

Deep Learning with Python

Chapter 1 DNN 01

■ 목차

1. DNN을 이용한 손글씨 인식 (MNIST)

- a. MNIST 손글씨 인식하기
- b. 일반적인 Softmax를 이용한 구현

2. DNN 성능 향상

- a. Deep Neural Network와 ReLU를 이용한 성능 향상
- b. Learning Rate 조정을 통한 성능 향상
- c. Weight Initialization를 이용한 성능 향상
- d. Deep and Wide Neural Network를 이용한 성능 향상
- e. Dropout을 이용한 성능 향상

2. Keras를 이용한 DNN 구현

- a. Keras 소개 및 특징
- b. Keras를 이용한 MNIST 손글씨 인식하기

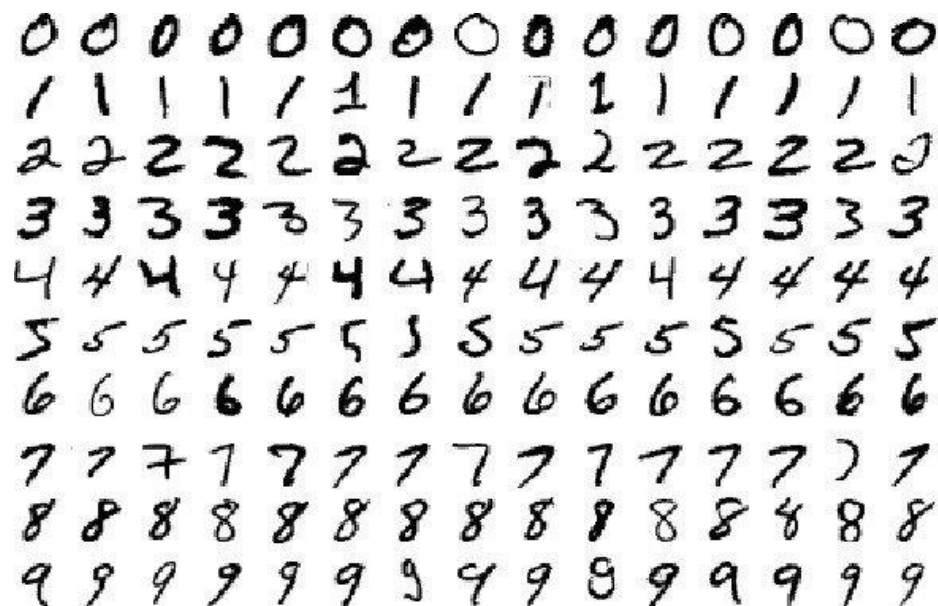
■ 학습목표

1. DNN을 이용한 손글씨 인식 알고리즘을 이해하고 tensorflow를 이용하여 직접 구현한다.
2. DNN 성능을 높이기 위한 방법을 이해하고 tensorflow를 이용하여 직접 구현한다.
3. Keras의 특징을 이해하고 DNN을 이용한 손글씨 인식 알고리즘을 Keras를 이용하여 구현한다.

1. DNN을 이용한 손글씨 인식 (MNIST)

1. DNN을 이용한 손글씨 인식(MNIST)

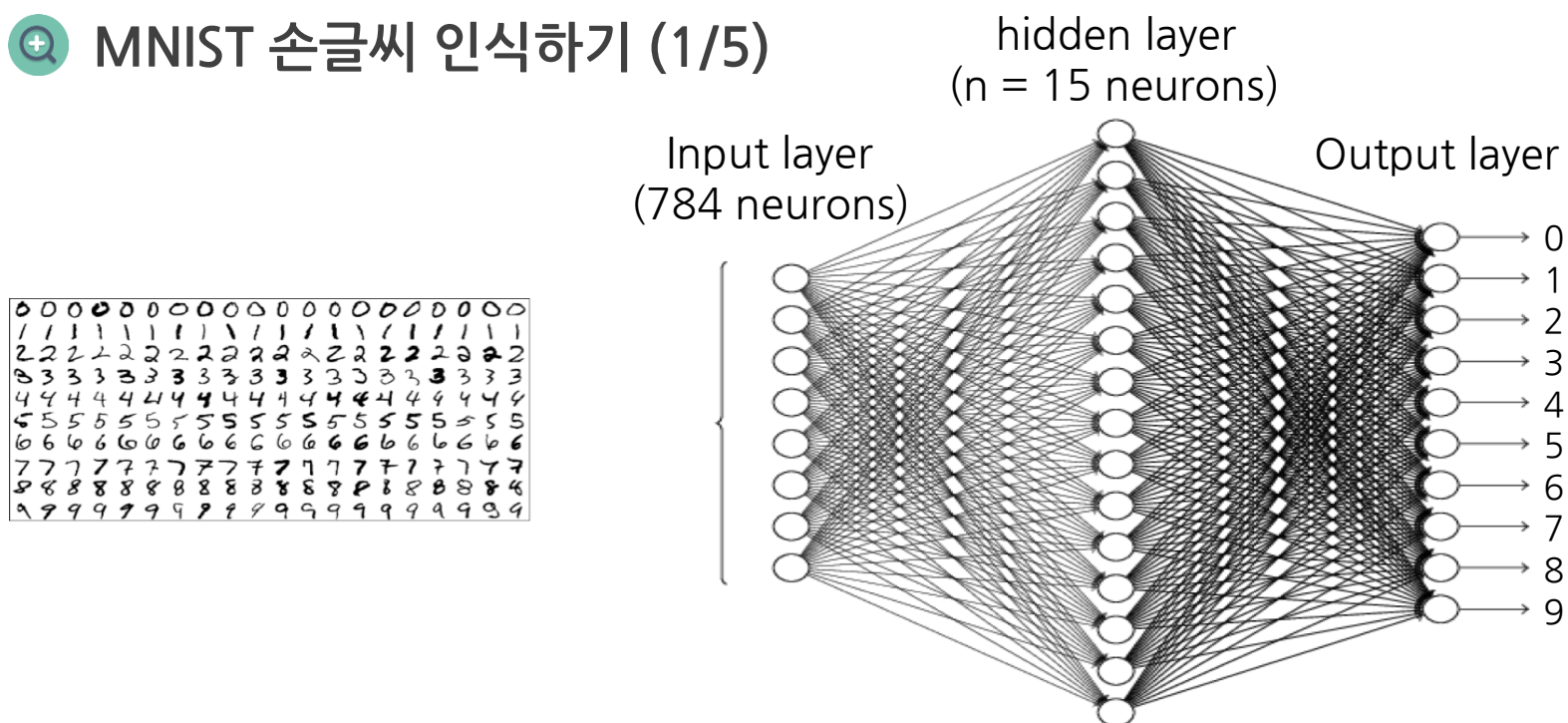
MNIST 데이터 셋



- 머신러닝 대가 Yann Lecun 뉴욕대 교수가 제공하는 데이터 셋
- 0~9사이의 숫자 이미지와 이에 해당하는 레이블(Label)로 구성된 데이터 셋이다.

1. DNN을 이용한 손글씨 인식(MNIST)

MNIST 손글씨 인식하기 (1/5)



- 손글씨 이미지는 가로 28, 세로 28픽셀로 학습해야하는 픽셀은 가로와 세로를 곱한 784픽셀이다.
- 784개의 입력 데이터에 대해서 신경망을 구성하여 0~9까지 10개의 출력이 나오도록 해야한다.

1. DNN을 이용한 손글씨 인식(MNIST)

🔍 MNIST 손글씨 인식하기 (2/5)



$x = 28 \times 28$ Image



Model



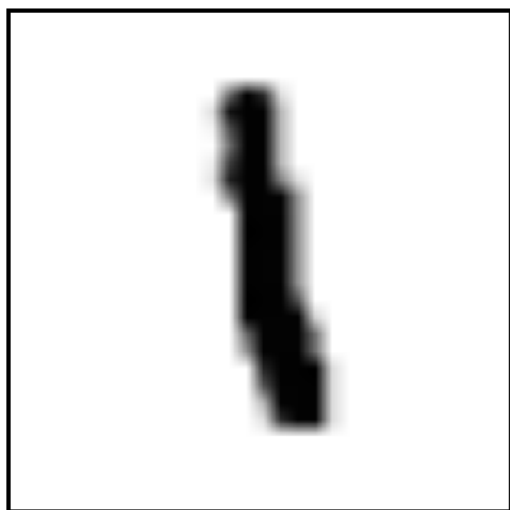
0

$y = \text{label}$

- MNIST 이미지를 입력(X)으로 받아서 어떤 숫자인지(Y)를 출력하는 손글씨 인식 모델을 실습할 것이다.
- 출력할 결과값은 0~9까지의 총 10개의 레이블(label)이다.

1. DNN을 이용한 손글씨 인식(MNIST)

🔍 MNIST 손글씨 인식하기 (3/5)



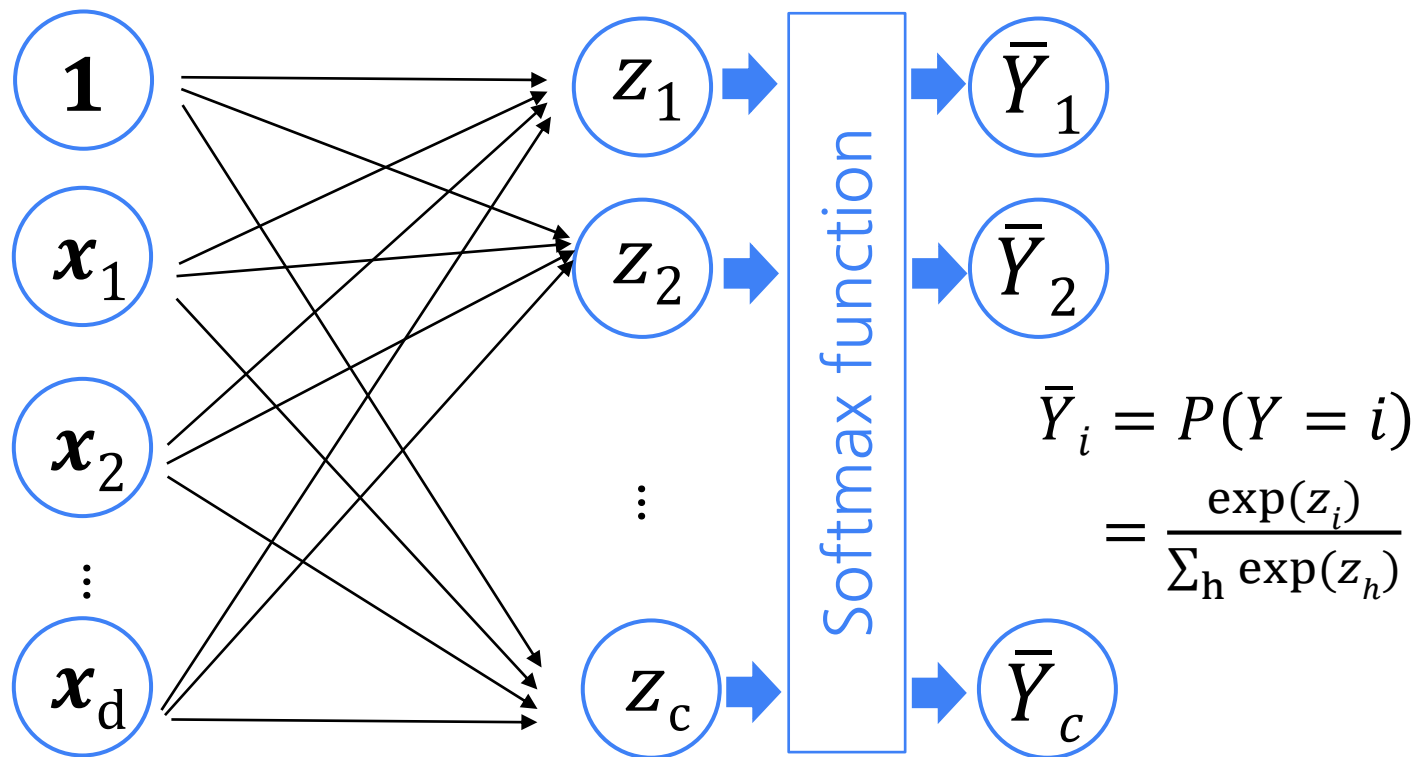
≈

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	.6	.8	0	0	0	0	0	0	0
0	0	0	0	0	0	.7	1	0	0	0	0	0	0	0
0	0	0	0	0	0	.7	1	0	0	0	0	0	0	0
0	0	0	0	0	0	.5	1	.4	0	0	0	0	0	0
0	0	0	0	0	0	0	1	.4	0	0	0	0	0	0
0	0	0	0	0	0	0	1	.4	0	0	0	0	0	0
0	0	0	0	0	0	0	1	.7	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	.9	1	.1	0	0	0	0	0
0	0	0	0	0	0	0	.3	1	.1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- 사람은 1이 그려진 이미지를 보면 이것이 숫자 1이라는 추상적인 의미를 쉽게 해석해낼 수 있다.
- 하지만 컴퓨터는 1이 그려진 이미지는 단지 픽셀 밝기 값으로 구성된 2차원 행렬이기 때문에 의미를 해석하는데 쉽지 않다.

1. DNN을 이용한 손글씨 인식(MNIST)

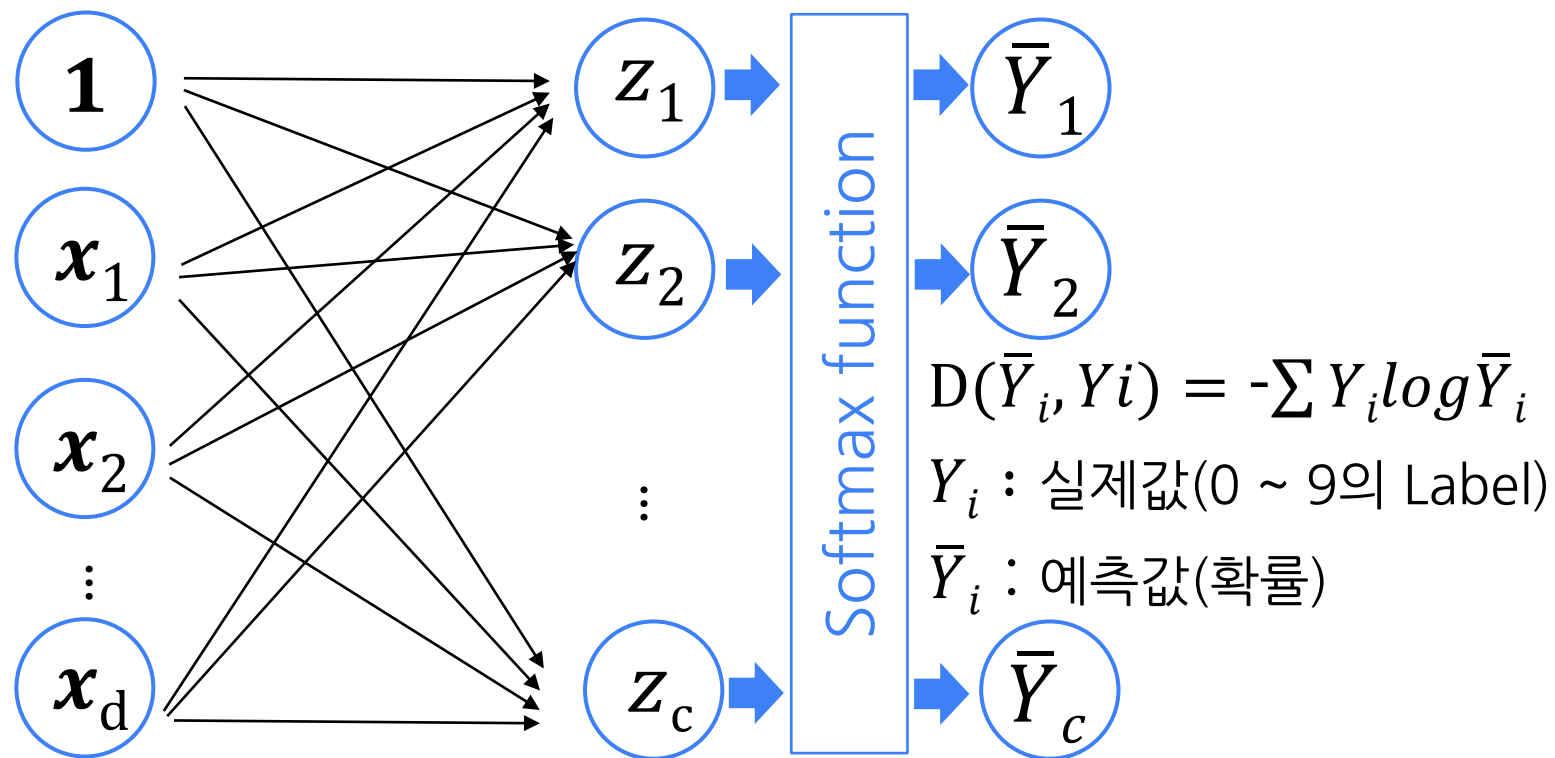
MNIST 손글씨 인식하기 (4/5)



- 이를 해결하기 위해 Softmax Regression 기법을 사용하여 0~9사이의 레이블(Label)을 예측한다.

1. DNN을 이용한 손글씨 인식(MNIST)

🔍 MNIST 손글씨 인식하기 (5/5)



- Softmax Regression 모델을 평가하는 비용함수로는 cross-entropy 함수를 사용한다.

1. DNN을 이용한 손글씨 인식(MNIST)

일반적인 Softmax를 이용한 구현: 전체 코드 (1/4)

```
import tensorflow as tf
import random

from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

tf.reset_default_graph()
tf.set_random_seed(1234)

learning_rate = 0.1
training_cnt = 15
batch_size = 100

X = tf.placeholder(tf.float32, [None, 784])
Y = tf.placeholder(tf.float32, [None, 10])
```

1. DNN을 이용한 손글씨 인식(MNIST)

일반적인 Softmax를 이용한 구현: 전체 코드 (2/4)

```
W = tf.Variable(tf.random_normal([784, 10]))
b = tf.Variable(tf.random_normal([10]))

logits = tf.matmul(X, W) + b

cost =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits, labels=Y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
op_train = optimizer.minimize(cost)

pred = tf.nn.softmax(logits)
prediction = tf.argmax(pred, 1)
true_Y = tf.argmax(Y, 1)
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction, true_Y),
dtype=tf.float32))
```

1. DNN을 이용한 손글씨 인식(MNIST)

일반적인 Softmax를 이용한 구현: 전체 코드 (3/4)

```
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

for epoch in range(training_cnt):
    avg_cost = 0
    total_batch = int(mnist.train.num_examples / batch_size)

    for i in range(total_batch):
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        feed_dict = {X: batch_xs, Y: batch_ys}
        c, _ = sess.run([cost, op_train], feed_dict=feed_dict)
        avg_cost += c / total_batch

    print('Epoch:', '%04d' % (epoch + 1), 'cost =',
          '{:.9f}'.format(avg_cost))

print('Learning Finished!')
```

1. DNN을 이용한 손글씨 인식(MNIST)

일반적인 Softmax를 이용한 구현: 전체 코드 (4/4)

```
print('Accuracy(train):', sess.run(accuracy, feed_dict={
    X: mnist.train.images, Y: mnist.train.labels}))

print('Accuracy(test):', sess.run(accuracy, feed_dict={
    X: mnist.test.images, Y: mnist.test.labels}))

r = random.randint(0, mnist.test.num_examples - 1)
print("Label: ", sess.run(tf.argmax(mnist.test.labels[r:r + 1],
1)))
print("Prediction: ", sess.run(
    prediction, feed_dict={X: mnist.test.images[r:r + 1]}))
```

1. DNN을 이용한 손글씨 인식(MNIST)

모델 구축(Build graph) (1/5)

```
from tensorflow.examples.tutorials.mnist import input_data
```

- tensorflow.examples.tutorials.mnist 라이브러리에서 input_data를 import한다.

```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

- input_data에서 “MNIST_data” 읽어온다.
- one_hot을 True로 설정하여 결과 데이터를 one-hot encoding 형태로 가져온다.

```
tf.reset_default_graph()  
tf.set_random_seed(1234)
```

- 재현성을 위해 tf.set_random_seed함수를 사용한다.
Tensorflow 그래프 마다 설정되므로 재실행 시 그래프 리셋이 필요하다.

1. DNN을 이용한 손글씨 인식(MNIST)

모델 구축(Build graph) (2/5)

학습을 위한 기초 파라미터 값 설정

- learning_rate: weight가 발산되지 않도록 조정하는 값으로 weight 값이 너무 작으면 train 되지 않을 수 있고, 너무 크면 overshooting이 발생할 수 있다.
- training_cnt: 전체 데이터 셋에 대한 학습 반복 횟수(Epoch)
- batch_size: 한번에 학습할 데이터의 수

```
# 파라미터값 설정  
learning_rate = 0.1  
training_cnt = 15  
batch_size = 100
```


1. DNN을 이용한 손글씨 인식(MNIST)

모델 구축(Build graph) (3/5)

tf graph input

- X : 들어오는 row는 정해진 게 없고, column 은 784개, 즉 입력 변수 784개
- Y : 들어오는 row는 정해진 게 없고, column 은 10개 ,즉 output 10개
- matrix를 사용 하기 때문에 입력 변수를 담는 placeholder는 1개로 된다.

```
X = tf.placeholder(tf.float32, [None, 784])  
Y = tf.placeholder(tf.float32, [None, 10])
```

tf.random_normal

- 784개의 픽셀마다 가중치들을 각각 학습하여 0부터 9까지 숫자를 인식
- weight, bias의 초기값을 난수로 생성

```
W = tf.Variable(tf.random_normal([784, 10]))  
b = tf.Variable(tf.random_normal([10]))
```

1. DNN을 이용한 손글씨 인식(MNIST)

🔍 모델 구축(Build graph) (4/5)

✓ matmul함수 사용

- 입력 X와 가중치 W를 곱하고 편향 b를 더하여 모델을 정의한다.

```
logits = tf.matmul(X, W) + b
```

✓ cost/loss function 구현

- 교차 엔트로피(cross-entropy) 사용
- 예측값과 실제값 사이의 확률분포 차이 계산
- 학습 방법으로 GradientDescent 함수 사용 (경사하강법)

```
cost =  
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits  
=logits, labels=Y))  
optimizer = tf.train.GradientDescentOptimizer(learning_rate)  
op_train = optimizer.minimize(cost)
```

1. DNN을 이용한 손글씨 인식(MNIST)

모델 구축(Build graph) (5/5)

학습된 예측값을 확인, 정확도 계산

- softmax 함수를 적용하여 출력값의 합이 항상 1이 되게 한다.
- 예측된 최대 값의 index 반환
- One-hot encoding 한 Y값도 최대값 1이 있는 index 반환
- 평균을 이용하여 예측값과 실제 데이터의 일치 여부를 계산

```
pred = tf.nn.softmax(logits)
prediction = tf.argmax(pred, 1)
true_Y = tf.argmax(Y, 1)
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction,
true_Y), dtype=tf.float32))
```

1. DNN을 이용한 손글씨 인식(MNIST)

모델 실행(run/update) (1/3)

- 모델을 실행하기 위해 세션을 열고 변수를 초기화 한다.

```
sess = tf.Session()  
init = tf.global_variables_initializer()  
sess.run(init)
```

1. DNN을 이용한 손글씨 인식(MNIST)

모델 실행(run/update) (2/3)

- Tensorflow에서 제공하는 훈련 데이터(train data)가 55,000개이기 때문에 여러 개의 batch로 나누어 학습을 진행하는 것이 효율적이다.
- total_batch는 $55,000/100 = 550$ 이다.
- training_cnt 만큼 반복하는 for 문 안에 total_batch 만큼 반복하는 for 문이 포함되어있다.
- avg_cost는 전체 cost를 total_batch만큼 나눈 값을 더하여 계산된다.

```
for epoch in range(training_cnt):  
    avg_cost = 0  
    total_batch = int(mnist.train.num_examples / batch_size)  
  
    for i in range(total_batch):  
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)  
        feed_dict = {X: batch_xs, Y: batch_ys}  
        c, _ = sess.run([cost, op_train], feed_dict=feed_dict)  
        avg_cost += c / total_batch
```

1. DNN을 이용한 손글씨 인식(MNIST)

모델 실행(run/update) (3/3)

- 전체 데이터 셋을 반복한 단계와 각 단계에 해당하는 cost인 avg_cost를 출력
- Accuracy는 훈련 데이터(Train data)로 학습한 모델을 시험 데이터(Test data)를 대상으로 적용한 정확도를 나타낸다.

```
print('Epoch:', '%04d' % (epoch + 1), 'cost =',  
      '{:.9f}'.format(avg_cost))  
  
print('Learning Finished!')
```

1. DNN을 이용한 손글씨 인식(MNIST)

모델 검정(test)

- Accuracy는 학습한 모델로 훈련 데이터(Train data)와 시험 데이터(Test data)를 대상으로 적용한 정확도를 나타낸다.
- r은 시험 데이터(Test data)에서 랜덤 하게 1개를 읽어 온 것이다.
- Label은 r에 해당하는 0~9사이의 실제 레이블(Label)
- Prediction은 r에 해당하는 이미지의 예측값(0~9사이의 레이블)

```
print('Accuracy(train):', sess.run(accuracy, feed_dict={
    X: mnist.train.images, Y: mnist.train.labels}))

print('Accuracy(test):', sess.run(accuracy, feed_dict={
    X: mnist.test.images, Y: mnist.test.labels}))

r = random.randint(0, mnist.test.num_examples - 1)
print("Label: ", sess.run(tf.argmax(mnist.test.labels[r:r + 1], 1)))
print("Prediction: ", sess.run(
    prediction, feed_dict={X: mnist.test.images[r:r + 1]}))
```

1. DNN을 이용한 손글씨 인식(MNIST)

학습 결과

```
Epoch: 0001 cost = 2.979994300
Epoch: 0002 cost = 1.101453354
Epoch: 0003 cost = 0.877323593
Epoch: 0004 cost = 0.769285322
Epoch: 0005 cost = 0.701442515
...
Epoch: 0012 cost = 0.510681413
Epoch: 0013 cost = 0.497372317
Epoch: 0014 cost = 0.485437911
Epoch: 0015 cost = 0.474215607
Learning Finished!
Accuracy(train): 0.8876182
Accuracy(test): 0.8897
Label: [5]
Prediction: [5]
```


2. DNN 성능 향상

2. DNN 성능 향상

DNN의 성능을 향상시키는 여러가지 방법

1. Deep Neural Network와 ReLU를 추가하여 모델 변경
2. Learning Rate 조정
3. 향상된 Optimizer 사용
4. 적절한 Weight Initializer 사용
5. Deep & Wide Neural Network 확장
6. Dropout 적용

2. DNN 성능 향상

🔍 Deep Neural Network와 ReLU를 추가하여 모델 변경

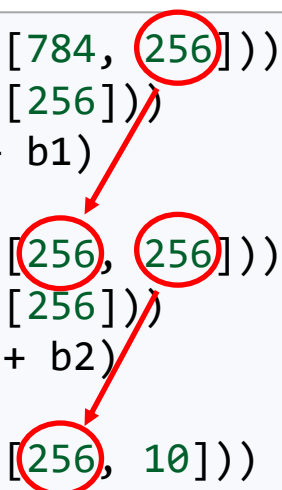
✓ 심층신경망(Deep Neural Network)구성과 ReLU함수 사용

- 이전 softmax를 이용한 구현과 다른 점은 여러 개의 layer를 추가하여 심층신경망을 구성한 것과 활성화 함수로 ReLU함수를 사용한 것이다.
- 각 layer의 결과값이 다음 layer의 입력값으로 연결되는 것을 주목한다.
- 동일한 결과값을 위해 seed 옵션을 설정한다.

```
W1 = tf.Variable(tf.random_normal([784, 256]))
b1 = tf.Variable(tf.random_normal([256]))
L1 = tf.nn.relu(tf.matmul(X, W1) + b1)

W2 = tf.Variable(tf.random_normal([256, 256]))
b2 = tf.Variable(tf.random_normal([256]))
L2 = tf.nn.relu(tf.matmul(L1, W2) + b2)

W3 = tf.Variable(tf.random_normal([256, 10]))
b3 = tf.Variable(tf.random_normal([10]))
logits = tf.matmul(L2, W3) + b3
```



2. DNN 성능 향상

🔍 Deep Neural Network와 ReLU를 추가한 학습 결과

Epoch: 0001 cost = 2.979994300
Epoch: 0002 cost = 1.101453354
Epoch: 0003 cost = 0.877323593
Epoch: 0004 cost = 0.769285322
Epoch: 0005 cost = 0.701442515

...

Epoch: 0012 cost = 0.510681413
Epoch: 0013 cost = 0.497372317
Epoch: 0014 cost = 0.485437911
Epoch: 0015 cost = 0.474215607

Learning Finished!

Accuracy(train): 0.8876182

Accuracy(test): 0.8897

Label: [5]

Prediction: [5]

Epoch: 0001 cost = 137.375869115

Epoch: 0002 cost = 3.042602665

Epoch: 0003 cost = 2.698789790

Epoch: 0004 cost = 2.398083852

Epoch: 0005 cost = 2.361849550

...

Epoch: 0012 cost = 2.142320057

Epoch: 0013 cost = 2.127378496

Epoch: 0014 cost = 2.085216938

Epoch: 0015 cost = 2.046381193

Learning Finished!

Accuracy(train): 0.22627273

Accuracy(test): 0.2279

Label: [2]

Prediction: [2]

!!!

2. DNN 성능 향상

Learning Rate 조정

- Cost가 기존 대비 큰 값(2.xx)을 가지며 더 이상 0에 가깝게 수렴하지 않는다.
- Overshooting의 가능성이 있으므로 learning_rate 값을 줄여본다.
- learning_rate 값은 10의 누승으로 다양하게 변경해본다.

파라미터값 설정

```
learning_rate = 0.01  
training_cnt = 15  
batch_size = 100
```

2. DNN 성능 향상

Learning Rate 조정(0.01) 후 학습 결과

Epoch: 0001 cost = 137.375869115
Epoch: 0002 cost = 3.042602665
Epoch: 0003 cost = 2.698789790
Epoch: 0004 cost = 2.398083852
Epoch: 0005 cost = 2.361849550

...

Epoch: 0012 cost = 2.142320057
Epoch: 0013 cost = 2.127378496
Epoch: 0014 cost = 2.085216938
Epoch: 0015 cost = 2.046381193
Learning Finished!

Accuracy(train): 0.22627273

Accuracy(test): 0.2279

Label: [2]

Prediction: [2]

Epoch: 0001 cost = 62.404162514
Epoch: 0002 cost = 11.952705775
Epoch: 0003 cost = 7.357152601
Epoch: 0004 cost = 5.255919425
Epoch: 0005 cost = 3.984169432

...

Epoch: 0012 cost = 1.125415904
Epoch: 0013 cost = 0.980516475
Epoch: 0014 cost = 0.867828476
Epoch: 0015 cost = 0.754599151
Learning Finished!

Accuracy(train): 0.9754

Accuracy(test): 0.9247

Label: [0]

Prediction: [0]

2. DNN 성능 향상

향상된 Optimizer 사용

Adam Optimizer 사용

- 딥러닝 Optimizer 중 가장 성능이 좋은 것으로 평가되는 Adam Optimizer를 사용한다.

```
optimizer = tf.train.AdamOptimizer(learning_rate)
op_train = optimizer.minimize(cost)
```

2. DNN 성능 향상

Adam Optimizer 적용 후 학습 결과

Epoch: 0001 cost = 62.404162514
Epoch: 0002 cost = 11.952705775
Epoch: 0003 cost = 7.357152601
Epoch: 0004 cost = 5.255919425
Epoch: 0005 cost = 3.984169432

...

Epoch: 0012 cost = 1.125415904
Epoch: 0013 cost = 0.980516475
Epoch: 0014 cost = 0.867828476
Epoch: 0015 cost = 0.754599151
Learning Finished!

Accuracy(train): 0.9754

Accuracy(test): 0.9247

Label: [0]

Prediction: [0]

Epoch: 0001 cost = 49.810910983
Epoch: 0002 cost = 9.035207869
Epoch: 0003 cost = 4.926626853
Epoch: 0004 cost = 3.380977759
Epoch: 0005 cost = 3.085615649

...

Epoch: 0012 cost = 1.041696607
Epoch: 0013 cost = 1.235588291
Epoch: 0014 cost = 0.969958042
Epoch: 0015 cost = 0.949117825
Learning Finished!

Accuracy(train): 0.9877273

Accuracy(test): 0.9666

Label: [1]

Prediction: [1]

2. DNN 성능 향상

적절한 Weight Initializer 사용

Xavier Initializer 사용

- 정확도를 높이기 위해 Xavier Initializer를 사용하여 가중치를 초기화 한다.
- 기존에는 tf.Variable 함수를 이용했지만 Xavier를 사용하기 위해서는 tf.get_variable 함수를 이용하여 가중치 변수를 만들고 초기화 한다.

```
W1 = tf.get_variable("W1", shape=[784, 256],  
                    initializer=tf.contrib.layers.xavier_initializer())
```

```
b1 = tf.Variable(tf.random_normal([256]))
```

```
L1 = tf.nn.relu(tf.matmul(X, W1) + b1)
```

⋮

```
W3 = tf.get_variable("W3", shape=[256, 10],  
                    initializer=tf.contrib.layers.xavier_initializer())
```

```
b3 = tf.Variable(tf.random_normal([10]))
```

```
logits = tf.matmul(L2, W3) + b3
```

```
pred = tf.nn.softmax(logits)
```

2. DNN 성능 향상

Xavier Initializer 적용 후 학습 결과

Epoch: 0001 cost = 49.810910983

Epoch: 0002 cost = 9.035207869

Epoch: 0003 cost = 4.926626853

Epoch: 0004 cost = 3.380977759

Epoch: 0005 cost = 3.085615649

...

Epoch: 0012 cost = 1.041696607

Epoch: 0013 cost = 1.235588291

Epoch: 0014 cost = 0.969958042

Epoch: 0015 cost = 0.949117825

Learning Finished!

Accuracy(train): 0.9877273

Accuracy(test): 0.9666

Label: [1]

Prediction: [1]

Epoch: 0001 cost = 0.310716602

Epoch: 0002 cost = 0.115760933

Epoch: 0003 cost = 0.076327007

Epoch: 0004 cost = 0.054352766

Epoch: 0005 cost = 0.041559012

...

Epoch: 0012 cost = 0.010041498

Epoch: 0013 cost = 0.010976161

Epoch: 0014 cost = 0.009182683

Epoch: 0015 cost = 0.011391509

Learning Finished!

Accuracy(train): 0.9825818

Accuracy(test): 0.9658

Label: [9]

Prediction: [9]

!!!

2. DNN 성능 향상

Learning Rate 재조정

- Weight 초기화 변경 이후 기존 대비 cost가 더 작은 값을 가지지만 학습 결과는 오히려 나빠졌다.
- learning rate를 변경해서 더 세밀하게 최적 값을 찾도록 시도해본다.
- learning_rate 값은 10의 누승으로 다양하게 변경해본다.

파라미터값 설정

```
learning_rate = 0.001  
training_cnt = 15  
batch_size = 100
```

2. DNN 성능 향상

Learning Rate 재조정 후 학습 결과

Epoch: 0001 cost = 0.275951076

Epoch: 0002 cost = 0.137877331

Epoch: 0003 cost = 0.118785933

Epoch: 0004 cost = 0.106765097

Epoch: 0005 cost = 0.098277551

...

Epoch: 0012 cost = 0.068597365

Epoch: 0013 cost = 0.066519971

Epoch: 0014 cost = 0.071701345

Epoch: 0015 cost = 0.062908032

Learning Finished!

Accuracy(train): 0.9825818

Accuracy(test): 0.9658

Label: [2]

Prediction: [2]

Epoch: 0001 cost = 0.310716602

Epoch: 0002 cost = 0.115760933

Epoch: 0003 cost = 0.076327007

Epoch: 0004 cost = 0.054352766

Epoch: 0005 cost = 0.041559012

...

Epoch: 0012 cost = 0.010041498

Epoch: 0013 cost = 0.010976161

Epoch: 0014 cost = 0.009182683

Epoch: 0015 cost = 0.011391509

Learning Finished!

Accuracy(train): 0.9979454

Accuracy(test): 0.9789

Label: [1]

Prediction: [1]

2. DNN 성능 향상

적절한 Weight Initializer 사용

He Initializer 사용

- Xavier Initializer 대신 He Initializer를 사용하여 가중치를 초기화 한다.
- 활성화 함수로 ReLU를 사용하는 경우 He Initializer를 사용하는게 효과적이다.
- 위 예제처럼 모형이 복잡하지 않은 경우에 사용해야 효과가 좋다.

```
W1 = tf.get_variable("W1", shape=[784, 256],
                      initializer=tf.keras.initializers.he_normal())

b1 = tf.Variable(tf.random_normal([256]))
L1 = tf.nn.relu(tf.matmul(X, W1) + b1)

      ⋮

W3 = tf.get_variable("W3", shape=[256, 10],
                      initializer=tf.keras.initializers.he_normal())

b3 = tf.Variable(tf.random_normal([10]))
logits = tf.matmul(L2, W3) + b3
pred = tf.nn.softmax(logits)
```

2. DNN 성능 향상

He Initializer 적용 후 학습 결과

```
Epoch: 0001 cost = 0.310716602
Epoch: 0002 cost = 0.115760933
Epoch: 0003 cost = 0.076327007
Epoch: 0004 cost = 0.054352766
Epoch: 0005 cost = 0.041559012
```

...

```
Epoch: 0012 cost = 0.010041498
Epoch: 0013 cost = 0.010976161
Epoch: 0014 cost = 0.009182683
Epoch: 0015 cost = 0.011391509
```

Learning Finished!

Accuracy(train): 0.9979454

Accuracy(test): 0.9789

Label: [1]

Prediction: [1]

```
Epoch: 0001 cost = 0.303994064
Epoch: 0002 cost = 0.110615502
Epoch: 0003 cost = 0.073939196
Epoch: 0004 cost = 0.051303278
Epoch: 0005 cost = 0.039809300
```

...

```
Epoch: 0012 cost = 0.010703627
Epoch: 0013 cost = 0.010910786
Epoch: 0014 cost = 0.009297840
Epoch: 0015 cost = 0.011863690
```

Learning Finished!

Accuracy(train): 0.99816364

Accuracy(test): 0.9795

Label: [3]

Prediction: [3]

2. DNN 성능 향상

Deep & Wide Neural Network 확장

Neural Network 크기 증가

- 은닉층 노드 수를 증가시키고 레이어를 추가하였다.

```
W1 = tf.get_variable("W1", shape=[784, 512],  
                      initializer=tf.keras.initializers.he_normal())  
b1 = tf.Variable(tf.random_normal([512]))  
L1 = tf.nn.relu(tf.matmul(X, W1) + b1)  
⋮  
  
W5 = tf.get_variable("W5", shape=[512, 10],  
                      initializer=tf.keras.initializers.he_normal())  
b5 = tf.Variable(tf.random_normal([10]))  
logits = tf.matmul(L4, W5) + b5
```

2. DNN 성능 향상

Deep & Wide Neural Network 확장 후 학습 결과

Epoch: 0001 cost = 0.303994064
Epoch: 0002 cost = 0.110615502
Epoch: 0003 cost = 0.073939196
Epoch: 0004 cost = 0.051303278
Epoch: 0005 cost = 0.039809300

...

Epoch: 0012 cost = 0.010703627
Epoch: 0013 cost = 0.010910786
Epoch: 0014 cost = 0.009297840
Epoch: 0015 cost = 0.011863690

Learning Finished!

Accuracy(train): 0.99816364

Accuracy(test): 0.9795

Label: [3]

Prediction: [3]

Epoch: 0001 cost = 0.282887203
Epoch: 0002 cost = 0.103374577
Epoch: 0003 cost = 0.070238777
Epoch: 0004 cost = 0.050822332
Epoch: 0005 cost = 0.041946811

...

Epoch: 0012 cost = 0.013867383
Epoch: 0013 cost = 0.018362136
Epoch: 0014 cost = 0.014810844
Epoch: 0015 cost = 0.013397067

Learning Finished!

Accuracy(train): 0.99854547

Accuracy(test): 0.982

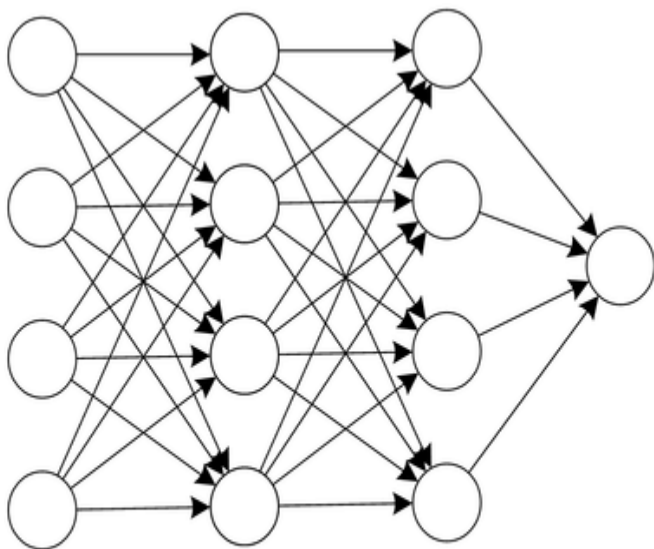
Label: [5]

Prediction: [5]

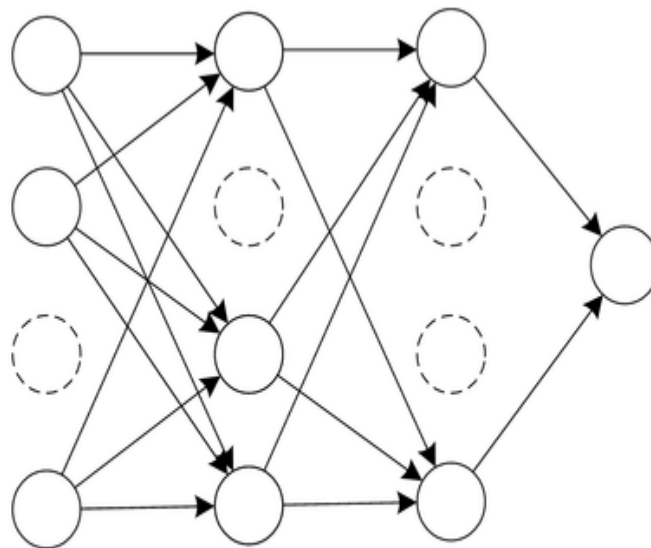
2. DNN 성능 향상

+ Dropout 기법

- Dropout은 Overfitting이 일어나지 않도록 중간 중간 무작위로 뉴런을 비활성화하여 성능을 향상 시키는 방법이다.
- 학습 시간은 다소 길어지지만 모델의 일반적인 예측 성능을 높여준다.



(a) Standard Neural Network



(b) Network after Dropout

2. DNN 성능 향상

Dropout 적용 (1/3)

`tf.placeholder` 함수 사용

- `keep_prob` : Dropout에서 유지할 노드를 위해 설정하는 것
- 보통 학습할 때(0.5~0.7)와 시험할 때(1)가 다르기 때문에 `placeholder` 함수를 사용하여 상황에 따라 `feed_dict`로 값을 준다.

```
keep_prob = tf.placeholder(tf.float32)
```

2. DNN 성능 향상

Dropout 적용 (2/3)

tf.nn.dropout 함수 사용

- tf.nn.dropout 함수를 이용한 layer를 하나 더 추가하여 overfitting을 방지하였다.

```
W1 = tf.get_variable("W1", shape=[784, 512],  
                      initializer=tf.keras.initializers.he_normal())  
b1 = tf.Variable(tf.random_normal([512]))  
L1 = tf.nn.relu(tf.matmul(X, W1) + b1)  
L1 = tf.nn.dropout(L1, keep_prob=keep_prob)  
  
⋮  
  
W5 = tf.get_variable("W5", shape=[512, 10],  
                      initializer=tf.keras.initializers.he_normal())  
b5 = tf.Variable(tf.random_normal([10]))  
logits = tf.matmul(L4, W5) + b5
```

2. DNN 성능 향상

Dropout 적용 (3/3)

- keep_prob는 학습 시 0.7, 테스트 시 1로 다르게 설정한다.
- Overfitting 정도에 따라 학습 시 수치를 적절히 조절한다.

```
for epoch in range(training_cnt):
    avg_cost = 0

    for i in range(total_batch):
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        feed_dict = {X: batch_xs, Y: batch_ys, keep_prob: 0.7}
        c, _ = sess.run([cost, op_train], feed_dict=feed_dict)
        avg_cost += c / total_batch

    print('Epoch:', '%04d' % (epoch + 1), 'cost =',
          '{:.9f}'.format(avg_cost))
    :

print('Accuracy(test):', sess.run(accuracy, feed_dict={
    X: mnist.test.images, Y: mnist.test.labels, keep_prob: 1}))
```

2. DNN 성능 향상

Dropout 적용 후 학습 결과

Epoch: 0001 cost = 0.282887203
Epoch: 0002 cost = 0.103374577
Epoch: 0003 cost = 0.070238777
Epoch: 0004 cost = 0.050822332
Epoch: 0005 cost = 0.041946811

...

Epoch: 0012 cost = 0.013867383
Epoch: 0013 cost = 0.018362136
Epoch: 0014 cost = 0.014810844
Epoch: 0015 cost = 0.013397067

Learning Finished!

Accuracy(train): 0.99854547

Accuracy(test): 0.982

Label: [5]

Prediction: [5]

Epoch: 0001 cost = 0.486314783
Epoch: 0002 cost = 0.177827160
Epoch: 0003 cost = 0.132101215
Epoch: 0004 cost = 0.108673234
Epoch: 0005 cost = 0.096628661

...

Epoch: 0012 cost = 0.054296678
Epoch: 0013 cost = 0.050866774
Epoch: 0014 cost = 0.049649445
Epoch: 0015 cost = 0.047520020

Learning Finished!

Accuracy(train): 0.9958182

Accuracy(test): 0.9845

Label: [8]

Prediction: [8]

2. DNN 성능 향상

DNN 성능 향상 과정

순서	내용	인식률 (Test Set)
0	Softmax 기본 구현	0.8894
1	Deep Neural Network + ReLU	0.2279
2	Learning Rate = 0.01	0.9247
3	Adam Optimizer 적용	0.9666
4	Xavier Initializer 적용	0.9658
5	Learning Rate = 0.001	0.9789
6	He Initializer 적용	0.9795
7	Deep & Wide Neural Network 확장	0.9820
8	Dropout 적용	0.9845

2. DNN 성능 향상

추가로 적용해 볼만한 요소

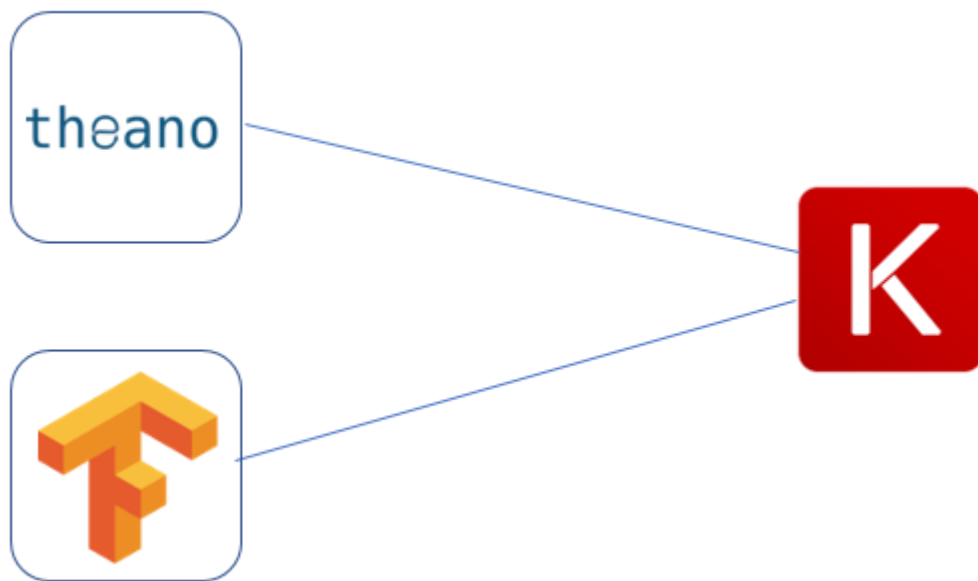
- 반복 학습 횟수(최대 epoch) 조절
- Batch size 및 Iteration 횟수 조절

3. Keras를 이용한 DNN 구현

3. Keras를 이용한 DNN 구현(MNIST)

Keras 소개

- 파이썬으로 작성된 오픈소스 신경망 라이브러리
- Theano와 Tensorflow를 백엔드로 사용
- 직관적인 API로 다양한 신경망 구성이 가능
- Window, Mac, Linux 등 다양한 운영체제에서 작동가능



3. Keras를 이용한 DNN 구현(MNIST)



Keras 특징

1. 모듈화(Modularity)

: 신경망층, 비용 함수, 최적화, 초기화기법, 활성화 함수, 정규화기법은 모두 독립적인 모듈이며 이러한 모듈을 조합하여 모델을 만들 수 있다.

2. 최소주의(Minimalism)

: 각 모듈은 짧고 간결하며, 모든 코드는 한 번 훑어보는 것으로 이해 가능해야 한다. 단, 반복 속도와 혁신성은 떨어질 수 있다.

3. 쉬운 확장성

: 새로운 클래스나 함수로 모듈을 아주 쉽게 추가할 수 있고, 고급 연구에 필요한 다양한 표현을 할 수 있다.

4. 모델 데이터 구조

: 파이썬 코드로 모델이 정의되며, 케라스에서 제공하는 시퀀스 모델로 원하는 레이어를 쉽게 순차적으로 쌓을 수 있다.

3. Keras를 이용한 DNN 구현(MNIST)

Keras 설치 및 실습

```
$ pip install keras
```

- 윈도우환경 아나콘다에서 keras를 설치한다.

```
$ jupyter notebook
```

- Tensorflow와 마찬가지로 jupyter notebook을 실행하여 실습한다.

3. Keras를 이용한 DNN 구현(MNIST)

Wide + Deep Dropout을 이용한 구현 : 전체 코드 (1/4)

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import Adam
from keras.utils import np_utils

np.random.seed(1234)

training_cnt = 15
BATCH_SIZE = 100
N_HIDDEN = 512
```

3. Keras를 이용한 DNN 구현(MNIST)

Wide + Deep Dropout을 이용한 구현 : 전체 코드 (2/4)

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

X_train /= 255
X_test /= 255
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
```

3. Keras를 이용한 DNN 구현(MNIST)

Wide + Deep Dropout을 이용한 구현 : 전체 코드 (3/4)

```
model = Sequential()
model.add(Dense(N_HIDDEN, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.3))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dropout(0.3))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dropout(0.3))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dropout(0.3))
model.add(Dense(10))
model.add(Activation('softmax'))
model.summary()
```

3. Keras를 이용한 DNN 구현(MNIST)

Wide + Deep Dropout을 이용한 구현 : 전체 코드 (4/4)

```
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])

model.fit(X_train, Y_train,
        batch_size=BATCH_SIZE, epochs=training_cnt,
        verbose=1)

score = model.evaluate(X_test, Y_test, verbose=1)
print("\nTest cost:", score[0])
print('Test accuracy:', score[1])
```

3. Keras를 이용한 DNN 구현(MNIST)

모델 구축(Build graph) (1/6)

필요한 library import 하기

- Keras 확장 라이브러리에 있는 필요한 모듈을 불러온다.
- mnist : 사용할 mnist 데이터셋이 있는 모듈
- Sequential : 순차형 모델 생성에 필요한 모듈
- Dense : 입출력 연결, Dropout : dropout, activation : 활성화함수 모듈
- Adam : optimizer 모듈
- np_utils : one hot 인코딩 모듈

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout,
Activation
from keras.optimizers import Adam
from keras.utils import np_utils
```


3. Keras를 이용한 DNN 구현(MNIST)

모델 구축(Build graph) (2/6)

np.random.seed함수 사용

- 재현성을 위해 np.random.seed함수를 사용한다.

```
np.random.seed(1234)
```

학습을 위한 기초 파라미터 값 설정

- training_cnt(전체 데이터 셋 학습 횟수)를 15로 설정한다.
- BATCH_SIZE(데이터 셋 분할 단위)를 100으로 설정한다.
- N_HIDDEN(은닉층 노드 수)를 512로 Wide하게 설정한다.

```
training_cnt = 15  
BATCH_SIZE = 100  
N_HIDDEN = 512
```

3. Keras를 이용한 DNN 구현(MNIST)

모델 구축(Build graph) (3/6)

MNIST 데이터 셋 불러오기

- MNIST 학습용 데이터 셋과 검정용 데이터셋을 불러온다.

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

입력층 정의와 정규화 설정

- X_train은 60000개의 행으로 구성되고 784(28 X 28)개의 값을 갖는다.
- X_test는 마찬가지로 10000개의 행으로 구성
- 각 데이터는 GPU 연산을 지원하기 위해 float32타입으로 변환하고 정규화한다.
- 각 픽셀을 최대 강도값인 255로 나누어 [0,1]범위로 정규화한다.

```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
```

```
X_train /= 255
X_test /= 255
```

3. Keras를 이용한 DNN 구현(MNIST)

모델 구축(Build graph) (4/6)

X 변수 Shape 출력해보기

- 학습 데이터셋과 검정 데이터 셋의 행의 수를 출력한다.

```
print(X_train.shape[0], 'train samples')  
print(X_test.shape[0], 'test samples')
```

One-hot Encoding하기

- 0~9사이의 10개 Class의 출력값을 One-hot Encoding 한다.

```
Y_train = np_utils.to_categorical(y_train, 10)  
Y_test = np_utils.to_categorical(y_test, 10)
```

3. Keras를 이용한 DNN 구현(MNIST)

모델 구축(Build graph) (5/6)

신경망 정의하기

- 심층 신경망을 만들기위해 Sequential 객체인 model을 정의하고 층층이 쌓는다.
- 입력층의 차원을 설정해주고 활성화 함수는 ReLU를 사용하였다.
- Xavier 가중치 초기화는 Default 설정돼있으므로 따로 정의하지 않는다.
- Dropout(0.3)은 학습 시 제외할 30%의 노드를 의미한다. (70%의 노드는 유지)
- 마지막 층의 활성화 함수는 Softmax, 노드 수는 10개로 정의한다.

```
model = Sequential()
model.add(Dense(N_HIDDEN, input_dim=784))
model.add(Activation('relu'))
model.add(Dropout(0.3))
model.add(Dense(N_HIDDEN))
                                :
model.add(Activation('relu'))
model.add(Dropout(0.3))
model.add(Dense(10))
model.add(Activation('softmax'))
model.summary()
```

3. Keras를 이용한 DNN 구현(MNIST)

모델 구축(Build graph) (6/6)

Loss function, Optimizer, Metric 정의하기

- 모델의 Loss function으로 멀티 레이블 예측에 적합한 범주형 크로스 엔트로피 함수를 사용한다.
- 최적화방법으로 Adam Optimizer를 사용, Learning rate는 0.001로 default다.
- 모델 평가 항목(Metric)으로 Accuracy(Y값을 정확히 예측한 비율)를 사용한다.

```
model.compile(loss='categorical_crossentropy',  
              optimizer=Adam(),  
              metrics=['accuracy'])
```

모델 학습시키기

- 모델을 학습시키기 위해서 model.fit 함수를 사용한다.
- 모델에 학습에 사용할 데이터로 학습 데이터 셋(Train)을 지정한다.
- Epochs와 batch_size, validation_split은 앞서 정의했던 파라미터 값을 사용
- verbose의 세가지 옵션(0: 과정 생략, 1: 과정 보기, 2: 횟수와 loss만 확인)

```
model.fit(X_train, Y_train,  
         batch_size=BATCH_SIZE, epochs=training_cnt, verbose=1)
```

3. Keras를 이용한 DNN 구현(MNIST)

모델 검정(test)

모델 평가하기

- 모델의 평가를 위해서 model.evaluate 함수를 사용한다.
- 데이터로 시험용 데이터 셋(Test)을 사용한다. verbose=1로 평가 과정을 출력
- score[0]은 검정의 Loss, score[1]은 검정의 Accuracy를 출력한다.

```
score = model.evaluate(X_test, Y_test, verbose=1)
print("\nTest Loss:", score[0])
print('Test accuracy:', score[1])
```

3. Keras를 이용한 DNN 구현(MNIST)

학습 결과 (1/2)

60000 train samples
10000 test samples

60000개의 train 데이터 셋과 10000개의 test 데이터 셋

Layer (type)	Output Shape	Param #
=====		
dense_47 (Dense)	(None, 512)	401920
activation_47 (Activation)	(None, 512)	0
dropout_37 (Dropout)	(None, 512)	0
⋮		
dense_51 (Dense)	(None, 10)	5130
activation_51 (Activation)	(None, 10)	0
=====		
Total params: 1,195,018		출력층 노드 수는 0~9사이의 Y값인 10개의 Class
Trainable params: 1,195,018		
Non-trainable params: 0		

3. Keras를 이용한 DNN 구현(MNIST)

학습 결과 (2/2)

Epoch 1/15

60000/60000 [=====] - 30s 498us/step - loss:
0.3064 - acc: 0.9054

Epoch 2/15

60000/60000 [=====] - 30s 506us/step - loss:
0.1433 - acc: 0.9581

⋮

Epoch 14/15

60000/60000 [=====] - 27s 456us/step - loss:
0.0450 - acc: 0.9870

Epoch 15/15

60000/60000 [=====] - 28s 462us/step - loss:
0.0464 - acc: 0.9869
10000/10000 [=====] - 1s 85us/step

Test Loss: 0.07134897730251114

Test accuracy: 0.9825

Epoch는 15번, Loss: 비용, acc: Y값을 정확히 예측한 비율,
val_loss: validation 데이터 셋의 비용,
val_acc: validation 데이터 셋의 accuracy
Test Loss: Test 데이터 셋의 비용,
Test accuracy: Test 데이터 셋의 accuracy

감사합니다.

참고자료(Reference)

“모두를 위한 딥러닝 강좌 시즌 1”, 김성훈

1. MNIST 문자인식

<http://solarisailab.com/archives/303>

<https://localab.jp/blog/simple-neural-network-using-tensorflow/>

2. Keras 소개 및 특징

<https://blog.naver.com/sundooedu/221315683165>

3. Keras 이미지

<https://mparsec.com/wp-content/uploads/2017/07/anacondafeaturedimg.png>