인공지능

HW2

2017803094 최지훈
소프트웨어학부
인공지능
박병준 교수님
월6, 수5 강의

●작성한 소스코드의 구동 환경

아나콘다의 버전은 4.7.12고 Python 3.7.4에서 작성하고 사용 IDE는 jupyter Notebook입니다. 또한 텐서플로우 버전은 1.14.0을 사용했습니다.

1. 다층 퍼셉트론 구현

●참조 모델 사이트와 관련 정보

참조 모델은 강의 교재의 p 683을 토대로 참고했습니다. 우선 CNN 모델과 동일하게 간단하게 데이터셋을 읽을 수 있는 MNIST를 사용하고, 데이터셋을 읽어올 경우 one_hot 인코딩 형태로 받아왔습니다. 그런 다음 다양한 파라미터들과 신경망 구조 관련 파라미터를 설정했습니다. 그리고 텐서 그래프 입력 변수를 넣었습니다. x는 필기체 영상이고, y는 그걸 토대로 숫자를 출력해 내는 것입니다.

층에서 가중치와 편차항을 사용하기 위해서 이것도 설정을 해주었습니다. tf.random_normal은 정규분포 형태의 난수 생성으로 랜덤한 값을 사용하기 위해 이를 사용했습니다.

여기서 구성한 신경망은 은닉층 한 개와 출력층으로 되어 있고, 은닉층은 15개의 노드를 갖도록 했습니다. 입력 데이터의 크기가 28 x 28이므로, 입력층에서는 784개의 노드를 만들었습니다. 출력에선 0부터 9까지 각 숫자에 대한확률을 출력하도록 10개의 노드를 만들었습니다.

multilayer_perceptron 함수를 사용해서 다층 퍼셉트론을 구성하는 은닉층은 reLu를 사용했습니다. 그리고 출력층에선 활성화 함수를 사용하지 않았습니다.

그런 다음 텐서보드로 보이기 위해 name_scope를 지정해줬습니다.
cost의 경우 오차를 정의하기 위해 출력층에서 분류 확률을 알아낼 수 있는 softmax 연산을 사용했습니다. 최적화 방법은 우선 AdamOptimizer을 사용했습니다.

정확도는 correct_prediction의 평균으로 설정했습니다.

다층 퍼셉트론은 역전파 알고리즘을 사용합니다. 그리고 전체적으로 코드에 주석을 달아놨습니다.

●작성한 코드에 대한 설명

```
import tensorflow as tf
import os
import matplotlib.pyplot as plt
```

```
# MNIST 데이터 적재
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

os.environ['TF_CPP_MIN_LOG_LEVEL']='2' # 경고 메시지 화면출력 금지

#파라미터
learning_rate = 0.001 #신경망 학습률
training_epochs = 100 # 학습 횟수 (epoch)
batch_size = 100 # 미니배치의 크기
display_step = 10 # 중간결과 출력 간격

#신경망 구조 관련 파라미터
n_hidden_1 = 15 # 은닉층의 노드 개수
n_input = 784 #입력층의 노드 개수 MNIST 데이터 (28x28)
n_classes = 10 # 출력층의 노드 수 MNIST 부류 개수(숫자 0~9)

#텐서 그래프 입력 변수
x = tf.placeholder("float", [None, n_input]) #입력 : 필기체 영상
y = tf.placeholder("float", [None, n_classes]) #출력 : 숫자

#학습모델 MLP정의
def multilayer_perceptron(x, weights, biases):
    #ReLu를 사용하는 은닉층
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    #출력층 (활성화 함수 미사용)
    out_layer = tf.matmul(layer_1, weights['out']) + biases['out']
    return out_layer

#학습할 파라미터: 가중치(weights), 편차항(biases)
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'out': tf.Variable(tf.random_normal([n_hidden_1, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

#신경망 모델 구성, 출력값 pred : 입력 x에 대한 신경망의 출력
```

```
pred = multilayer_perceptron(x, weights, biases)

with tf.name_scope("cost"):
    #비용(오차) 정의 (신경망 출력 pred, 목표 출력 y): 교차 엔트로피 사용
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y))
    #학습 알고리즘 설정
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
    tf.summary.scalar('cost', cost)

with tf.name_scope("accuracy"):
    #모델 테스트 : out의 최대값 노드와 y 노드가 같으면 정답
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
    #correct_prediction 평균
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    tf.summary.scalar('accuracy', accuracy)


init = tf.global_variables_initializer() #변수 초기화 지정

#데이터 플로우 그래프 실행
with tf.Session() as sess:
    print("Start....")
    writer = tf.summary.FileWriter("./logs/nn_logs", sess.graph)  # for 0.8
    merged = tf.summary.merge_all()

    sess.run(init)
    summary, acc = sess.run([merged, accuracy], feed_dict={x: mnist.test.images, y:
mnist.test.labels})
    total_batch = int(mnist.train.num_examples / batch_size) #배치 개수

    for epoch in range(training_epochs): #정해진 횟수 만큼 학습
        avg_cost = 0.
        for i in range(total_batch): #미니 배치
            batch_x, batch_y = mnist.train.next_batch(batch_size) #적재
            #역전파 알고리즘 적용
            _, c = sess.run([optimizer, cost], feed_dict={x: batch_x, y: batch_y})
            avg_cost += c / total_batch #평균 손실(오류) 계산

            if epoch %display_step == 0: #현재 학습 상황 출력
                print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost))
            summary, acc = sess.run([merged, accuracy], feed_dict={x: mnist.test.images,
```

```
y: mnist.test.labels})
                writer.add_summary(summary, i)

    print(accuracy.eval({x: mnist.test.images, y: mnist.test.labels}))
    print("End...")
```

참조 모델을 토대로 구현한 모델은 학습 알고리즘과 활성함수를 바꿔보았습니다. 활성함수는 강의자료에 나와있는 것처럼 sigmoid 함수를 사용, 그리고 학습 알고리즘은 AdamOptimizer에서 GradientDescentOptimizer을 사용했습니다.

```
layer_1 = tf.nn.sigmoid(layer_1)
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(cost)
```

이런 식으로 사용했습니다.

●각 실험 결과 로그 캡쳐 및 텐서보드 출력물

우선 참조 모델의 로그 결과를 보인다음, 텐서보드도 보이겠습니다.
각 학습 결과 타이밍의 시작부분과 끝부분을 캡쳐했습니다.

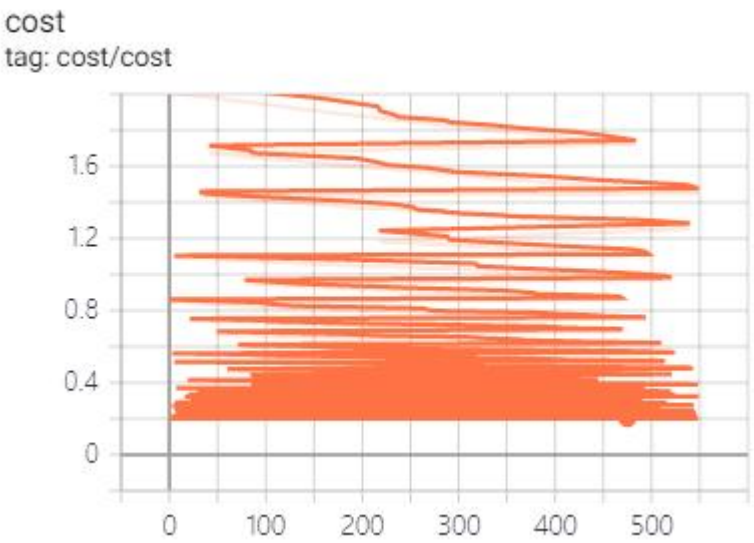
```
Start....
Epoch: 0001 cost= 0.058835699
Epoch: 0001 cost= 0.114240074
Epoch: 0001 cost= 0.161470802
Epoch: 0001 cost= 0.214088780
Epoch: 0001 cost= 0.259045105
Epoch: 0001 cost= 0.307635144
Epoch: 0001 cost= 0.358324512
Epoch: 0001 cost= 0.405354573
Epoch: 0001 cost= 0.448498816
Epoch: 0001 cost= 0.496896879
Epoch: 0001 cost= 0.544603927
Epoch: 0001 cost= 0.589191551
Epoch: 0001 cost= 0.629140438
Epoch: 0001 cost= 0.668647007
Epoch: 0001 cost= 7.222032920
Epoch: 0001 cost= 7.226735962
Epoch: 0001 cost= 7.232063919
Epoch: 0001 cost= 7.236450663
Epoch: 0001 cost= 7.241236590
Epoch: 0001 cost= 7.246476212
Epoch: 0011 cost= 0.000935562
Epoch: 0011 cost= 0.002313721
Epoch: 0011 cost= 0.003559541
Epoch: 0011 cost= 0.004682820
Epoch: 0011 cost= 0.006194666
Epoch: 0011 cost= 0.007953648
Epoch: 0011 cost= 0.009337784
Epoch: 0011 cost= 0.010406632
```

```
Epoch: 0011 cost= 0.638630305
Epoch: 0011 cost= 0.639623820
Epoch: 0011 cost= 0.640469240
Epoch: 0011 cost= 0.641645184
Epoch: 0011 cost= 0.642803477
Epoch: 0011 cost= 0.644236350
Epoch: 0011 cost= 0.645184465
Epoch: 0021 cost= 0.000461647
Epoch: 0021 cost= 0.001223990
Epoch: 0021 cost= 0.002016472
Epoch: 0021 cost= 0.002478607
Epoch: 0021 cost= 0.003323040
Epoch: 0021 cost= 0.003953900
Epoch: 0021 cost= 0.004846496
Epoch: 0021 cost= 0.318140227
Epoch: 0021 cost= 0.318621773
Epoch: 0021 cost= 0.319148783
Epoch: 0021 cost= 0.319547150
Epoch: 0021 cost= 0.320115891
Epoch: 0031 cost= 0.000435498
Epoch: 0031 cost= 0.001107376
Epoch: 0031 cost= 0.001305284
Epoch: 0031 cost= 0.227258929
Epoch: 0031 cost= 0.227819896
Epoch: 0031 cost= 0.228264797
Epoch: 0031 cost= 0.228611150
Epoch: 0041 cost= 0.000461288
Epoch: 0041 cost= 0.000709497
Epoch: 0041 cost= 0.001346529
Epoch: 0041 cost= 0.001649535
Epoch: 0041 cost= 0.002010940
Epoch: 0041 cost= 0.188527769
Epoch: 0041 cost= 0.188764045
Epoch: 0041 cost= 0.189116786
Epoch: 0041 cost= 0.189559489
Epoch: 0051 cost= 0.000281290
Epoch: 0051 cost= 0.000468306
Epoch: 0051 cost= 0.000651797
Epoch: 0051 cost= 0.000811495
Epoch: 0051 cost= 0.001214397
Epoch: 0051 cost= 0.001565413
```

```
Epoch: 0051 cost= 0.165925102
Epoch: 0051 cost= 0.166048699
Epoch: 0051 cost= 0.166202191
Epoch: 0051 cost= 0.166589960
Epoch: 0051 cost= 0.167002475
Epoch: 0051 cost= 0.167200045
Epoch: 0061 cost= 0.000115143
Epoch: 0061 cost= 0.000583391
Epoch: 0061 cost= 0.000669749
Epoch: 0061 cost= 0.000886318
Epoch: 0061 cost= 0.001033016
.
Epoch: 0061 cost= 0.151773990
Epoch: 0061 cost= 0.151988857
Epoch: 0061 cost= 0.152206037
Epoch: 0061 cost= 0.152421121
Epoch: 0061 cost= 0.152586607
Epoch: 0071 cost= 0.000238832
Epoch: 0071 cost= 0.000540543
Epoch: 0071 cost= 0.000999734
Epoch: 0071 cost= 0.001444490
Epoch: 0071 cost= 0.001761064
Epoch: 0071 cost= 0.001995028
Epoch: 0071 cost= 0.002275618
.
Epoch: 0081 cost= 0.133390883
Epoch: 0081 cost= 0.133642322
Epoch: 0081 cost= 0.133816949
Epoch: 0081 cost= 0.134182525
Epoch: 0091 cost= 0.000074432
Epoch: 0091 cost= 0.000254853
Epoch: 0091 cost= 0.000644927
Epoch: 0091 cost= 0.000774235
Epoch: 0091 cost= 0.000950937
Epoch: 0091 cost= 0.001291099
Epoch: 0091 cost= 0.001702032
Epoch: 0091 cost= 0.126259869
Epoch: 0091 cost= 0.126529526
Epoch: 0091 cost= 0.126640293
Epoch: 0091 cost= 0.126720919
Epoch: 0091 cost= 0.127038731
Epoch: 0091 cost= 0.127168008
Epoch: 0091 cost= 0.127364605
0.947
End...
```

끝나는 지점에서 점점 학습 오차를 나타내는 교차 엔트로피 값이 점점 감소함을 알 수 있었습니다.

참조모델의 정확도와 cost 텐서보드입니다.



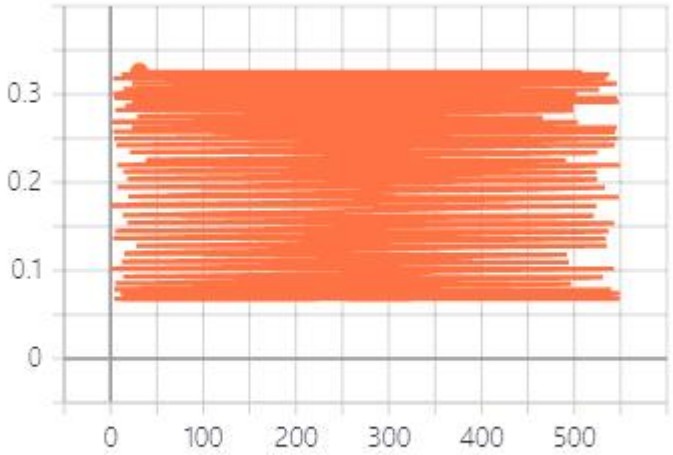
다음은 활성화 함수와 최적화 방법을 바꾼 생성 모델입니다.

```
Start....
Epoch: 0001 cost= 0.009559446
Epoch: 0001 cost= 0.018878047
Epoch: 0001 cost= 0.028235471
Epoch: 0001 cost= 0.037671442
Epoch: 0001 cost= 0.047124451
Epoch: 0001 cost= 0.056059762
Epoch: 0001 cost= 0.065368387
Epoch: 0001 cost= 0.074798034
Epoch: 0001 cost= 0.083584023
Epoch: 0001 cost= 4.549667887
Epoch: 0001 cost= 4.557238713
Epoch: 0001 cost= 4.564746298
Epoch: 0001 cost= 4.571455959
Epoch: 0001 cost= 4.579128081
Epoch: 0011 cost= 0.004833585
Epoch: 0011 cost= 0.009284745
Epoch: 0011 cost= 0.014567957
Epoch: 0011 cost= 0.019650982
Epoch: 0011 cost= 0.024693623
Epoch: 0011 cost= 2.740193519
Epoch: 0011 cost= 2.744882813
Epoch: 0011 cost= 2.749348821
Epoch: 0021 cost= 0.004174894
Epoch: 0021 cost= 0.008085996
Epoch: 0021 cost= 0.012157204
Epoch: 0021 cost= 0.016330204
Epoch: 0021 cost= 0.020936840
Epoch: 0021 cost= 2.364247455
Epoch: 0021 cost= 2.368719824
Epoch: 0021 cost= 2.372665513
Epoch: 0021 cost= 2.377227843
Epoch: 0021 cost= 2.381967063
Epoch: 0021 cost= 2.386134885
Epoch: 0031 cost= 0.004075339
Epoch: 0031 cost= 0.008021149
Epoch: 0031 cost= 0.012222456
Epoch: 0031 cost= 0.016076639
Epoch: 0031 cost= 0.020001709
Epoch: 0031 cost= 0.024204875
Epoch: 0031 cost= 0.028276685
Epoch: 0031 cost= 0.032360741
```

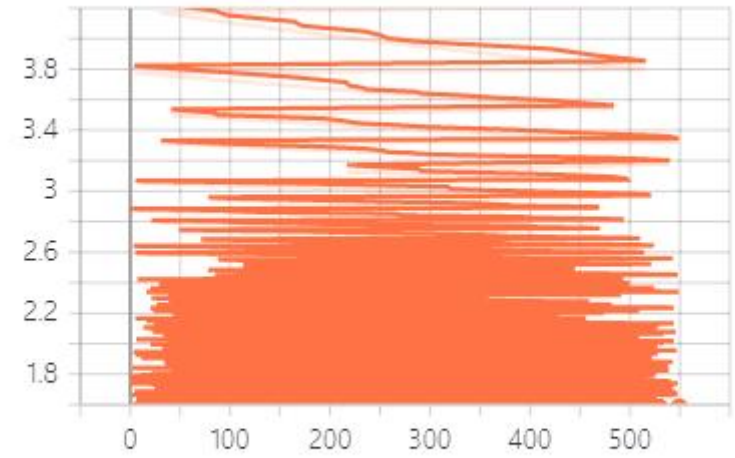
```
Epoch: 0031 cost= 2.164812308
Epoch: 0031 cost= 2.168634059
Epoch: 0031 cost= 2.172424131
Epoch: 0031 cost= 2.176346962
Epoch: 0031 cost= 2.180590295
Epoch: 0031 cost= 2.184368992
Epoch: 0041 cost= 0.003784911
Epoch: 0041 cost= 0.007433416
Epoch: 0041 cost= 0.011320375
Epoch: 0041 cost= 0.015051938
Epoch: 0041 cost= 0.018524432
Epoch: 0041 cost= 0.022513543
Epoch: 0041 cost= 2.028707149
Epoch: 0041 cost= 2.032811989
Epoch: 0041 cost= 2.036410289
Epoch: 0041 cost= 2.040058030
Epoch: 0041 cost= 2.043956218
Epoch: 0041 cost= 2.047571427
Epoch: 0051 cost= 0.003529891
Epoch: 0051 cost= 0.007144978
Epoch: 0051 cost= 0.010395612
Epoch: 0051 cost= 0.014229906
Epoch: 0051 cost= 0.017827901
Epoch: 0051 cost= 0.021365425
Epoch: 0051 cost= 1.923015839
Epoch: 0051 cost= 1.926227942
Epoch: 0051 cost= 1.929814617
Epoch: 0051 cost= 1.933251748
Epoch: 0051 cost= 1.936725058
Epoch: 0051 cost= 1.939989491
Epoch: 0051 cost= 1.943297082

Epoch: 0061 cost= 0.003221685
Epoch: 0061 cost= 0.006618255
Epoch: 0061 cost= 0.010126029
Epoch: 0061 cost= 0.013387424
Epoch: 0061 cost= 0.016401500
Epoch: 0061 cost= 0.020055304
```

```
Epoch: 0061 cost= 1.844223407
Epoch: 0061 cost= 1.847419130
Epoch: 0061 cost= 1.850957154
Epoch: 0061 cost= 1.854440926
Epoch: 0061 cost= 1.857471080
Epoch: 0071 cost= 0.002927410
Epoch: 0071 cost= 0.006120526
Epoch: 0071 cost= 0.009413620
Epoch: 0071 cost= 0.012353247
Epoch: 0071 cost= 0.015665723
Epoch: 0071 cost= 0.019067450
Epoch: 0071 cost= 1.763411345
Epoch: 0071 cost= 1.766684225
Epoch: 0071 cost= 1.769817360
Epoch: 0071 cost= 1.773096155
Epoch: 0071 cost= 1.776416908
Epoch: 0071 cost= 1.779721102
Epoch: 0071 cost= 1.783276676
Epoch: 0081 cost= 0.003075566
Epoch: 0081 cost= 0.006325541
Epoch: 0081 cost= 0.009505602
Epoch: 0081 cost= 0.012689779
Epoch: 0081 cost= 0.015923423
Epoch: 0081 cost= 0.019110657
Epoch: 0081 cost= 0.022373482
Epoch: 0091 cost= 1.627068066
Epoch: 0091 cost= 1.629628420
Epoch: 0091 cost= 1.632517717
Epoch: 0091 cost= 1.635648953
Epoch: 0091 cost= 1.638506858
Epoch: 0091 cost= 1.641305820
Epoch: 0091 cost= 1.644440935
Epoch: 0091 cost= 1.647629477
Epoch: 0091 cost= 1.650456980
Epoch: 0091 cost= 1.653559548
Epoch: 0091 cost= 1.656510924
0.4884
End...
```

accuracy
tag: accuracy/accuracy



cost
tag: cost/cost



●실험 결과 분석 및 고찰

참조 모델과 생성 모델 둘 다 학습을 시킬수록 학습의 오차가 줄어들고 있음은 볼 수 있었습니다. 하지만 참조 모델에 비해 생성 모델의 학습 오차가 압도적으로 컸고, 정확도 또한 0.94에서 0.48로 거의 절반 가량 줄어들었습니다. 이러한 이유가 생긴 이유는 활성화 함수와 최적화 기법이 좋지 않았기 때문입니다. sigmoid 함수보다는 relu 함수가 더 뛰어남을 알 수 있었고, GradientDescentOptimizer보다는 AdamOptimizer이 좋음도 알 수 있었습니다. 시간이 오래 걸려서 learning_rate라든지 학습에 필요한 파라미터 등은 고쳐보지 않았는데 기회가 된다면 바꿔보도록 하겠습니다. 하지만 얼추 예상이 가는 부분이 있는데, learning_rate를 증가시킨다면 성능이 더 좋아질 거 같다든지 하는 방향이 있습니다.

2. CNN 구현

●참조 모델 사이트와 관련 정보

https://coderkoo.tistory.com/13 를 참조했습니다. CNN의 내용에 초점을 맞추기 위해 가장 간단하게 데이터셋을 읽을 수 있는 MNIST를 사용하고, 데이터셋을 읽어올 경우 one_hot 인코딩 형태로, 이미지 모양을 그대로 받아오기 위해 reshape = false로 설정했습니다.
CNN 모델은

- **Conv-ReLU-Pool-Conv-ReLU-Pool-Conv-ReLU-Pool-FC-SM**

이 자료는 강의자료에 있는 내용입니다. 참조 모델 사이트에서의 모델은 여기서 층 하나만 없이 Conv-ReLU-Pool이 두 번 반복되는 구조를 가지고 있습니다. 즉, 2 개의 Convolution Layer을 거쳐서 Fully Connected Layer로 만든 뒤에 결과를 내는 것입니다.
28x28x1 이미지에서 1개의 Convolution Layer을 거쳐서 14x14x4를 만들고, 거기서 또 다음 Convolution Layer을 거쳐서 7x7x8을 만들고 그 다음에 10개를 뽑아내는 것입니다.

처음 Kernel, 즉 필터는 4x4x1 필터를 4장 사용하기 위해 shape = [4,4,1,4]로 주고, tf.truncated_normal()을 통해서 초기화 하였습니다. 그 다음 Kernel을 Conv한 뒤에 같은 사이즈만큼 더하기 위한 변수를 만들고, 4장의 Kernel을 사용하기 때문에 shape=[4]로 설정했습니다.
Convolution 연산을 했습니다. Stride는 1,1,1,1로 설정했기 때문에 여기서 사용하는 Stride는 1칸씩 움직입니다. Conv1 변수에 활성함수를 ReLU를 사용했습니다. 그 다음에 풀링의 경우 최대값을 구하는 max_pool 방식을 사용했습니다.

두 번째 Kernel은 4x4x4 필터를 8장 사용하기 위해서 shape = [4,4,4,8]을 주고, 초기화를 하였습니다. 그 다음 변수에는 8장의 Kernel을 사용하기 위해 shape=[8]을 사용했습니다. Convolution 연산과, relu 활성함수, max pooling 방식은 위와 동일하기 때문에 설명을 생략하겠습니다.

두 번의 Convolution layer을 거치고 나서 7x7x8의 결과가 나옵니다. Pool2_flat을 보면 7x7x8의 결과물을 1차로 펼치는 것입니다. 앞의 –1은 배치사이즈입니다. 그리고 8*7*7에 대하여 행렬의 곱을 해야 하기에 W1의 크기는 8*7*7 x 10입니다.

텐서보드에서 그래프를 보기 위해서 이름을 설정해주고 loss와 accuracy를 구해줬습니다. 그리고 log 파일로 기록해야 텐서보드에서 볼 수 있기 때문에 파일도 하나 만들었습니다.
그리고 목적 함수는 AdamOptimizer을 사용했습니다.

기존 참조 모델에서는 for문을 10000번씩 돌았습니다. 이렇게 돌린 다면 조금 더 정확해질 수 있으나 과제를 하는 시간을 절약하기 위해서 저는 200번으로 돌렸습니다.

●작성한 코드에 대한 설명

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import timeit
```

걸리는 시간을 알기 위해 넣었습니다.

```
start = timeit.default_timer()
mnist = input_data.read_data_sets('MNIST_data', one_hot=True, reshape=False)

X = tf.placeholder(tf.float32, shape=[None, 28, 28, 1])
Y_Label = tf.placeholder(tf.float32, shape=[None, 10])
```

데이터플로우 그래프 실행 시 텐서를 전달하기 위해 사용됩니다. 자료형은 32비트 float을 사용합니다.

X는 입력, Y_Label은 출력을 의미합니다.

```
Kernel1 = tf.Variable(tf.truncated_normal(shape=[4, 4, 1, 4], stddev=0.1))
Bias1 = tf.Variable(tf.truncated_normal(shape=[4], stddev=0.1))
Conv1 = tf.nn.conv2d(X, Kernel1, strides=[1, 1, 1, 1], padding='SAME') + Bias1
Activation1 = tf.nn.relu(Conv1)
Pool1 = tf.nn.max_pool(Activation1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

Kernel2 = tf.Variable(tf.truncated_normal(shape=[4, 4, 4, 8], stddev=0.1))
Bias2 = tf.Variable(tf.truncated_normal(shape=[8], stddev=0.1))
Conv2 = tf.nn.conv2d(Pool1, Kernel2, strides=[1, 1, 1, 1], padding='SAME') + Bias2
Activation2 = tf.nn.relu(Conv2)
Pool2 = tf.nn.max_pool(Activation2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

W1 = tf.Variable(tf.truncated_normal(shape=[8 * 7 * 7, 10])) #8x7x7
B1 = tf.Variable(tf.truncated_normal(shape=[10]))
```

모델의 학습 가능한 변수를 정의할 때 Variable을 사용합니다. 가중치, 편향 등의 파라미터를 저장하는 데 사용됩니다. 이는 정의할 때 반드시 초기화가 되어야 합니다.

```
Pool2_flat = tf.reshape(Pool2, [-1, 8 * 7 * 7])
OutputLayer = tf.matmul(Pool2_flat, W1) + B1
```

reshape로 Pool2가 가지고 있는 텐서의 원소를 유지하면서 모양을 바꾸는 것입니다.

```
with tf.name_scope("loss"):
    Loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y_Label, logits=OutputLayer))
```

readuce_mean은 지정된 축의 원소들 중 평균값을 의미합ㄴ디ㅏ.

```
    train_step = tf.train.AdamOptimizer(0.005).minimize(Loss)
    tf.summary.scalar('loss', Loss)

with tf.name_scope("accuracy"):
    correct_prediction = tf.equal(tf.argmax(OutputLayer, 1), tf.argmax(Y_Label, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    tf.summary.scalar('accuracy', accuracy)

with tf.Session() as sess:
    print("Start....")
    writer = tf.summary.FileWriter("./logs/nn_logs", sess.graph)  # for 0.8
```
프로그램 실행 중 해당 정보를 로그 파일에 저장하려고 실행하는 명령어입니다.

```
    merged = tf.summary.merge_all()

    sess.run(tf.global_variables_initializer())
    summary, acc = sess.run([merged, accuracy], feed_dict={X: mnist.test.images, Y_Label: mnist.test.labels})
    for i in range(201):
        trainingData, Y = mnist.train.next_batch(64)
        sess.run(train_step,feed_dict={X:trainingData,Y_Label:Y})
        if i%10 == 0 :
            print(sess.run(accuracy, feed_dict={X:mnist.test.images, Y_Label:mnist.test.labels}))
        summary, acc = sess.run([merged, accuracy], feed_dict={X: mnist.test.images, Y_Label: mnist.test.labels})
        writer.add_summary(summary, i)

    end = timeit.default_timer()
    print("End...")
    print("Run time : ", end – start)
```
기본적으로 참조 모델 코드를 바탕으로 하나씩 변한 부분을 하나씩 추가하겠습니다.
우선 참조 모델 코드의 경우 위에서 전체적으로 설명했으므로 넘어가겠습니다.

첫 번째 구현한 모델은 참조 모델에서 Convolution layer의 활성 함수를 sigmoid 함수를 사용한 것입니다. 성능이 얼마나 악화되는지 궁금해서 사용해봤습니다.
```
Activation1 = tf.nn.sigmoid(Conv1)  Activation2 = tf.nn.sigmoid(Conv2)
```
이런 방법을 사용할 수 있습니다. 혹은 쌍곡 탄젠트 함수인 tf.nn.tanh를 사용해도 비슷한 결과를 얻었을 것 같습니다.

두 번째 구현한 모델은 첫 번째 모델에서 악화된 것을 봤으니 다시 활성 함수 ReLU를 사용

하고 더 좋은 결과를 만들어 내기 위해 같은 유형의 Convolution Layer을 하나 더 추가한 것입니다.

```
Activation1 = tf.nn.relu(Conv1) Activation2 = tf.nn.relu(Conv2)
Kernel3 = tf.Variable(tf.truncated_normal(shape=[4, 4, 8, 16], stddev=0.1)) # 레이어 한개 더 추가 # 4x4x8 16장
Bias3 = tf.Variable(tf.truncated_normal(shape=[16], stddev=0.1)) # 16장
Conv3 = tf.nn.conv2d(Pool2, Kernel3, strides=[1, 1, 1, 1], padding='SAME') + Bias3
Activation3 = tf.nn.relu(Conv3)
Pool3 = tf.nn.max_pool(Activation3, ksize=[1, 1, 1, 1], strides=[1, 1, 1, 1], padding='SAME')
```

세 번째 구현한 모델은 두 번째 모델에서 풀링 방식과
목적 함수를 GradientDescentOptimizer로 바꾼 것입니다.

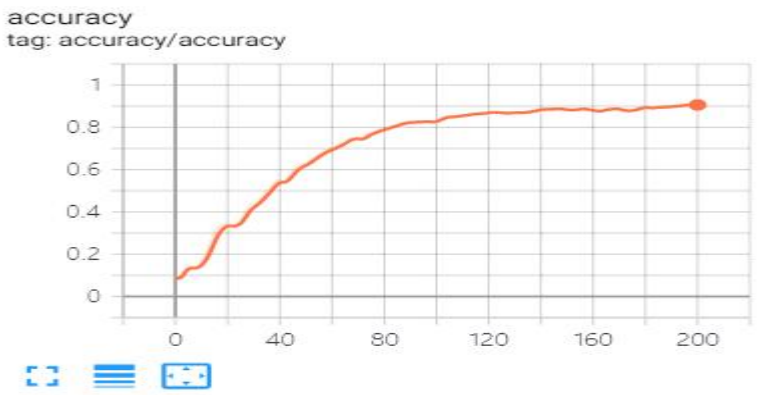
```
Pool1 = tf.nn.avg_pool(Activation1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
Pool2 = tf.nn.avg_pool(Activation2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
Pool3 = tf.nn.avg_pool(Activation3, ksize=[1, 1, 1, 1], strides=[1, 1, 1, 1], padding='SAME')
train_step = tf.train.GradientDescentOptimizer(0.005).minimize(Loss)
```

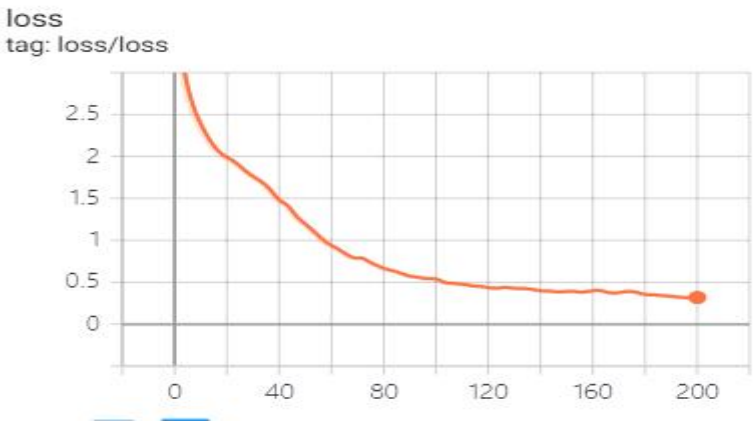
●각 실험 결과 로그 캡쳐 및 텐서보드 출력물

Start와 End 사이는 정확도입니다. 그림 순으로 jupyter notebook에서의 출력물, 텐서 보드에서의 accuracy, loss 순으로 되어 있습니다.

참조 모델입니다.

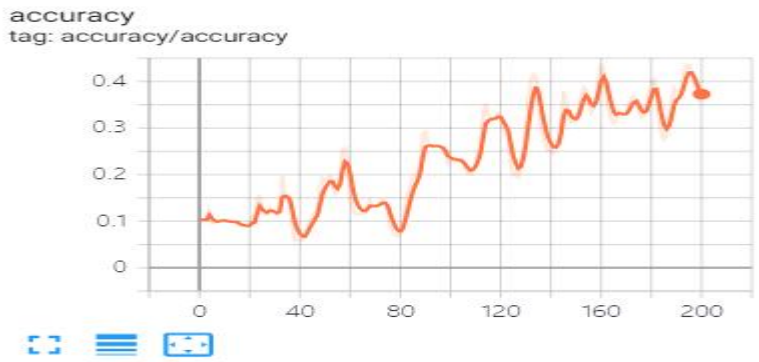
```
Start....
0.0883
0.1604
0.3414
0.4358
0.5498
0.6291
0.7005
0.7472
0.7935
0.8258
0.8298
0.8566
0.8719
0.8698
0.8858
0.8817
0.8774
0.8846
0.895
0.8995
0.906
End...
Run Time :   185.18174689999998
```

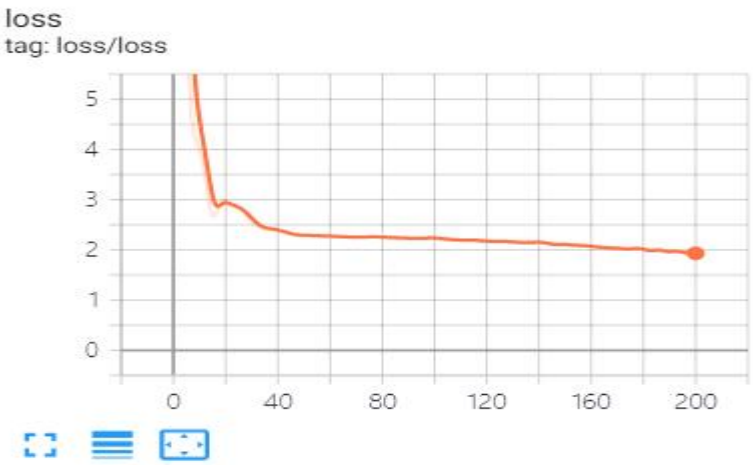




활성 함수를 sigmoid로 변환한 첫 번째 모델입니다.

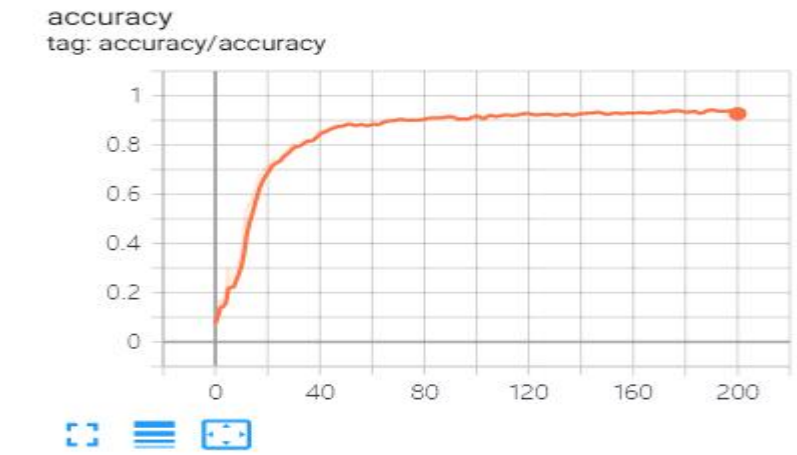
```
Start....
0.1032
0.1028
0.0892
0.1166
0.0591
0.1883
0.1732
0.132
0.0744
0.2917
0.2315
0.2266
0.3232
0.3063
0.2497
0.3175
0.4373
0.3312
0.3922
0.3784
0.3599
End...
Run time :  182.04388609999995
```

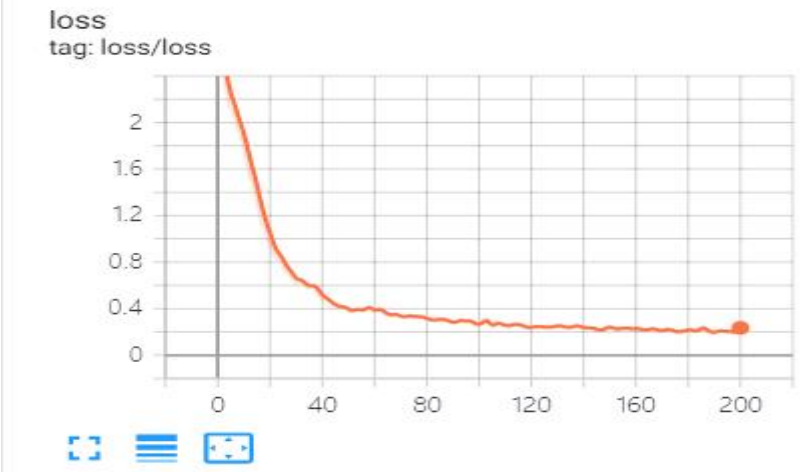




활성 함수를 다시 relu로 바꾸고 커널 층을 한 개 더 늘린 두 번째 모델입니다.

```
Start....
0.074
0.3512
0.7076
0.8025
0.8561
0.8874
0.8872
0.9058
0.9063
0.9161
0.9207
0.922
0.9264
0.919
0.9297
0.9211
0.9292
0.9375
0.9296
0.9441
0.917
End...
Run time :  210.7734011
```





두 번째 모델에서 drop_out을 추가 해봤습니다. 이것은 새로운 모델이 아니라 그냥 drop_out 추가할 시 어떻게 되는지 궁금해서 해봤습니다.

```
Start....
0.0839
0.1132
0.1345
0.1721
0.2245
0.2994
0.4007
0.4917
0.5724
0.6465
0.67
0.705
0.7336
0.7491
0.7642
0.7599
0.7909
0.7995
0.7948
0.8057
0.8092
End...
Run time :  277.3146697
```

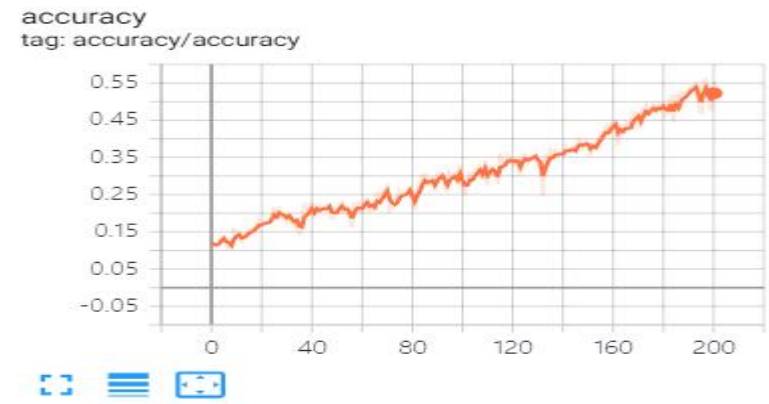
두 번째 모델에서 우선 풀링 방식만 max_pool에서 avg_pool로 바꿔 보았습니다.

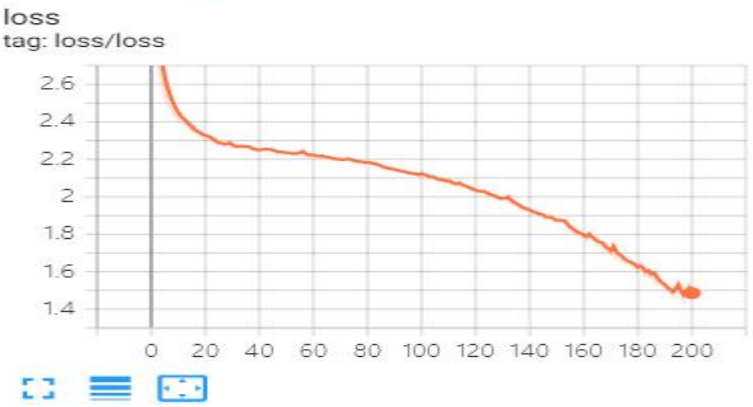
```
Start....
0.0613
0.1453
0.2531
0.4815
0.6069
0.6873
0.7617
0.823
0.8336
0.8689
0.872
0.8975
0.9066
0.9203
0.9165
0.9125
0.9227
0.9237
0.9275
0.926
0.9395
End...
Run time :  214.7755042
```

두 번째 모델에 비해서는 악화되기는 했지만 그래도 꽤 준수한 모델이라 생각되어 더욱 악화된 모델을 보고 싶어 목적 함수를 바꾼 세 번째 모델을 만들었습니다.

두 번째 모델에서 풀링 방식을 avg_pool로 바꾸고 목적 함수를 GradientDescentOptimizer로 바꾼 마지막 모델입니다.

```
Start....
0.1206
0.1505
0.1699
0.1775
0.2298
0.2032
0.2084
0.2747
0.2266
0.3035
0.2622
0.2745
0.3324
0.3287
0.3579
0.3855
0.4454
0.4699
0.4982
0.5319
0.5522
End...
Run time :  216.13924599999999
```





●실험 결과 분석 및 고찰

참조 모델은 꽤 괜찮은 수준의 성능을 보여주고 있습니다. for문을 10000정도로 돌렸다면 보다 괜찮은 성능을 보여줬을 것 같습니다.
활성 함수는 강의 시간에 배운 바와 마찬가지로 시그모이드 함수나 쌍곡탄젠트 함수를 사용하기 보다는 relu 함수를 사용하는 것이 낫다는 것을 알았습니다. 시그모이드 함수와 쌍곡탄젠트 함수는 기존 신경망의 활성화 함수인데 둘 다 도함수가 0에서 멀어질 때 거의 0으로 수렴하기에 이는 그레디언트의 성분이 사라지므로 이들 둘으로는 학습을 할 수가 없다는 것을 알았습니다.

또한 목적 함수도 GradientDescentOptimizer을 사용하기보다 AdamOptimizer을 사용하는 것이 낫다는 것도 알았습니다.

커널의 층을 추가한다면 성능이 더 좋아짐도 알았습니다.

두 번째 모델에서
Pool1 = tf.nn.dropout(Pool1,keep_prob=0.8) 이런 식으로 dropout도 넣어 봤으나 정확도 결과는 오히려 떨어짐을 파악할 수 있었습니다.

풀링 방식 또한 avg_pool보다는 max_pool이 나음을 깨달았습니다.
max pooling이 더 나은 이유는 average pooling이나 확률적 풀링인 stochastic pooling의 경우 이상적이지 않은 값을 선택할 수 있기 때문입니다.