Term Project – MNIST 성능 향상 (오류 1% 이하로 개선)

컴퓨터과학부 2018920031 유승리 | 인공지능 | 과제 #4

1. 개요

> loss: nan이 출력되는 문제점 개선

이번 프로젝트에서 다루는 문제의 유형은 multi class classification이기 때문에 objective function으로 categorical cross entropy를 사용한다. categorical cross entropy의 식은 다음과 같다.

$$E(w) = -\sum_{n=1}^{N}\sum_{k=1}^{K} d_{nk} \log y_k(x_n; w)$$

이와 같이, 이 식에는 log가 포함되어 log의 진수로는 양수만 올 수 있기 때문에, 교재의 코드 중 loss를 구하는 식인

```
loss = -tf.reduce_sum(t * tf.log(p))
```

에서 p는 양수여야 한다. 하지만 많은 수의 iteration을 반복하게 되면, p값이 작아져 양수의 범위를 벗어나는 경우가 생기기 때문에 loss: nan이 출력되는 것이다. 따라서 p의 범위를 양수(대략 1e-10 ~ 1.0)로 정해주기 위해서

```
loss = -tf.reduce_sum(t * tf.log(tf.clip_by_value(p, 1e-10, 1.0)))
```

위와 같은 코드로 바꾸었다.

> 이번 과제에서의 필수 요소 반영

- 모든 경우의 filter size는 3×3이다.
- 모든 경우의 convolution hidden layer를 1개 추가하여, hidden layer를 3개 만들었다.

2. 개별 요소의 조작에 의한 성능 비교 (50,000회 학습)

신경망이 복잡해질수록 batch size를 감소시켜야 하므로, 기존 교재 코드의 batch size인 50 대신 32를 이용하였다. 또한 각 경우에서 나머지 조건은 동일하게 유지하였다.

> node 개수에 따른 성능 비교

node 개수	최대 accuracy	해당 epoch
512	0.992300	49500
1024	**0.993700**	**35000**
2048	0.993100	50000

> filter의 개수에 따른 성능 비교 (hidden layer 3개)

filter 개수	최대 accuracy	해당 epoch
32-64-32	**0.993900**	**37000**
32-16-128	0.993000	47500
32-64-128	0.993000	50000
128-32-64	**0.993900**	**35000**
64-32-16	0.991900	46500
64-128-128	**0.994400**	**36000**
32-64-64	0.993100	38500
32-32-128	0.992500	35000

> activation function (fully connected layer)

activation function	최대 accuracy	해당 epoch
ReLU	**0.993900**	**37000**
tanh	0.992400	50000
leaky ReLU	0.993200	47500
sigmoid	0.991900	49000

> activation function (hidden layer의 cutoff)

activation function	최대 accuracy	해당 epoch
ReLU	**0.993900**	**37000**
leaky ReLU	0.992600	50000

> training 시 dropout 비율 (keep_prob)

keep_prob	최대 accuracy	해당 epoch
0.5	**0.993900**	**37000**
0.25	0.993700	45000
0.75	0.993200	49500

> hidden layer의 b_conv

b_conv	최대 accuracy	해당 epoch
0.1-0.1-0.1	**0.993900**	**37000**
0.2-0.2-0.2	0.993000	48000
0.05-0.05-0.05	0993700	46000

> learning rate

learning rate	최대 accuracy	해당 epoch
0.0001	**0.993900**	**37000**
0.0002	0.993700	41000
0.00005	0.992100	45000

3. 여러 요소의 조작을 통해 산출한 최대 성능 (200,000회 학습)

위에서 비교한 결과를 토대로 여러 요소를 조작하여 수 회의 실험을 통해 발견한 4가지의 조합(수정 코드 ①~④)은 다음 표와 같다. 교재에서 제공된 코드에서 filter size를 3×3으로 조정한 코드와 각 수정 코드 조합에 대해서 3회씩 실험을 진행하였으며, 파란색 음영에 해당하는 값은 그 평균값이다. 따라서 산출된 최대 성능은 **수정 코드 ②**에서였다.

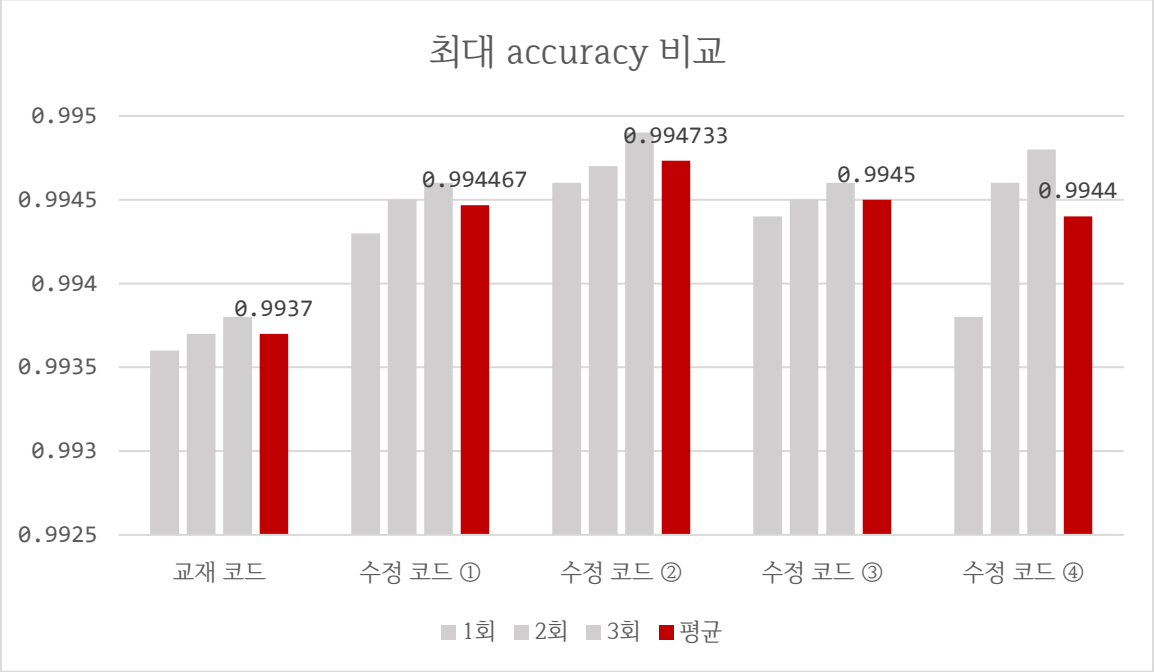
단, 다음 표에서의 값들은 500회의 학습마다 출력한 값이므로 아주 정확한 것은 아니기 때문에, 3회씩 실험을 진행하여 이 점을 보완하고자 하였다.

> 비교

	교재에서 제공된 코드 (filter 3×3)	수정 코드 ①	수정 코드 ②	수정 코드 ③	수정 코드 ④
최대 accuracy / 해당 step	0.993600 / 182000	0.994300 / 197500	0.994600 / 170000	0.994400 / 160000	0.993800 / 104500
	0.993700 / 192000	0.994500 / 145000	0.994700 / 197000	0.994500 / 182500	0.994600 / 126500
	0.993800 / 142500	0.994600 / 145500	**0.994900** / 153500	0.994600 / 167500	0.994800 / 196500
	0.993700	0.994467	**0.994733**	0.994500	0.994400
교재 코드의 최대 accuracy의 평균 (0.993700)이상의 값이 출력된 첫 step		82500	65500	124500	104500
		45000	79500	107500	74500
		100000	55500	96500	63000
		75833	**66833**	109500	80667
filter size	3×3	3×3			
hidden layer 개수	2	3			
batch size	50	32			
node 개수	1024	1024	512	1024	512
filter 개수	32-64	64-128-128			
activation function (fully connected)	ReLU	ReLU		ReLU	
activation function (hidden cutoff)	ReLU	ReLU		leaky ReLU	
training 시 keep_prob	0.5	0.7			
b_conv	0.1	0.1			
learning rate	0.0001	0.0001			

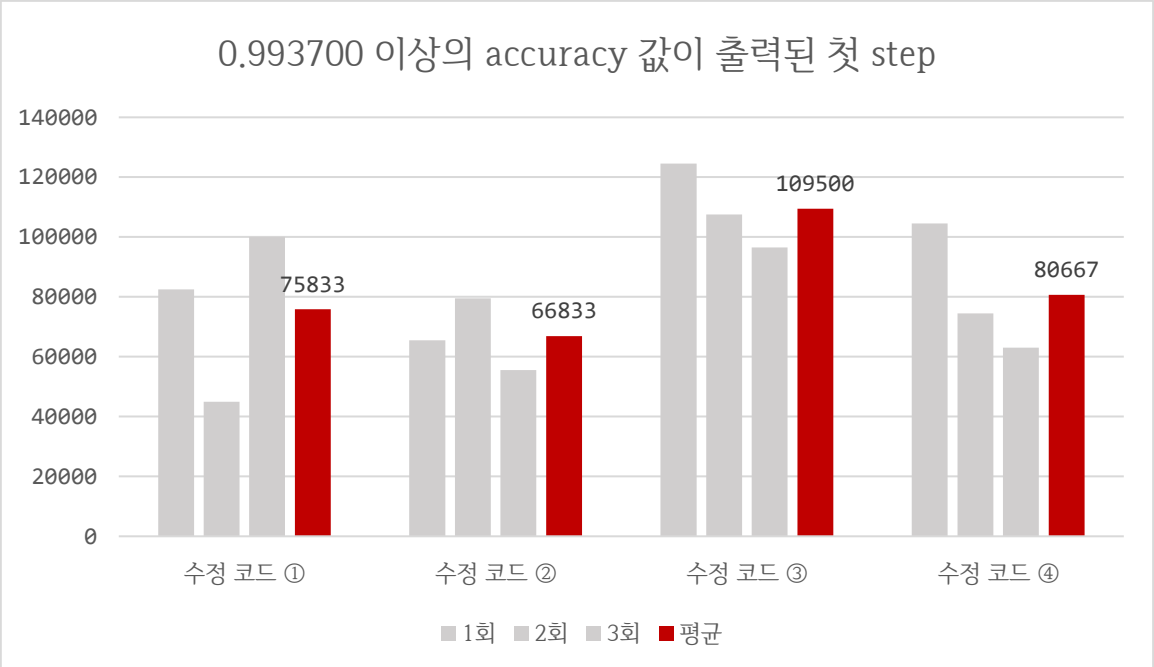
> 관련 그래프

- 최대 accuracy 비교 그래프



⇒ 수정 코드 ②의 평균 최대 accuracy가 가장 높다.

- 교재 코드의 최대 accuracy의 평균(0.993700)이상의 값이 출력된 첫 step 그래프



⇒ 수정 코드 ②에서 0.993700 이상의 accuracy 값이 제일 적은 학습 횟수만에 출력되었다.

> 수정 코드 ②

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data

np.random.seed(20160704)
tf.set_random_seed(20160704)
```

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

```
num_filters1 = 64

x = tf.placeholder(tf.float32, [None, 784])
x_image = tf.reshape(x, [-1,28,28,1])

W_conv1 = tf.Variable(tf.truncated_normal([3,3,1,num_filters1], # filter size 3X3
                                  stddev=0.1))
h_conv1 = tf.nn.conv2d(x_image, W_conv1,
                   strides=[1,1,1,1], padding='SAME')

b_conv1 = tf.Variable(tf.constant(0.1, shape=[num_filters1]))
h_conv1_cutoff = tf.nn.relu(h_conv1 + b_conv1)

h_pool1 = tf.nn.max_pool(h_conv1_cutoff, ksize=[1,2,2,1],
                     strides=[1,2,2,1], padding='SAME')
```

```
num_filters2 = 128

W_conv2 = tf.Variable(
          tf.truncated_normal([3,3,num_filters1,num_filters2], # filter size 3X3
                              stddev=0.1))
h_conv2 = tf.nn.conv2d(h_pool1, W_conv2,
                   strides=[1,1,1,1], padding='SAME')

b_conv2 = tf.Variable(tf.constant(0.1, shape=[num_filters2]))
h_conv2_cutoff = tf.nn.relu(h_conv2 + b_conv2)

h_pool2 = tf.nn.max_pool(h_conv2_cutoff, ksize=[1,2,2,1],
                     strides=[1,2,2,1], padding='SAME')
```

```
# additional hidden layer
num_filters3 = 128

W_conv3 = tf.Variable(
          tf.truncated_normal([3,3,num_filters2,num_filters3], # filter size 3X3
                              stddev=0.1))
h_conv3 = tf.nn.conv2d(h_pool2, W_conv3,
                   strides=[1,1,1,1], padding='SAME')

b_conv3 = tf.Variable(tf.constant(0.1, shape=[num_filters3]))
h_conv3_cutoff = tf.nn.relu(h_conv3 + b_conv3)

h_pool3 = tf.nn.max_pool(h_conv3_cutoff, ksize=[1,2,2,1],
                     strides=[1,2,2,1], padding='SAME')
```

```
# fully connected layer, dropout layer, softmax function
h_pool3_flat = tf.reshape(h_pool3, [-1, 4*4*num_filters3])

num_units1 = 4*4*num_filters3 # fully connected layer에 입력할 데이터 개수
num_units2 = 512 # fully connected layer의 node 개수

w2 = tf.Variable(tf.truncated_normal([num_units1, num_units2]))
b2 = tf.Variable(tf.constant(0.1, shape=[num_units2]))
hidden2 = tf.nn.relu(tf.matmul(h_pool3_flat, w2) + b2)

keep_prob = tf.placeholder(tf.float32) # dropout probability
hidden3_drop = tf.nn.dropout(hidden2, keep_prob)

w0 = tf.Variable(tf.zeros([num_units2, 10]))
b0 = tf.Variable(tf.zeros([10]))
p = tf.nn.softmax(tf.matmul(hidden3_drop, w0) + b0)
```

```
t = tf.placeholder(tf.float32, [None, 10])
loss = -tf.reduce_sum(t * tf.log(tf.clip_by_value(p, 1e-10, 1.0)))
train_step = tf.train.AdamOptimizer(0.0001).minimize(loss)
correct_prediction = tf.equal(tf.argmax(p, 1), tf.argmax(t, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

```
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()
```

```
i = 0
for _ in range(200000):
    i += 1
    batch_xs, batch_ts = mnist.train.next_batch(32) # 신경망이 복잡해질수록 작은 batch size
    sess.run(train_step,
             feed_dict={x:batch_xs, t:batch_ts, keep_prob:0.7}) # training 시 parameter
최적화
    if i % 500 == 0:
        loss_vals, acc_vals = [], []
        for c in range(4):
            start = len(mnist.test.labels) // 4 * c
            end = len(mnist.test.labels) // 4 * (c+1)
            loss_val, acc_val = sess.run([loss, accuracy],
                feed_dict={x:mnist.test.images[start:end],
                           t:mnist.test.labels[start:end],
                           keep_prob:1.0})  # parameter 최적화 완료 후 미지의 데이터에 대한 예측할
때 (test 시에는 모두 사용)
            loss_vals.append(loss_val)
            acc_vals.append(acc_val)
        loss_val = np.sum(loss_vals)
        acc_val = np.mean(acc_vals)
        print ('Step: %d, Loss: %f, Accuracy: %f'
               % (i, loss_val, acc_val))
        saver.save(sess, 'cnn_session', global_step=i)
```

> 수정 코드 ② 출력 결과 (최대 accuracy: 0.994900)

```
Step: 151000, Loss: 420.222534, Accuracy: 0.993800
Step: 151500, Loss: 440.065247, Accuracy: 0.993700
Step: 152000, Loss: 405.952393, Accuracy: 0.994000
Step: 152500, Loss: 418.306213, Accuracy: 0.994000
Step: 153000, Loss: 450.434052, Accuracy: 0.993800
Step: 153500, Loss: 371.392639, Accuracy: 0.994900
Step: 154000, Loss: 402.801147, Accuracy: 0.994100
Step: 154500, Loss: 376.180969, Accuracy: 0.994100
Step: 155000, Loss: 384.539307, Accuracy: 0.994000
Step: 155500, Loss: 464.982605, Accuracy: 0.993700
Step: 156000, Loss: 486.717865, Accuracy: 0.992800
```