

AI Programming

Lab 03: Optimizers for DNN

소속: 컴퓨터정보공학부

학번: 2019202103

이름: 이은비

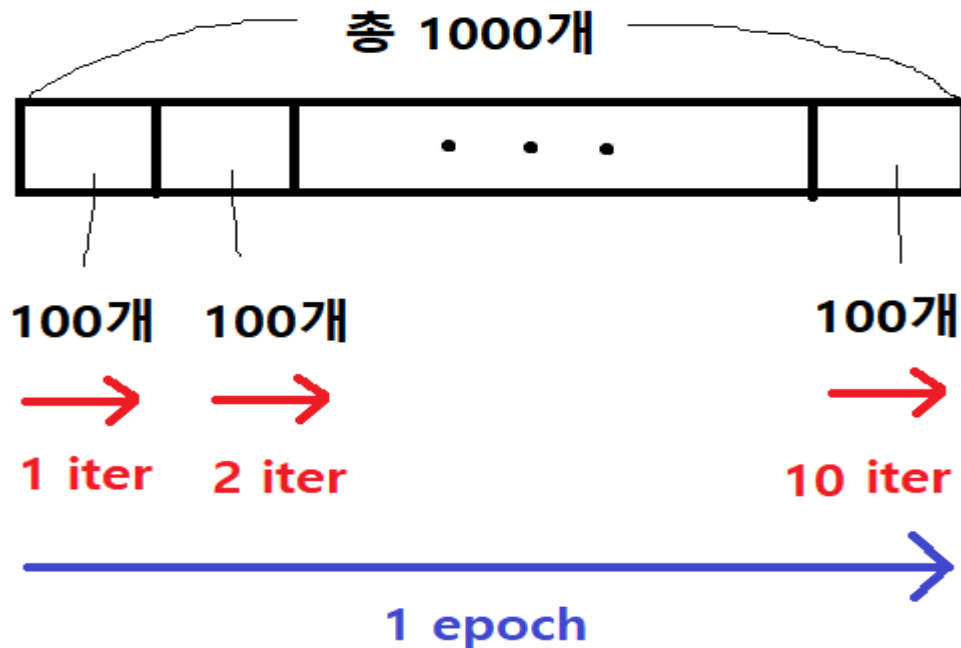
제출날짜: 2023/10/17

<Lab Objective>

Lab03의 주제에 맞게 DNN (Deep Neural Network)의 최적화를 하는 과정을 거칩니다. 이전 Lab의 주제와 내용에 맞춰 전개된 과정을 간단하게 정리하면 데이터 전처리후, forward와 backward에서 쓰이는 sigmoid 와 softmax 같은 함수를 먼저 정의하고 forward와 backward과정에서 matrix multiplication을 진행하여서 그에 대한 loss와 accuracy를 일정 epoch마다 출력하였습니다. 이를 응용하여서 Lab03에서는 create_mini_batches라는 함수를 추가하였는데, 설명하기에 앞서 데이터를 하나씩 학습시키는 방법과 전체를 학습시키는 방법의 장단점을 보면 데이터를 하나 하나씩 신경망에 넣어 학습을 시키면 우선 장점으로는 신경망을 한번 학습시키는데 소요되는 시간이 매우 짧은 장점이 있지만 training dataset 전체에 대해서 training하기 때문에 계산 속도가 사실상 늦어집니다. GPGPU의 병렬처리를 통해 이와 같은 결점을 보완하지만 데이터를 한 개 씩 넣는다면 사실상 앞서 언급한 GPU의 병렬처리를 사용하지 않으니 그만큼 자원 낭비가 되며 오차를 줄이기 위해 사용하는 Loss Function에서 최적의 파라미터를 설정하는데 상당히 많이 해매게 되는 것을 알 수 있습니다.

전체 데이터를 입력하는 경우는 한번에 여러 개의 데이터에 대해서 신경망을 학습시킬 수 있으므로 오차를 줄일 수 있는 cost function의 최적의 parameter를 하나씩 학습하는 것보다 빠르게 알아낼 수 있다. LAB02에서도 이 경우처럼 전체 데이터셋을 input으로하여 그 과정을 진행하였습니다. 하지만 전체 데이터를 학습시키기 때문에 신경망을 한 번 학습시키는데 소요되는 시간이 매우 긴 것을 알 수 있습니다. 따라서 앞서 언급된 문제점을 보완하고 장점을 적용 시킨게 mini-batch인데,

이는 SGD(Stochastic Gradient Descent : 확률적 경사 하강법)와 배치를 섞은 것으로 전체 데이터를 N등분하여 각각의 학습 데이터를 배치 방식으로 학습시킵니다. 따라서 최대한 신경망을 한 번 학습시키는데 걸리는 시간을 줄이면서 전체 데이터를 반영할 수 있게 되며 효율적으로 CPU와 GPU를 활용할 수 있게 됩니다. 즉, Batch와 SGD의 단점을 극복, 절충한 개념인 Mini-batch라는 학습방식을 사용하는 것으로, 모든 데이터에 대해서 가중치 평균을 구하는 것이 아니라 전체 데이터에서 일부데이터를 묶음방식으로 하여서 데이터 학습을 진행시킵니다. (아래 그림 참고, 출처: <https://mole-starseeker.tistory.com/59>)



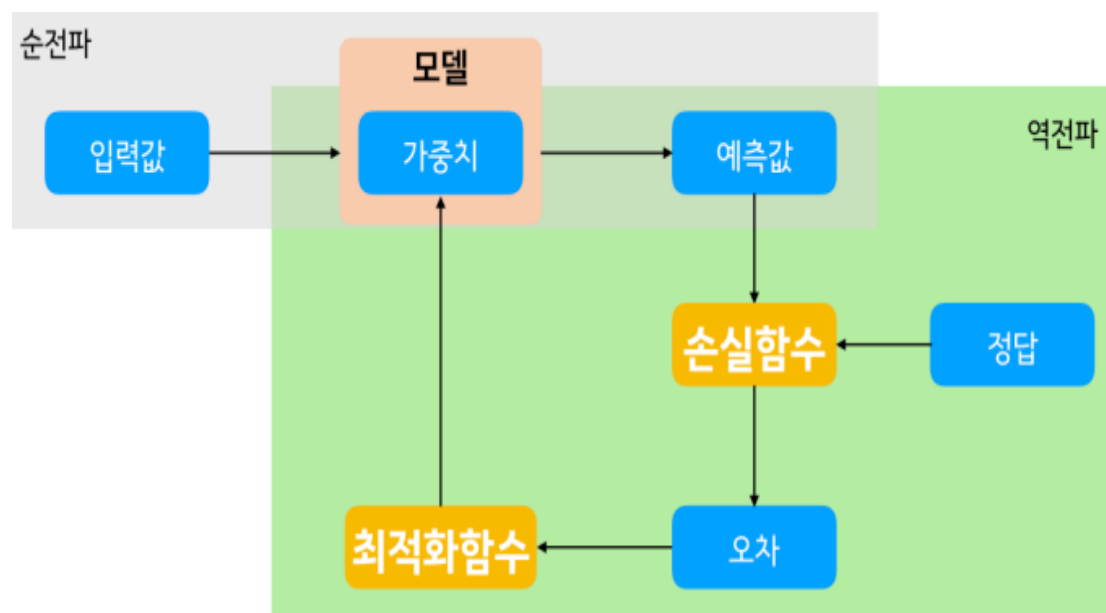
또한 optimizer settings 을 'sgd', 'momentum', 'adagrad', 'rmsprop', 'adam' 으로 하여서 그에 해당하는 optimizer를 set해서 각 optimizer에 따라 다르게 update된 weight와 bias를 출력하도록 합니다. 그리고 이후의 과정과 위에서 언급한 사항 외에는 동일한 flow로 진행하도록 합니다. 즉, 이번 LAB03에서는 mini_batch와 optimization function을 때에 따라 다르게 적용함으로써 Deep Neural Network의 optimization을 진행합니다.

<Whole simulation program flow>

LAB02에서와 기본적인 구조는 거의 비슷합니다. 아래 그림과 같은 epoch마다 아래 그림과 같은 과정을 거치는데 LAB02에서는 최적화 함수를 여러 개 두지 않은 것과 달리 LAB03에서는 최적화 함수를 여러 개 두었으며, 입력 또한 LAB02에서는 epoch 즉, training dataset 전체를 소진하는 것과 달리 mini batch방법을 통해 전체 training set을 여러 개로 나누어 input으로 합니다. 그리고 아래 그림과 같은 전체 flow를 loop를 통해 여러 번 거치게 함으로써 model training을 진행하도록 합니다. 데이터셋 로드 및 전처리->forward path에서 쓰이는 activation함수인 sigmoid function과 softmax function을 정의합니다->forward와 backward 함수를 정의합니다. ->Backward path에서 쓰이는 activation함수인 dJdz_sigmoid와 dJdz_softmax에 대해서 정의합니다.->forward path를 (my_forward라는 이름의 function) sigmoid,softmax와 forward함수를 이용하여 정의합니다.->backward path. 즉, back

propagation (my_backward라는 이름의 function)을 진행하는 함수를 앞서 정의한 dJdz_softmax, dJdz_sigmoid와 backward 함수를 이용하여 정의합니다. 그리고 my_loss 함수, 즉, loss 값을 return 하는 함수와 my_predict 함수 즉, y의 prediction 값을 return 하는 함수를 정의합니다. -> layer의 parameter 값을 initialize, weight initialize -> epoch. 즉, 전체 dataset를 create_mini_batches 함수를 통해 나눕니다. -> optimizer function을(sgd, momentum, adagrad, rmsprop, adam)을 정의합니다. -> 각 optimization function을 통과한 layer의 weight와 bias 값을 출력합니다. -> parameter 값을 set하고 epoch에 따른 loss 값을 출력합니다. y의 prediction 값을 구하고 그 값을 이용한 accuracy를 확인합니다. MLPclassifier를 이용한 model training and fitting을 진행한 후의 accuracy도 계산 및 출력합니다. -> 해당 data를 plt를 이용하여 plot하고, prediction 값과 실제 값을 확인합니다.

(아래 그림 출처: <https://welcome-to-dewy-world.tistory.com/86>)



<Brief comments on each code block (English only)>

Prepare Mini-MNIST Dataset(Data Preprocessing)

-> # Activation function for forward path -> # check the data in the preprocessed data set

-> # check the data in the preprocessed data set -> # Define Model class (Linear prediction for each function (forward,backward))

-> # Code for checking backward path process -> # activation functions for the backpropagation.

-> # Define Training Functions -> # set the parameters of each layer

-> # Initialize the weights -> # Define a function for Splitting Dataset into mini-Batches
(create_mini_batch) -> # check the result of the 'create_mini_batches' function

-> # Define Various Optimizers -> # optimizer functions (sgd, momentum, adagrad, rmsprop, adam)

-> # Optimizer Test -> # Create Optimizer Parameters (apply function to each layer)

-> # Trainig Simple Neural Network Model(3 layer model) -> # Evaluate Model Performance

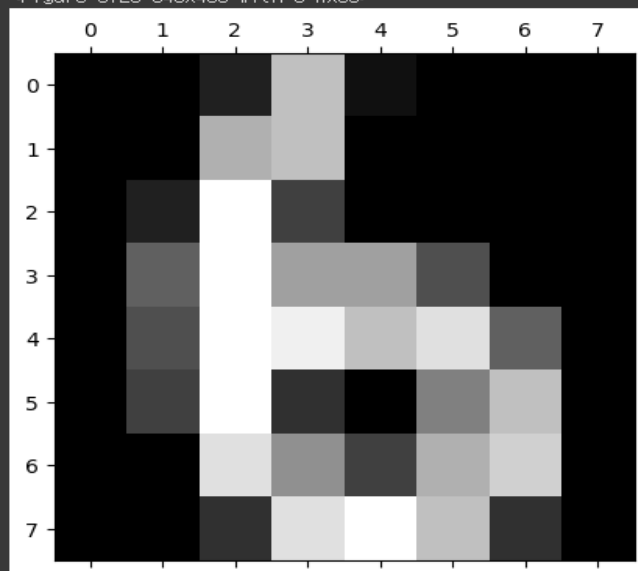
-> # Neural Network from scikit-learn -> #Test Model with a random sample

<Simulation results (screenshots)>

```
idx = np.random.randint(X_train.shape[0])
dimage = X_train_orig[idx].reshape((8,8))
# plt.figure(figsize=(4.32, 2.88)) # Adjust the width and height as needed
plt.gray()
plt.matshow(dimage)
plt.show()
print('The number is', y_train_num[idx]) # the number that is predicted
```

```
(1797, 64)
(1437, 64)
(1437, 10)
[ 0.  0.  0.  9. 16.  6.  0.  0.  0.  0.  4. 15.  6. 15.  0.  0.  0.  0.
  8. 11.  9. 11.  0.  0.  0.  0.  8. 16. 14.  2.  0.  0.  0.  0. 11. 16.
 13.  0.  0.  0.  0.  6. 14.  2. 12.  9.  0.  0.  0.  5. 16. 11.  5. 13.
  4.  0.  0.  0.  3.  8. 13. 16.  9.  0.]
```

<Figure size 640x480 with 0 Axes>



The number is 6

```

# Code for checking backward path process
np.random.seed(0)

tmp2 = myDenseLayer(2,5)
tmp2.wegt = np.random.randn(2,5)
tmp2.bias = np.random.randn(2)

x_in_t = np.random.randn(2,5)
x_t = np.random.randn(2,2)

dw2, db2, wdJdz2 = tmp2.backward(x_t,x_in_t)

print(f"x_t = \n{x_t}")
print(f"dw2 = \n{dw2}")
print(f"db2 = \n{db2}")
print(f"wdJdz2 = \n{wdJdz2}")

x_t =
[[ 0.8644362 -0.74216502]
 [ 2.26975462 -1.45436567]]
dw2 =
[[ 0.09610482  0.40788358 -0.77744815 -2.75311014  1.38754493]
 [-0.13322022 -0.27280893  0.45637388  1.73266967 -1.02972684]]
db2 =
[ 1.56709541 -1.09826535]
wdJdz2 =
[[ 2.25021216 -0.35921201  0.95838857  2.01371462  1.30965288]
 [ 5.42528537 -0.47351731  2.44162379  5.23639566  3.64173802]]

```

```

# set the parameters of each layer.
n_inputs = 64
n_hidden1 = 80
n_hidden2 = 70
n_classes = 10

l1 = myDenseLayer(n_hidden1, n_inputs)
l2 = myDenseLayer(n_hidden2, n_hidden1)
l3 = myDenseLayer(n_classes, n_hidden2)

layers = [l1, l2, l3]
print(X_train.shape, y_train.shape)
print(l1.wegt.shape, l1.bias.shape)
print(l2.wegt.shape, l2.bias.shape)
print(l3.wegt.shape, l3.bias.shape)

(1437, 64) (1437, 10)
(80, 64) (80,)
(70, 80) (70,)
(10, 70) (10,)

```

```

np.random.seed(1)

a = np.arange(20).reshape(10,2)
b = -np.arange(10,20).reshape(10,1)
# check the function(create_mini_batches)
c = create_mini_batches(a, b, 4)
for mini_X, mini_y in c:
    print(mini_X)
    print(mini_y, '\n')

```

```

[[ 4  5]
 [18 19]
 [12 13]
 [ 8  9]]
[[-12]
 [-19]
 [-16]
 [-14]]

[[ 0  1]
 [ 6  7]
 [ 2  3]
 [14 15]]
[[-10]
 [-13]
 [-11]
 [-17]]

[[16 17]
 [10 11]]
[[-18]
 [-15]]

```

아래 optimization function을 적용할 때 각 function마다 주석을 on/off해서 적용한 결과 입니다.

```

my_optimizer(lyr, opt, W_grad, B_grad, 'sgd', 10, 3)
print("For SGD:")
print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'momentum', 10, 3)
# print("For Momentum:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'adagrad', 10, 3)
# print("For Adagrad:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'rmsprop', 10, 3)
# print("For RMSProp:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'adam', 10, 3)
# print("For adam:")
# print(lyr.wegt[0], lyr.bias[0])

```

```

For SGD:
[8.49607236 7.8403      6.18428956] -14.853210006882223

```

```
# my_optimizer(lyr, opt, W_grad, B_grad, 'sgd', 10, 3)
# print("For SGD:")
# print(lyr.wegt[0], lyr.bias[0])
my_optimizer(lyr, opt, W_grad, B_grad, 'momentum', 10, 3)
print("For Momentum:")
print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'adagrad', 10, 3)
# print("For Adagrad:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'rmsprop', 10, 3)
# print("For RMSProp:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'adam', 10, 3)
# print("For adam:")
# print(lyr.wegt[0], lyr.bias[0])
```



For Momentum:
[8.20893718 8.06473334 4.86839241] -15.873602504984119

```
# my_optimizer(lyr, opt, W_grad, B_grad, 'sgd', 10, 3)
# print("For SGD:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'momentum', 10, 3)
# print("For Momentum:")
# print(lyr.wegt[0], lyr.bias[0])
my_optimizer(lyr, opt, W_grad, B_grad, 'adagrad', 10, 3)
print("For Adagrad:")
print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'rmsprop', 10, 3)
# print("For RMSProp:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'adam', 10, 3)
# print("For adam:")
# print(lyr.wegt[0], lyr.bias[0])
```



For Adagrad:
[7.48870745 6.53638603 8.33582567] -7.195664047068676

```
# my_optimizer(lyr, opt, W_grad, B_grad, 'sgd', 10, 3)
# print("For SGD:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'momentum', 10, 3)
# print("For Momentum:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'adagrad', 10, 3)
# print("For Adagrad:")
# print(lyr.wegt[0], lyr.bias[0])
my_optimizer(lyr, opt, W_grad, B_grad, 'rmsprop', 10, 3)
print("For RMSProp:")
print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'adam', 10, 3)
# print("For adam:")
# print(lyr.wegt[0], lyr.bias[0])
```




For RMSProp:
[9.04298935 9.5870574 16.48540354] -15.77061874091831


```

# my_optimizer(lyr, opt, W_grad, B_grad, 'sgd', 10, 3)
# print("For SGD:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'momentum', 10, 3)
# print("For Momentum:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'adagrad', 10, 3)
# print("For Adagrad:")
# print(lyr.wegt[0], lyr.bias[0])
# my_optimizer(lyr, opt, W_grad, B_grad, 'rmsprop', 10, 3)
# print("For RMSProp:")
# print(lyr.wegt[0], lyr.bias[0])
my_optimizer(lyr, opt, W_grad, B_grad, 'adam', 10, 3)
print("For adam:")
print(lyr.wegt[0], lyr.bias[0])

```

 For adam:
 [3.12812265 6.11310547 -0.00659625] -5.620045550802262

```

optimizer = 'sgd' # set the optimizer
alpha = 0.01 # set the value of alpha that is learning rate.
n_epochs = 1000 # set the number of epochs

for epoch in range(n_epochs):

    batches = create_mini_batches(X_train, y_train, batch_size=64)
    for one_batch in batches:
        X_mini, y_mini = one_batch
        batch_len = X_mini.shape[0] # last batch might have different length

        # Forward Path
        a_1, a_2, a_3 = my_forward(layers, X_mini)

        # Backward Path
        d_1, d_2, d_3 = my_backward(layers, a_1, a_2, a_3, X_mini, y_mini)

        dw_1, db_1 = d_1
        dw_2, db_2 = d_2
        dw_3, db_3 = d_3

        # Update weights and biases
        my_optimizer(l1, o1, dw_1, db_1, solver=optimizer, learning_rate=alpha, iter=epoch+1)
        my_optimizer(l2, o2, dw_2, db_2, solver=optimizer, learning_rate=alpha, iter=epoch+1)
        my_optimizer(l3, o3, dw_3, db_3, solver=optimizer, learning_rate=alpha, iter=epoch+1)

    if ((epoch+1)%100==0):
        loss_J = my_loss(layers, X_train, y_train)
        print('Epoch: %4d, loss: %10.8f' % (epoch+1, loss_J))

```

```

Epoch: 100, loss: 0.34530735
Epoch: 200, loss: 0.19123897
Epoch: 300, loss: 0.12508008
Epoch: 400, loss: 0.08817055
Epoch: 500, loss: 0.06534233
Epoch: 600, loss: 0.05036992
Epoch: 700, loss: 0.04011914
Epoch: 800, loss: 0.03284979
Epoch: 900, loss: 0.02752606
Epoch: 1000, loss: 0.02351377

```

```
from sklearn.metrics import accuracy_score
```

```
y_pred = my_predict(layers, X_test)
```

```
# calculate the accuracy
```

```
accuracy_score(y_pred, y_test)
```

```
0.9388888888888889
```

```
from sklearn.neural_network import MLPClassifier
```

```
# set the parameters of MLPClassifier
```

```
mlp = MLPClassifier(hidden_layer_sizes=(80, 70, ), activation='logistic', solver='sgd', #  
                    alpha=0.01, learning_rate_init=0.01, max_iter=1000)
```

```
# Training/Fitting the Model
```

```
mlp.fit(X_train, y_train_num)
```

```
# Making Predictions
```

```
s_pred = mlp.predict(X_test)
```

```
accuracy_score(s_pred, y_test) # calculate the accuray
```

```
0.9694444444444444
```

```
X_input = np.expand_dims(X_test[idx], 0)
```

```
y_pred = my_predict(layers, X_input)
```

```
s_pred = mlp.predict(X_input)
```

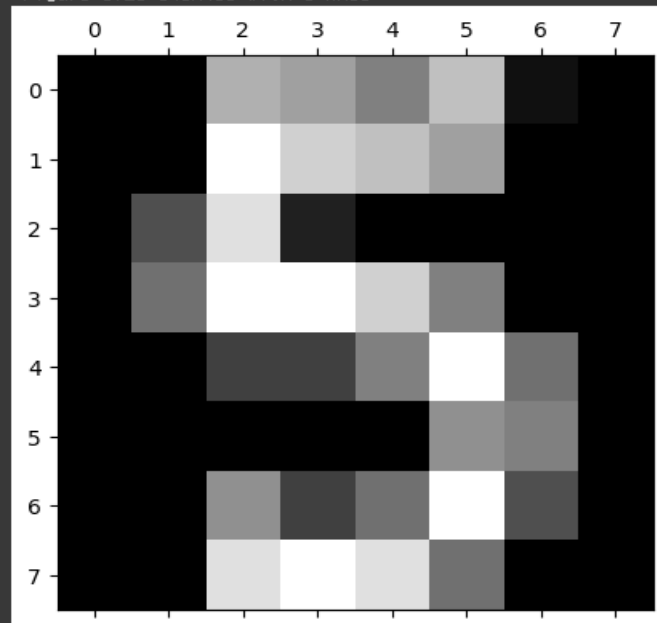
```
# print out the predicted number.
```

```
print('My prediction is ' + str(y_pred[0]))
```

```
print('sk prediction is ' + str(s_pred[0]))
```

```
print('Actual number is ' + str(y_test[idx]))
```

<Figure size 640x480 with 0 Axes>



```
My prediction is 5
```

```
sk prediction is 5
```

```
Actual number is 5
```

<Discussion>

Dimension 이 안 맞아 이후의 epoch 마다의 loss 출력은 물론 그에 따라 accuracy 또한 출력되지 않았습니니다. forward 의 input 으로 들어가는 shape size 가 달라서 forward function call 이 진행되지 않아 입력하는 형식을 변화시켜서 reshape 하는 과정을 거쳤는데,

my_forward function call 진행한 후 결과가 layer 를 차례로 통과하면서 결과적으로 확인된 shape 가 (73,10)처럼 (,10)이 결과가 되고 따라서 그에 맞게 my_backward function call 했을 때 operands could not be broadcast together with shapes (73,10) (1,64) 와 같은 오류가 발생했습니다. (아래 그림 참고) 그래서 중간에 y_mini 에 대한 shape 변경 등 이 코드 블록안에서 해결되지 않아서 그래서 코드 블록의 처음부터 확인하면서 다시 작성하는 과정을 거쳤는데 마침 backward 확인용 코드가 공지로 올라와서 확인한 결과 이것이 잘못된 것을 알 수 있었습니다. 그런데 알고보니 이 부분이 잘못되어 있어서 다시 수업자료를 확인하고 수업자료에 맞게 식을 수정하는 과정을 거쳤습니다. 그렇게 정정하고 나니 이후의 과정이 정상적으로 출력하였습니다.

```
(73, 64) (1, 64)
-----
ValueError                                Traceback (most recent call last)
<ipython-input-23-d2bd76d985ba> in <cell line: 7>()
    21         #Backward Path
    22
--> 23     d_1, d_2, d_3 = my_backward(layers, a_1, a_2, a_3, X_mini, y_mini)
    24
    25     dw_1, db_1 = d_1

----- 1 frames -----
<ipython-input-13-66a22c63019f> in dJdz_softmax(y_hat, y)
     5
     6 def dJdz_softmax(y_hat, y):
--> 7     dJdz = y_hat - y # backpropagation through activation function
     8     return dJdz

ValueError: operands could not be broadcast together with shapes (73,10) (1,64)
```