

**Term Project: MNIST 성능 향상**

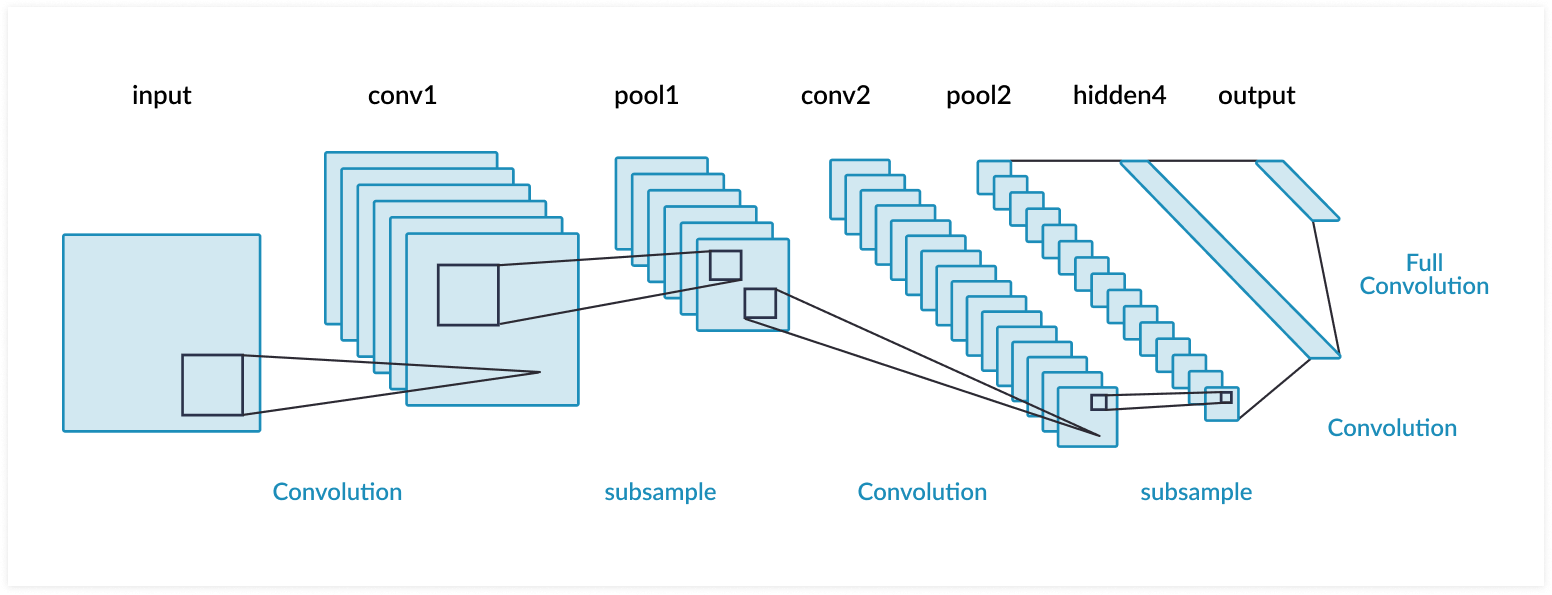
컴퓨터과학부

2016920004

권은진

1. 개요
   1. 목표: Test data에 대하여 오류 1% 이하로 개선하기
   2. 실험 내용: 책 “텐서플로로 시작하는 딥러닝”의 5장 소스코드를 참고하였으며, 성능을 개선할 수 있는 방법을 아래 사항을 고려하여 고찰하였다.

* Convolutional Hidden Layer 개수
* Learning Rate
* Fully Connected Layer 의 노드 개수
* Dropout 의 rate
* mini-batch 학습에서의 batch size
* Dropout 이 배치된 위치
* Fully Connected Layer 에서 활성화 함수 사용 유무
  1. 초기 다층 신경망 아키텍처



* 필터 크기 3x3, 첫번째 합성곱 신경망에서의 필터 개수는 32개, 두번째 신경망에서의 필터 개수는 64개이다.
* 사용한 최적화 함수는 AdamOptimizer() 로, 학습률을 0.0001로 지정하였다.
* Fully Connected Layer 의 노드 개수는 1024개이다.
* Dropout 은 Fully Connected Layer에 한 번만 수행되며, rate은 0.5이다.
* mini-batch 학습에서의 batch size는 50이다.
* 초기 가중치는 표준편차 0.1을 갖도록 난수를 생성하여 지정하였으며, 난수의 시드값은 실험 당일 날짜(20191201)이다.
* Fully Connected Layer 에서는 활성화 함수 사용로 relu를 사용한다.
* 초기 반복횟수는 2000이다.
  1. 초기 소스코드

|  |
| --- |
| import tensorflow as tf  import numpy as np  import matplotlib.pyplot as plt  from tensorflow.examples.tutorials.mnist import input\_data  np.random.seed(20191201)  tf.set\_random\_seed(20191201)  # Input Layer  mnist = input\_data.read\_data\_sets("./data/", one\_hot=True)  x = tf.placeholder(tf.float32, [None, 784])  x\_image = tf.reshape(x, [-1,28,28,1])  # Hidden Layer 1  W\_conv1 = tf.Variable(tf.truncated\_normal([3,3,1,32], 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=[32]))  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')  # Hidden Layer 2  W\_conv2 = tf.Variable(tf.truncated\_normal([3,3,32,64], 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=[64]))  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')  # Fully Connected Layer  num\_units1 = 7\*7\*64  num\_units2 = 1024  h\_pool\_flat = tf.reshape(h\_pool2, [-1, num\_units1])  w2 = tf.Variable(tf.truncated\_normal([num\_units1, num\_units2]))  b2 = tf.Variable(tf.constant(0.1, shape=[num\_units2]))  FC = tf.nn.relu(tf.matmul(h\_pool2\_flat, w2) + b2)  keep\_prob = tf.placeholder(tf.float32)  FC\_drop = tf.nn.dropout(FC, keep\_prob)  # Output Layer  w0 = tf.Variable(tf.zeros([num\_units2, 10]))  b0 = tf.Variable(tf.zeros([10]))  p = tf.nn.softmax(tf.matmul(FC\_drop, w0) + b0)  # Optimizer & Evaluator  t = tf.placeholder(tf.float32, [None, 10])  loss = -tf.reduce\_sum(t \* tf.log(p))  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))  # Training  sess = tf.InteractiveSession()  sess.run(tf.global\_variables\_initializer())  for i in range(1, 2001):  batch\_xs, batch\_ts = mnist.train.next\_batch(50)  sess.run(train\_step,feed\_dict={x:batch\_xs, t:batch\_ts, keep\_prob:0.5})  if i % 100 == 0:  loss\_vals, acc\_vals = [], []  for c in range(4):  start = int(len(mnist.test.labels) / 4 \* c)  end = int(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})  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)) |

* 1. 초기 결과

|  |
| --- |
| Step: 100, Loss: 5082.247070, Accuracy: 0.872700  Step: 200, Loss: 3290.780273, Accuracy: 0.906800  Step: 300, Loss: 2661.613770, Accuracy: 0.925000  Step: 400, Loss: 2259.271973, Accuracy: 0.931700  Step: 500, Loss: 2052.433105, Accuracy: 0.939300  Step: 600, Loss: 1868.204346, Accuracy: 0.945200  Step: 700, Loss: 1724.504395, Accuracy: 0.950800  Step: 800, Loss: 1561.319702, Accuracy: 0.954000  Step: 900, Loss: 1497.562622, Accuracy: 0.954600  Step: 1000, Loss: 1357.353027, Accuracy: 0.960600  Step: 1100, Loss: 1269.200928, Accuracy: 0.962200  Step: 1200, Loss: 1214.817017, Accuracy: 0.963500  Step: 1300, Loss: 1212.025269, Accuracy: 0.963600  Step: 1400, Loss: 1146.572754, Accuracy: 0.966200  Step: 1500, Loss: 1085.940796, Accuracy: 0.967300  Step: 1600, Loss: 1011.688293, Accuracy: 0.971000  Step: 1700, Loss: 1018.463623, Accuracy: 0.970500  Step: 1800, Loss: 980.152405, Accuracy: 0.971200  Step: 1900, Loss: 923.292725, Accuracy: 0.973000  Step: 2000, Loss: 920.533447, Accuracy: 0.971300 |

1. 실행 결과
   1. Convolutional Hidden Layer 개수를 늘려 정확도 개선하기
      1. 필터가 64개인 3번째 층을 추가한 경우

|  |
| --- |
| ''' 생략 '''  # Hidden Layer 3  W\_conv3 = tf.Variable(tf.truncated\_normal([3,3,64,128], 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=[128]))  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  num\_units1 = 4\*4\*128  num\_units2 = 1024  h\_pool\_flat = tf.reshape(h\_pool3, [-1, num\_units1])  ''' 생략 ''' |
| Step: 100, Loss: 4992.253418, Accuracy: 0.868600  Step: 200, Loss: 2807.954346, Accuracy: 0.923700  Step: 300, Loss: 2004.146851, Accuracy: 0.944100  ...  Step: 1800, Loss: 749.662476, Accuracy: 0.976000  Step: 1900, Loss: 671.675415, Accuracy: 0.977700  Step: 2000, Loss: 663.901123, Accuracy: 0.978500 |

* + 1. 필터가 128개인 4번째 층을 추가한 경우

|  |
| --- |
| ''' 생략 '''  # Hidden Layer 4  W\_conv4 = tf.Variable(tf.truncated\_normal([3,3,128,256], stddev=0.1))  h\_conv4 = tf.nn.conv2d(h\_pool3, W\_conv4, strides=[1,1,1,1], padding='SAME')  b\_conv4 = tf.Variable(tf.constant(0.1, shape=[256]))  h\_conv4\_cutoff = tf.nn.relu(h\_conv4 + b\_conv4)  h\_pool4 = tf.nn.max\_pool(h\_conv4\_cutoff, ksize=[1,2,2,1],  strides=[1,2,2,1], padding='SAME')  # Fully Connected Layer  num\_units1 = 2\*2\*256  num\_units2 = 1024  h\_pool\_flat = tf.reshape(h\_pool4, [-1, num\_units1])  ''' 생략 ''' |
| Step: 100, Loss: 4799.766113, Accuracy: 0.875000  Step: 200, Loss: 2877.061035, Accuracy: 0.917700  Step: 300, Loss: 1990.244141, Accuracy: 0.941200  ...  Step: 1800, Loss: 610.434692, Accuracy: 0.981100  Step: 1900, Loss: 740.382324, Accuracy: 0.975900  Step: 2000, Loss: 650.583008, Accuracy: 0.979400 |

* 1. Learning Rate 를 변화시켜 정확도 개선하기
     1. loss가 2000회의 반복에서는 천천히 낮아지는 것 같다고 판단하여, learning rate 을 0.001로 수정하였다.
     2. 그 외의 신경망 구조는 이전 단계인 a.의 두번째와 동일하다.

|  |
| --- |
| ''' 생략 '''  train\_step = tf.train.AdamOptimizer(0.001).minimize(loss)  ''' 생략 ''' |
| Step: 100, Loss: 2352.456299, Accuracy: 0.927200  Step: 200, Loss: 2185.607422, Accuracy: 0.930600  Step: 300, Loss: 1293.315186, Accuracy: 0.958700  ...  Step: 1800, Loss: 419.709534, Accuracy: 0.985400  Step: 1900, Loss: 689.691650, Accuracy: 0.979200  Step: 2000, Loss: 364.407593, Accuracy: 0.988900 |

* 1. Fully Connected Layer 의 노드 개수를 변화시켜 정확도 개선하기
     1. 은닉층이 증가하면서 연산량이 많아졌기 때문에, Fully Connected Layer의 노드 개수를 1024개에서 512개로 줄였다.
     2. 정확도를 개선하기 보다는 동일한 정확도라도 보다 빠르게 결과가 나오는 것을 목적으로 하였다.
     3. 그 외의 신경망 구조는 이전 단계인 b. 와 동일하다.
     4. loss 가 전보다 감소하는 폭이 커졌고, 2000번 학습 후 loss가 전보다 개선되었기에 계속해서 노드 개수를 512개로 지정하였다.

|  |
| --- |
| ''' 생략 '''  # Fully Connected Layer  num\_units1 = 7\*7\*64  num\_units2 = 512 # modified  h\_pool\_flat = tf.reshape(h\_pool2, [-1, num\_units1])  w2 = tf.Variable(tf.truncated\_normal([num\_units1, num\_units2]))  b2 = tf.Variable(tf.constant(0.1, shape=[num\_units2]))  FC = tf.nn.relu(tf.matmul(h\_pool2\_flat, w2) + b2)  keep\_prob = tf.placeholder(tf.float32)  FC\_drop = tf.nn.dropout(FC, keep\_prob)  ''' 생략 ''' |
| Step: 100, Loss: 2464.836914, Accuracy: 0.923100  Step: 200, Loss: 1466.890137, Accuracy: 0.954700  Step: 300, Loss: 1623.952881, Accuracy: 0.950500  ...  Step: 1800, Loss: 550.432434, Accuracy: 0.983000  Step: 1900, Loss: 653.821228, Accuracy: 0.979500  Step: 2000, Loss: 360.496460, Accuracy: 0.988300 |

* 1. Dropout 의 rate을 변화시켜 정확도 개선하기
     1. 기존 dropout 의 rate은 0.5로 전결합층에서 절반정도 노드를 선택해서 학습을 진행한다. rate을 낮추는 것보다 높이는 경우 더 높은 정확도를 보였으며, 이후 실험에서도 dropout 의 rate을 0.7로 지정했다.
     2. 참고로, 같은 구조에서 dropout의 rate만 변경하였을 때 0.6, 0.7, 0.8 각각 실험하였을 때 loss는 0.7일 때가 제일 낮았다.
     3. 또한, 정확도는 전보다 낮았지만 2000번의 반복과정에서 100회 단위로 측정된 loss는 전보다 개선되었다.
     4. 그 외의 신경망 구조는 이전 단계인 c. 와 동일하다.

|  |
| --- |
| ''' 생략 '''  for i in range(1, 2001):  batch\_xs, batch\_ts = mnist.train.next\_batch(50)  sess.run(train\_step,feed\_dict={x:batch\_xs, t:batch\_ts, keep\_prob:0.5})  ''' 생략 ''' |
| Step: 100, Loss: 2568.578369, Accuracy: 0.923200  Step: 200, Loss: 1441.291992, Accuracy: 0.953700  Step: 300, Loss: 1044.123291, Accuracy: 0.966100  ...  Step: 1800, Loss: 476.119019, Accuracy: 0.985200  Step: 1900, Loss: 487.455933, Accuracy: 0.983700  Step: 2000, Loss: 351.488953, Accuracy: 0.987600 |

* 1. mini-batch 학습에서의 batch size를 변화시켜 정확도 개선하기
     1. 특별한 이유가 있기보다, 기존의 batch size가 50인 것이 작은 것 같아서 크기를 조금씩 늘려봤는데 전보다 높은 정확도를 얻게 되었다.
     2. 이후 실행에서도 모두 batch size 를 100으로 하였다.

|  |
| --- |
| ''' 생략 '''  for i in range(1, 2001):  batch\_xs, batch\_ts = mnist.train.next\_batch(100)  ''' 생략 ''' |
| Step: 100, Loss: 1748.815918, Accuracy: 0.946900  Step: 200, Loss: 1133.622070, Accuracy: 0.965400  Step: 300, Loss: 1085.177490, Accuracy: 0.965800  ...  Step: 1800, Loss: 322.225616, Accuracy: 0.990200  Step: 1900, Loss: 290.678894, Accuracy: 0.991500  Step: 2000, Loss: 330.809570, Accuracy: 0.988800 |

* 1. Dropout 이 배치된 위치를 변화시켜 정확도 개선하기
     1. 첫번째, 세번째 풀링 계층 이후에 각각 Dropout을 배치한 경우: loss의 변화가 가장 컸던 실험으로, 처음에는 저조한 정확도였으나 2000번째 학습을 진행했을 때 가장 높은 정확도가 나타났다. 그러나 2000번을 넘어가면 정확도가 다시 낮아진다.

|  |
| --- |
| ''' 생략 '''  # Hidden Layer 1  W\_conv1 = tf.Variable(tf.truncated\_normal([3,3,1,32], 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=[32]))  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')  L1 = tf.nn.dropout(h\_pool1, keep\_prob=keep\_prob)  ''' 생략 '''  # Hidden Layer 3  W\_conv3 = tf.Variable(tf.truncated\_normal([3,3,64,128], 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=[128]))  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')  L3 = tf.nn.dropout(h\_pool3, keep\_prob=keep\_prob)  ''' 생략 ''' |
| Step: 100, Loss: 7450.582520, Accuracy: 0.735100  Step: 200, Loss: 2434.583740, Accuracy: 0.926300  Step: 300, Loss: 1376.474243, Accuracy: 0.957800  ...  Step: 1800, Loss: 382.171295, Accuracy: 0.987400  Step: 1900, Loss: 335.686951, Accuracy: 0.988600  Step: 2000, Loss: 299.449768, Accuracy: 0.989900 |
| Step: 2100, Loss: 359.526215, Accuracy: 0.987900  Step: 2200, Loss: 381.155945, Accuracy: 0.987600  Step: 2300, Loss: 371.269196, Accuracy: 0.988100  ...  Step: 3800, Loss: 249.691040, Accuracy: 0.991800  Step: 3900, Loss: 368.477753, Accuracy: 0.988700  Step: 4000, Loss: 323.174072, Accuracy: 0.988900 |

* + 1. 두번째, 네번째 풀링 계층 이후에 각각 Dropout을 배치한 경우: 배치 전의 실험보다 정확도가 높게 나왔으며, 2000번을 넘어가도 계속해서 loss가 이전 실험보다 상대적으로 줄어들었다.

|  |
| --- |
| ''' 생략 '''  # Hidden Layer 2  W\_conv2 = tf.Variable(tf.truncated\_normal([3,3,32,64], 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=[64]))  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')  L2 = tf.nn.dropout(h\_pool2, keep\_prob=keep\_prob)  ''' 생략 '''  # Hidden Layer 4  W\_conv4 = tf.Variable(tf.truncated\_normal([3,3,128,256], stddev=0.1))  h\_conv4 = tf.nn.conv2d(h\_pool3, W\_conv4, strides=[1,1,1,1], padding='SAME')  b\_conv4 = tf.Variable(tf.constant(0.1, shape=[256]))  h\_conv4\_cutoff = tf.nn.relu(h\_conv4 + b\_conv4)  h\_pool4 = tf.nn.max\_pool(h\_conv4\_cutoff, ksize=[1,2,2,1],  strides=[1,2,2,1], padding='SAME')  L4 = tf.nn.dropout(h\_pool4, keep\_prob=keep\_prob)  ''' 생략 ''' |
| Step: 100, Loss: 4543.503906, Accuracy: 0.892400  Step: 200, Loss: 1551.728516, Accuracy: 0.951800  Step: 300, Loss: 1208.570068, Accuracy: 0.963200  ...  Step: 1800, Loss: 358.361877, Accuracy: 0.987900  Step: 1900, Loss: 308.389343, Accuracy: 0.989800  Step: 2000, Loss: 339.900330, Accuracy: 0.989200 |
| Step: 2100, Loss: 333.137543, Accuracy: 0.989400  Step: 2200, Loss: 377.446075, Accuracy: 0.988800  Step: 2300, Loss: 329.307098, Accuracy: 0.989800  ...  Step: 3700, Loss: 271.494049, Accuracy: 0.990600  Step: 3800, Loss: 310.579865, Accuracy: 0.989700  Step: 3900, Loss: 374.938293, Accuracy: 0.988700  Step: 4000, Loss: 314.594360, Accuracy: 0.989900 |

* + 1. 은닉층에 모두 Dropout 을 배치한 경우: 은닉층에 배치한 경우들 중 가장 높은 loss를 보였으며, 이를 통해 Dropout을 모두 배치하는 것보다 부분적으로 배치하는 것이 더 좋다고 판단하였다.

|  |
| --- |
| Step: 100, Loss: 22345.871094, Accuracy: 0.197100  Step: 200, Loss: 8158.172363, Accuracy: 0.780200  Step: 300, Loss: 3097.824219, Accuracy: 0.918200  ...  Step: 1800, Loss: 451.913147, Accuracy: 0.984700  Step: 1900, Loss: 487.760101, Accuracy: 0.983700  Step: 2000, Loss: 398.861694, Accuracy: 0.988000 |

* + 1. Fully Connected Layer 에 Dropout을 배치하지 않은 경우: 이전 실험 “두번째, 네번째 풀링 계층 이후에 각각 Dropout을 배치한 경우”에서 Fully Connected Layer 에 Dropout만 제거한 실험이다. 확실히 전보다 개선된 정확도를 보였다. 반복횟수를 4000회로 늘릴 경우 목표로 두었던 오차 1% 미만을 달성할 수 있었다.

|  |
| --- |
| ''' 생략 '''  # Fully Connected Layer  num\_units1 = 2\*2\*256  num\_units2 = 512  L4\_flat = tf.reshape(L4, [-1, num\_units1])  w1 = tf.Variable(tf.truncated\_normal([num\_units1, num\_units2]))  b1 = tf.Variable(tf.zeros([num\_units2]))  FC = tf.nn.relu(tf.matmul(L4\_flat, w1) + b1)  ''' 생략 ''' |
| Step: 100, Loss: 4865.157227, Accuracy: 0.866400  Step: 200, Loss: 1691.621338, Accuracy: 0.943400  Step: 300, Loss: 1075.843140, Accuracy: 0.966500  ...  Step: 1800, Loss: 307.249268, Accuracy: 0.989700  Step: 1900, Loss: 311.040283, Accuracy: 0.989700  Step: 2000, Loss: 303.228363, Accuracy: 0.990500 |
| Step: 2100, Loss: 358.492218, Accuracy: 0.988100  Step: 2200, Loss: 323.465485, Accuracy: 0.989000  Step: 2300, Loss: 321.490906, Accuracy: 0.988800  ...  Step: 3800, Loss: 238.506287, Accuracy: 0.992100  Step: 3900, Loss: 265.360718, Accuracy: 0.992200  Step: 4000, Loss: 241.803360, Accuracy: 0.992300 |

* 1. Fully Connected Layer 에서 활성화 함수 사용 유무에 따른 정확도 변화
     1. Fully Connected Layer 에 활성화 함수 및 Dropout을 배치하지 않은 경우: 이전 실험 “두번째, 네번째 풀링 계층 이후에 각각 Dropout을 배치한 경우”에서 Fully Connected Layer 에 활성화 함수 및 Dropout을 모두 제거한 실험이다. 지금까지의 실험 중 가장 높은 정확도를 보였다. 반복횟수를 늘릴 경우 loss가 지속적으로 큰 변화 없이 감소하는 모습을 보였으며, 정확도는 최대 99.3% 가 되었다.

|  |
| --- |
| ''' 생략 '''  # Fully Connected Layer  num\_units1 = 2\*2\*256  num\_units2 = 512  L4\_flat = tf.reshape(L4, [-1, num\_units1])  w1 = tf.Variable(tf.truncated\_normal([num\_units1, num\_units2]))  b1 = tf.Variable(tf.zeros([num\_units2]))  FC = tf.matmul(L4\_flat, w1) + b1  ''' 생략 ''' |
| Step: 100, Loss: 3813.618164, Accuracy: 0.889700  Step: 200, Loss: 1340.638550, Accuracy: 0.957700  Step: 300, Loss: 1017.487305, Accuracy: 0.966500  ...  Step: 1800, Loss: 329.876068, Accuracy: 0.988900  Step: 1900, Loss: 253.664368, Accuracy: 0.991300  Step: 2000, Loss: 248.288635, Accuracy: 0.991400 |
| Step: 2100, Loss: 361.571228, Accuracy: 0.987600  Step: 2200, Loss: 330.185303, Accuracy: 0.989500  Step: 2300, Loss: 315.612518, Accuracy: 0.989700  ...  Step: 3800, Loss: 290.584351, Accuracy: 0.990500  Step: 3900, Loss: 225.680328, Accuracy: 0.993400  **Step: 4000, Loss: 245.907135, Accuracy: 0.993100** |

1. 결론



[개선 전 다층 신경망]



[개선 후 다층 신경망]

은닉층이 늘어날수록 정확도가 생각(처음에는 0.2%는 올라가지 않을까 생각했다.)보다 많이 개선되지 않았는데, 2000 번이라는 짧은 반복 횟수로는 파악하기 힘들었을 것이라 생각된다. 목표는 5000회 미만으로 정확도를 99% 만드는 것이었고, 실제로 2000번째일 때 99%로 진입하기도 하였다.

오차를 줄이는데 크게 기여한 부분은 Dropout의 배치 유무라고 판단을 했다. 그 이유는 똑같은 조건일 때 배치한 위치에 따라 정확도의 차이가 심했고, 있을 때보다 없을 때 더 나은 상황(전-결합층에서의 배치유무)이 생기기도 했기 때문이다. 전-결합층에서 Dropout 이 없는 것이 정확도가 더 높은 이유는 출력층에 가까운 층이기 때문이라고 생각을 한다. 실제로 특징 변수가 나온 레이어를 지나 출력층에 가까운 부분은 특징 변수에 따른 분류가 이루어지기 때문에, 노드를 비활성화해서 분류작업에 필요한 계산을 하지 않는 것이 오차를 줄이는데 도움이 되는지 의문이다. 실제로 Dropout이 없거나 Dropout의 rate이 1에 가까울 경우 정확도가 더 높게 나왔다.

추가적으로 전-결합층에서 노드의 개수에 따라 정확도를 측정해보았는데 512개와 1024개 사이에 평균값으로 노드 개수를 지정한 경우는 오히려 512개일 때보다 정확도가 낮았으며, loss도 높게 나왔다. 256개로 노드 개수를 지정했을 때는 512개와 1024개 사이에 평균값으로 노드 개수를 지정한 경우보다 정확도는 높았으나 512개일 때보다 정확도가 낮았다. 전-결합층에서 노드의 개수가 미치는 영향은 경우의 수가 많다보니 더 많은 실험을 하지 못했으나 500~700개 사이가 적절한 것이라 생각한다.