AI assignment 3 report

2014121136 구한모

1) Framework : 작업은 tensorflow 를 이용하였으며, 노트북으로 작업한 관계로 구글 colab에서 작업하였음.

2) 코드내용 및 설명. (숫자 : code line) :

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

import tensorflow as tf

# Parameters
learning_rate = 0.01
training_epochs = 20
batch_size = 100
display_step = 1

X = tf.placeholder("float", [None, 784])

Y = tf.placeholder("float", [None, 10])

weights = {
    'h_conv1': tf.Variable(tf.random_normal([5, 5, 1, 4],mean=0, stddev=0.1)),
    'h_conv2': tf.Variable(tf.random_normal([5, 5, 4, 12], mean=0, stddev=0.1)),
    'h_fc': tf.Variable(tf.random_normal([4*4*12, 10]))
}

biases = {
    'b_conv1': tf.Variable(tf.random_normal([4], mean =0, stddev=0.1)),
    'b_conv2': tf.Variable(tf.random_normal([12], mean=0, stddev=0.1)),
    'b_fc': tf.Variable(tf.random_normal([10]))
}

def lenet_mnist(x):
    x_image = tf.reshape(x, [-1, 28, 28, 1])
    conv1 = tf.nn.relu(tf.nn.conv2d(x_image, weights['h_conv1'], strides =[1,1,1,1], padding="VALID")
                       + biases['b_conv1'])
    pool1 = tf.nn.max_pool(conv1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')

    conv2 = tf.nn.relu(tf.nn.conv2d(pool1, weights['h_conv2'], strides=[1,1,1,1], padding="VALID")
                       +biases['b_conv2'])
    pool2 = tf.nn.max_pool(conv2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
    pool2_flat = tf.reshape(pool2, [-1,4*4*12])
```

```
    fc = tf.matmul(pool2_flat, weights['h_fc']) + biases['b_fc']
    return fc

logits = lenet_mnist(X)

loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=Y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            _, c = sess.run([train_op, loss_op], feed_dict={X: batch_x, Y: batch_y})
            avg_cost += c / total_batch
        if epoch % display_step == 0:
            print("Train Set: Epoch", '%d' % (epoch+1), "Average Loss: {:.6f}".format(avg_cost),
                  "lr:{:.2e}".format(learning_rate))

    pred = tf.nn.softmax(logits)
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(Y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    ac, averageLoss = sess.run([accuracy, loss_op], feed_dict={X: mnist.test.images, Y:mnist.test.labels})
    print("Test Set: Average Loss : {:.6f}".format(averageLoss), "Accuracy:", "%d/10000"%(ac*10000),
          "({:.2f}%)".format(ac*100))
```

2,3 : tensorflow의 튜토리얼 내의 mnist데이터를 가져와 "mnist"변수에 저장

5 : tensorflow를 import하여 후에 tf관련 함수 들을 쓸 수 있게 함.

8 : Gradient Descent에 쓰일 learning rate을 0.01으로 설정

9 : training epoch의 수를 20번으로 설정, 즉 data example set을 20번 돌릴 예정.

10 : batch size를 100으로 설정. data를 하나 하나 넣을 때 마다 학습하는 것이 아닌, 학습하는 data 묶음의 단위의 크기.

11: display_step : 뒤의 60번 코드에서 쓰이지만, train process에서 average loss를 출력하는 epoch의 주기

13: 초기 mnist data의 크기인 28*28의 픽셀 값을 담을 수 있는 tensor변수 설정.

145 mnist의 실제 결과값인 lable을 담을 수 있는 10크기의 tensor변수 설정

17~21 : convolutional network 및, fully connected layer 연산에 쓰일 weight들의 tensor변수 설정. 처음에는 random_normal을 이용하여 임의로 설정해준다. 이때, 임의 값들의 평균은 기존대로 0으로 설정하지만, 표준편차의 값을 기존의 1이 아니라 0.1로 함으로써 gradient descent의 back prop연산 결과를 weight들에 상대적으로 크게 반영할 수 있도록 하여 더 빠른 학습이 가능하도록 하였다.

(18) 또한 h_conv1은 mnist의 image를 다시 28*28의 이미지로 reshape한 결과의 행렬을 받아 convolution을 진행하려 하며, 이때 필터의 크기는 5*5, greyscale인 만큼 channel은 1이고 LeNet 설계에 따라 이를 우선 4 channel의 feature로 mapping하려 한다. 따라서 shape을 [5, 5, 1, 4]로 설정하였다.

(19) h_conv2는 후에 입력된 MNIST 이미지가 h_conv1 처리되어 나온 값을 max pool하여 나온 feature(12*12*4)를 다시한번 처리하기 위한 convolution weight들이다. LeNet 설계에 따라 5*5필터들을 이용하여 나온 4개의 channel의 12*12 feature들을 다시 한번 convolution을 수행하여 12개의 feature로 mapping하기 위하여, shape은 [5, 5, 4, 12]로 설정한다.

(20) h_fc는 h_conv2 연산을 통해 나온 feature들(4*4*12, 여기서 4 = ((28-4)/2 – 4)/2 로 28*28짜리 image를 5*5필터를 이용한 conv와 반으로 줄이는 max_pool을 두 번 거친 결과이다.)로 0~9 중 하나로 선택하는 one-hot vector를 계산하는 fully-connected layer의 인 만큼, shape을 [4*4*12, 10]으로 설정한다.

23~27 : convolutional newtwork와 fully connected layer 연산에 쓰이는 bias 값들을 random_normal을 이용하여 임의로 설정해준다. 역시, 표준편차는 0.1로 적게 설정하였다. b_conv1과 b_conv2는 각각, h_conv1 convolution, H-conv2 결과에 더하고 난 뒤 ReLu연산이 들어가게 되는데, 이때 쓰이는 상수 값들이다. b_fc는 fully connected layer 이후에 더해지는 10개 크기의 상수 벡터이다.

위의 줄로 대략적인 LeNet의 global data들을 선언하였다.

29~48: lenet의 model을 설정해준다.

30: MNIST의 input image들을 28 * 28 * 1의 행렬의 형태로 다시 재배열한다.

31~32 : MNIST의 input image들을 h_conv1라는 5*5*4의 필터를 써서 stride 1로 convolve하여, 24*24*4의 feature의 결과로 mappng 한다.(tf.nn.conv2d) 이때, 24*24*4의 결과를 낼 수 있게 padding은 채우지 않고 "VALID"로 설정한다. Pooling은 각 feature들 의 구역 중에서 특징적인 feature들, 혹은 그것들을 대표하는 값만 모아서 간추릴 수 있게(down sampling) 하는 과정이고, 그 중 여기서 쓰이는 max pooling은 각 설정한(여기서는 2*2) 구역 중에서 가장 큰 feature의 값으로 간추리는 과정이다. convolution 결과의 feature들에 bias인 b_conv1을 더하고, ReLu를 취한다(tf.nn.relu). 이렇게 나온 feature를 conv1로 하자.

33: conv1의 가로 세로 사이즈를 반 씩으로(12*12*4) 줄이기 위해 max pooling을 실시한다. 반으로 줄이기 위하여 stride=[1,2,2,1], ksize=[1,2,2,1] 으로 설정하여 tf.nn.max_pool 함수에 conv1와 함께 인수로 전달하고 결과 값을 pool1에 담아준다.

35~37 : 31~32와 비슷하게, pool1의 결과물을 다시 5*5*12의 필터를 써서 stride 1로 convolve하여 8*8*12의 feature로 mapping한다. 역시 padding="VALID"이고 이 결과에 bias인 b_conv2를 더한 뒤 relu를 취한다. 이렇게 나온 결과를 다시 가로 세로 사이즈 반 씩으로 max pooling을 실시한다. 그렇게 해서 나온 4*4*12의 결과를 pool2에 저장한다.

38 : 4*4*12개의 feature들을 10개의 뉴런들과 fully connect할 수 있도록 다시 pool2의 feature들을 1차원의 벡터로 reshape하여 pool2_flat에 저장한다.

40 : pool2_flat에 저장한 뒤 h_fc의 weight들과 행렬 곱을 실시하고 상수항인 bias, b_fc를 더하여 최종 예측 벡터 fc를 반환받는다.

43: 이러한 lenet의 image X(후에 train에서 받을 때는 batch의 X값, 즉 MNIST의 트레이닝 이미지 값)에 대한 예측 값(labeling)을 logit에 담는다.

45: 예측 값(labeling)과 실제 Y값(실제 labeling, or group)사이의 cross entropy를 softmax 하여 cost, 혹은 loss로 잡는 부분이다.

46: 위에서 설정한 learning rate = 0.1로 하여 gradient descent algorithm을 통해 back prop을 수행할 수 있도록 optimizer를 만들어주는 부분이다. 즉 loss function이 감소하는 벡터 방향으로 feature들을 그 기울기값에 learning rate을 곱하여 이동하게 된다.

47. 위에서 설정한 loss 즉, cross entropy의 softmax를 줄이는 방식으로 optimize할 수 있도록 training 방식을 설정한다. 즉, 예측 결과의 weight들의 절댓값 크기 자체가 큰 것으로 인한 loss의 크기가 과대평가 혹은 과소평가 되지 않도록 softmax로 정규화를 시켜놓고 그 cost를 최소화 시키는 방식으로 optimize하게 된다.

48: 상단과 현 부분에서 설정한 전역변수들(ex. weights['h_conv1'] 은 임의의 값으로)을 초기화하는 부분이다..

50~73 : 실제 세션이 동작하는 부분이다.

51: 세션을 시작하면서 48줄에서 설정하였듯 전역변수들을 초기화시킨다.

53~63 : epoch의 개수만큼의 횟수로 data example 학습을 돌린다.

54 : 각 epoch별로 average cost를 계산하기 위해 average cost를 0으로 초기화.

55: 전체 data example을 batch 크기만큼 나눈 것을 total batch개수로 하여 batch단위로 학습을 total batch만큼 반복시킨다.

57~60: batch별 반복을 의미한다. batch_X 와 batch_Y에 batch별 X값과 Y값을 받은 뒤, sess.run을 통해 상단(45, 47줄)에서 설정한 대로 loss는 cross entropy의 softmax값으로 이를 줄이는 방향으로 gradient descent를 통해 training을 할 수 있도록 session에 전달한다. 이를 통해 batch의 cost값을 받는다. 이 받은 cost값을 한 epoch당 전체 batch 사이즈로 나누어 더하여 epoch이 끝났을 때 한 epoch당 batch당 평균 cost를 받을 수 있도록 한다.

61~63: 매 epoch마다 training average loss를 출력하고자 함으로 display step을 1로 설정하여 average loss와

learning rate을 출력한다.

65, 66: 학습을 완료한 LeNet의 예측 값을 실제 labeling과 비교하여 얼마나 정확한지를 leNet의 예측 값인 logit과 실제 labeling Y값과의 비교를 통해 측정한다. argmax를 통해 실제 어떤 label이 제일 큰지 (logit의 경우는 LeNet이 어떤 label로 판단했는지, Y의 경우는 실제 어떤 label인지)를 알아낸 뒤, eqaul함수를 통해 이 label의 결과가 같은지를 비교하여 같으면 true, 다르면 false의 값을 가진 tensor값으로 받게 된다.

67 : 비교 tensor값에서 true인 값은 1, false인 값은 0으로 환산하여 실제 전체 tensor 의 value들 중 몇 개가 true인지에 대한 비율을 tf.reduce_mean을 통해 부동소수점으로 환산하여 이를 저장한다.

68 : test case에 대한 loss를 알 수 있도록 session을 통해 실제 mnist.test 값들을 feed_dict로 전달하여 실제 LeNet의 test case에 대한 performance를 측정한다. 이에 따른 정확도 환산을 67줄에서 정의한 accuracy의 방법과 기존의, loss를 계산하는 방법인 cross entropy의 soft max인 loss_op를 전달하여 test case에서의 정확도와 loss를 받는다.

69:이를 출력한다.

3) Screenshot of output

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
('Train Set: Epoch', '1', 'Average Loss: 1.242107', 'lr:1.00e-02')
('Train Set: Epoch', '2', 'Average Loss: 0.616201', 'lr:1.00e-02')
('Train Set: Epoch', '3', 'Average Loss: 0.451092', 'lr:1.00e-02')
('Train Set: Epoch', '4', 'Average Loss: 0.370110', 'lr:1.00e-02')
('Train Set: Epoch', '5', 'Average Loss: 0.321073', 'lr:1.00e-02')
('Train Set: Epoch', '6', 'Average Loss: 0.289542', 'lr:1.00e-02')
('Train Set: Epoch', '7', 'Average Loss: 0.265523', 'lr:1.00e-02')
('Train Set: Epoch', '8', 'Average Loss: 0.248260', 'lr:1.00e-02')
('Train Set: Epoch', '9', 'Average Loss: 0.232403', 'lr:1.00e-02')
('Train Set: Epoch', '10', 'Average Loss: 0.221131', 'lr:1.00e-02')
('Train Set: Epoch', '11', 'Average Loss: 0.209980', 'lr:1.00e-02')
('Train Set: Epoch', '12', 'Average Loss: 0.200984', 'lr:1.00e-02')
('Train Set: Epoch', '13', 'Average Loss: 0.193593', 'lr:1.00e-02')
('Train Set: Epoch', '14', 'Average Loss: 0.185907', 'lr:1.00e-02')
('Train Set: Epoch', '15', 'Average Loss: 0.181115', 'lr:1.00e-02')
('Train Set: Epoch', '16', 'Average Loss: 0.175728', 'lr:1.00e-02')
('Train Set: Epoch', '17', 'Average Loss: 0.170749', 'lr:1.00e-02')
('Train Set: Epoch', '18', 'Average Loss: 0.165855', 'lr:1.00e-02')
('Train Set: Epoch', '19', 'Average Loss: 0.162088', 'lr:1.00e-02')
('Train Set: Epoch', '20', 'Average Loss: 0.157862', 'lr:1.00e-02')
('Test Set: Average Loss : 0.150315', 'Accuracy:', '9549/10000', '(95.49%)')
```

4) Analasys of training process and result.

A. Average Loss의 감소
epoch을 거듭할수록 Average Loss는 감소하였다. Gradient Descent 알고리즘에 의해서 Average Loss가 감소하는 방향으로 각 weight들이 그 값을 바꾸기 때문일 것이다.

B. Learning Rate의 크기
Learning Rate의 크기가 조금만 커져도 그 Average Loss의 감소 정도가 확연히 다르다. 0.01인 현재의 learning rate을 0.1로 수정하는 경우 Average Loss가 2.3 언저리에서 전혀 발전하지(줄어들지) 못하는 모습을 보인다. 이는 learning rate자체가 너무 크기 때문에 gradient가 줄어드는 방향으로

feature들을 이동하더라도, 실제 loss값이 줄어드는 것보다 더욱 큰 방향으로 이동되기 때문으로 보인다.

C. stddev의 크기

초기 weight들을 임의 설정할 때, stddev를 기존에 주어진 1이나, 현재의 0.1보다는 큰 0.5 등으로 설정할 경우 average loss가 감소하는 추세가 상당히 늦다. 이는 초반에 설정된 임의의 값의 편차가 gradient 값이나 learning rate에 비해 너무 크면, 임의의 값이 loss 가 줄어드는 방향으로 이동하는 것에 비해 너무 크기 때문으로 분석된다. 따라서 epoch값을 크게 하는 경우, 결국은 다시 average loss가 줄어들고 accuracy가 높아지는 방향으로 학습을 진행하게 된다.