
      (즉, tf.matmul(x, W) + b)에 softmax\_cross\_entropy\_with\_logits을 사용합니다).   
      이렇게 하는 이유는 이 수학적으로 보다 안정적인 함수가 내부적으로 소프트맥스 활성을 계산하기 때문. 당신의 코드에서도 tf.nn.(sparse\_)softmax\_cross\_entropy\_with\_logits를 사용하는 것을 고려.)
* 모델이 할 일을 알고 있다면, 이를 텐서플로우를 통해 학습시키는 것은 매우 간단
* 텐서플로우는 하고자 하는 연산의 전체 그래프를 알고 있으므로, 손실에 설정한 변수들이 어떻게 영향을 주는지를 역전파(backpropagation) 알고리즘을 자동으로 사용하여 매우 효율적으로 정의
* 그러고 나서 텐서플로우는 선택한 최적화 알고리즘을 적용하여 변수를 수정하고 손실을 줄일 수 있음
  + train\_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross\_entropy)
    - 텐서플로우에게 학습 비율 0.5로 경사 하강법(gradient descent algorithm)을 적용하여 크로스 엔트로피를 최소화하도록 지시
    - 경사하강법이란, 텐서 플로우가 각각의 변수를 비용을 줄이는 방향으로 조금씩 이동시키는 매우 단순한 방법
  + 텐서플로우는 다른 여러 최적화 알고리즘을 제공
* 학습을 실행시키기 전에 마지막으로 우리가 작성한 변수들을 초기화하는 작업을 추가
  + init = tf.global\_variables\_initializer()
* 이제 Session에서 모델을 실행시키고, 변수들을 초기화 하는 작업을 실행시킬 수 있음
  + sess = tf.Session()  
    sess.run(init)
* 학습을 시킴 - 여기선 학습을 1000번 시킴
  + for i in range(1000):  
      batch\_xs, batch\_ys = mnist.train.next\_batch(100)  
      sess.run(train\_step, feed\_dict={x: batch\_xs, y\_: batch\_ys})
    - 반복되는 루프의 각 단계마다, 우리는 학습 데이터셋에서 무작위로 선택된 100개의 데이터로 구성된 “배치(batch)”를 가져옴
    - 그 다음엔 placeholder의 자리에 데이터를 넣을 수 있도록 train\_step을 실행하여 배치 데이터를 넘김
    - 무작위 데이터의 작은 배치를 사용하는 방법은 확률적 학습(stochastic training)이라고 부름   
      -여기서는 확률적 경사 하강법
    - 이상적으로는 학습의 매 단계마다 전체 데이터를 사용하고 싶지만, 그렇게 하면 작업이 무거워짐
    - 따라서 그 대신에 매번 서로 다른 부분집합을 사용하는 것
      * 작업 내용은 가벼워지지만 전체 데이터를 쓸 때의 이점은 거의 다 얻을 수 있음.

**모델 평가하기**

* 우선 모델이 라벨을 올바르게 예측했는지 확인
  + tf.argmax는 텐서 안에서 특정 축을 따라 가장 큰 값의 인덱스를 찾기에 매우 유용한 함수
  + EX) tf.argmax(y, 1)는 우리의 모델이 생각하기에 각 데이터에 가장 적합하다고 판단한 (가장 증거값이 큰)라벨, tf.argmax(y\_, 1)는 실제 라벨
  + tf.equal을 사용하여 예측이 맞았는지 확인할 수 있음
  + correct\_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y\_,1))
  + 이렇게 하면 부울 값으로 이루어진 리스트를 얻게 됨
  + 얼마나 많이 맞았는지 판단하려면, 이 값을 부동 소수점 값으로 변환한 후 평균을 계산
  + accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))
  + 테스트 데이터를 대상으로 정확도 계산
  + print(sess.run(accuracy, feed\_dict={x: mnist.test.images, y\_: mnist.test.labels}))
    - 결과는 약 92% 정도 나옴
    - 매우 안좋은 결과임. 왜냐하면 매우 단순한 모델을 사용했기 때문