HW #11
Machine Leaning

21600004
Kang Seok-Un

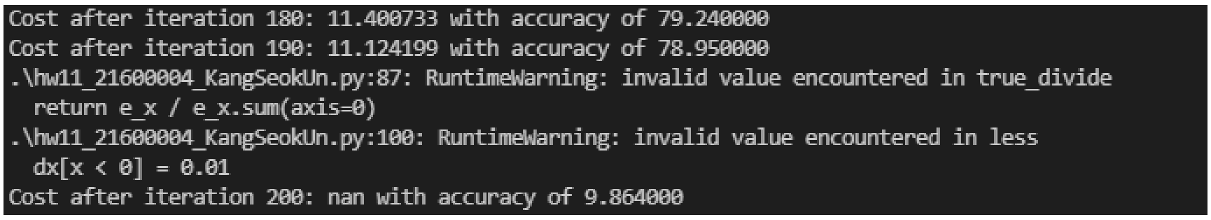
Introduction

In this assignment, like many previous assignments, a famous public dataset MNIST dataset is used. In particular, in this assignment, task #1 implementing forwarding and back-propagation on my own and task #2 implemented using PyTorch library are performed. And the execution time and accuracy of Task #1 and Task#2 are compared.

Experiment

Task#1: Own Back-Propagation MLP

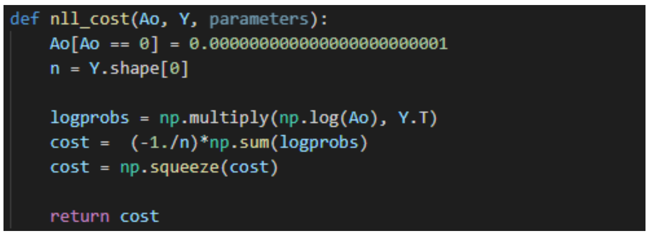
The activation function used in this task is sigmoid. The reasons for using Sigmoid are as follows. Initially, the ReLU function was intended to be used as an activation function. However, as can be seen in Figure 1, an error occurred that did not know exactly what the problem was. Many trials were conducted to solve this problem, but no solution could be found. Therefore, I tried to solve the problem by changing the activation function from ReLU to Sigmoid.



```
Cost after iteration 180: 11.400733 with accuracy of 79.240000
Cost after iteration 190: 11.124199 with accuracy of 78.950000
.\hw11_21600004_KangSeokUn.py:87: RuntimeWarning: invalid value encountered in true_divide
  return e_x / e_x.sum(axis=0)
.\hw11_21600004_KangSeokUn.py:100: RuntimeWarning: invalid value encountered in less
  dx[x < 0] = 0.01
Cost after iteration 200: nan with accuracy of 9.864000
```

Figure 1. Error Occurred by activation function "ReLU"

In addition, in Figure 2, when Ao is 0, an error occurred because log (0) was not defined. To prevent this, if Ao is 0, Ao is changed to a very small number,10e-24.

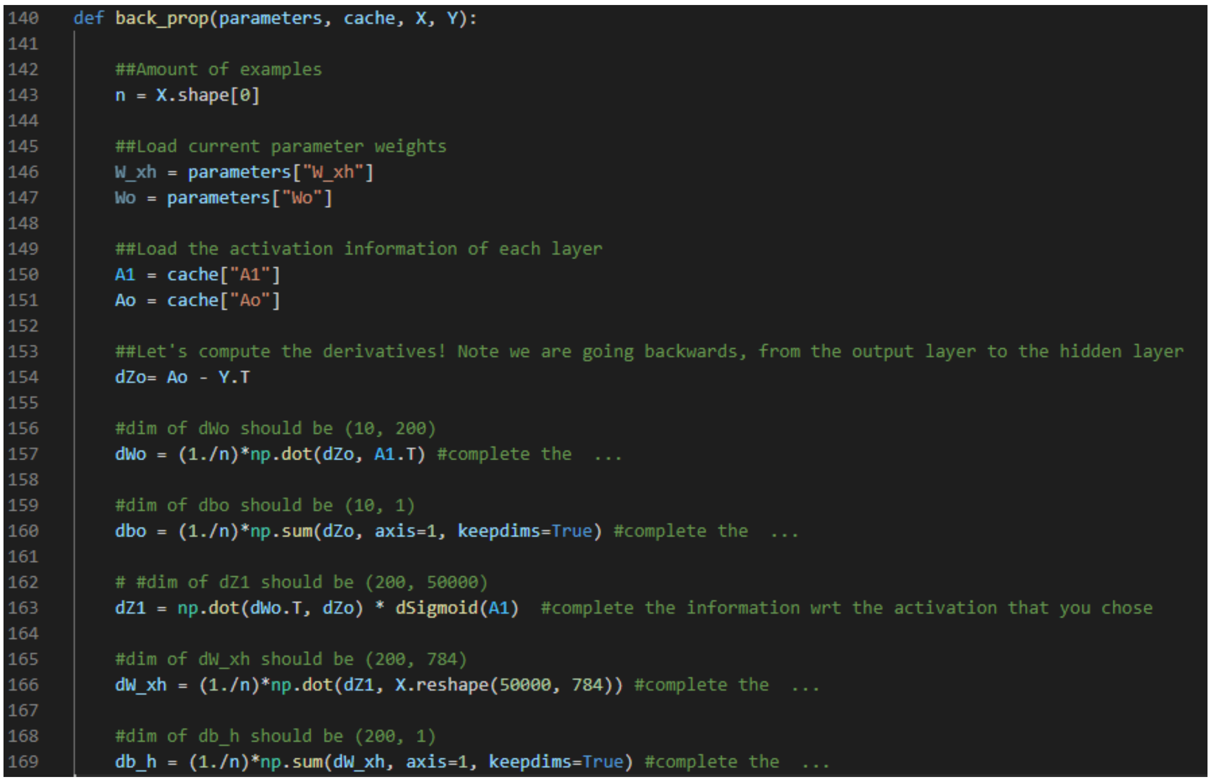


```
def nll_cost(Ao, Y, parameters):
    Ao[Ao == 0] = 0.000000000000000000000001
    n = Y.shape[0]

    logprobs = np.multiply(np.log(Ao), Y.T)
    cost =  (-1./n)*np.sum(logprobs)
    cost = np.squeeze(cost)

    return cost
```

Figure 2. The log zero error

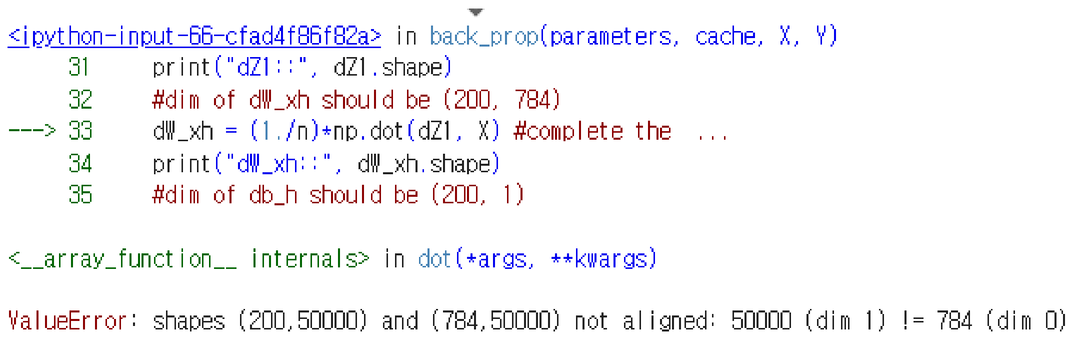
In the case of Back Propagation, TA provided an excellent catch, making it much easier to implement the RBF Network, the last assignment. First, I tried to fill the blank by referring to pdf p.19 to p.22. In particular, as indicated by the comment, great effort was made to conform to the shape of the data.

Also, let's look at line 166 of Figure 3. The reason why "X.reshape(50000, 784)" is used in this part is in Figure 4. In the case of Figure 4, it is still before converting from colab to local python code. However, as you can see, it is a problem of shape or dimension. However, if print out the shape of Input X, it is [50000, 784], so it is not a problem even if does not transpose. In addition, since the dimension is marked 1 and 0, I tried to forcefully fit it using the reshape function.



```python
def back_prop(parameters, cache, X, Y):

    ##Amount of examples
    n = X.shape[0]

    ##Load current parameter weights
    W_xh = parameters["W_xh"]
    Wo = parameters["Wo"]

    ##Load the activation information of each layer
    A1 = cache["A1"]
    Ao = cache["Ao"]

    ##Let's compute the derivatives! Note we are going backwards, from the output layer to the hidden layer
    dZo= Ao - Y.T

    #dim of dWo should be (10, 200)
    dWo = (1./n)*np.dot(dZo, A1.T) #complete the  ...

    #dim of dbo should be (10, 1)
    dbo = (1./n)*np.sum(dZo, axis=1, keepdims=True) #complete the  ...

    # #dim of dZ1 should be (200, 50000)
    dZ1 = np.dot(dWo.T, dZo) * dSigmoid(A1)  #complete the information wrt the activation that you chose

    #dim of dW_xh should be (200, 784)
    dW_xh = (1./n)*np.dot(dZ1, X.reshape(50000, 784)) #complete the  ...

    #dim of db_h should be (200, 1)
    db_h = (1./n)*np.sum(dW_xh, axis=1, keepdims=True) #complete the  ...
```

Figure 3. The implementation about Back-Propagation

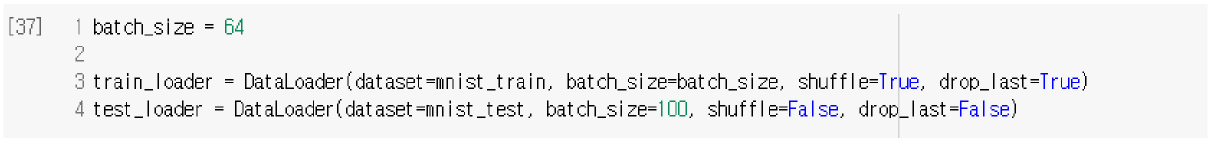


```
<ipython-input-66-cfad4f86f82a> in back_prop(parameters, cache, X, Y)
     31     print("dZ1::", dZ1.shape)
     32     #dim of dW_xh should be (200, 784)
---> 33     dW_xh = (1./n)*np.dot(dZ1, X) #complete the  ...
     34     print("dW_xh::", dW_xh.shape)
     35     #dim of db_h should be (200, 1)

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (200,50000) and (784,50000) not aligned: 50000 (dim 1) != 784 (dim 0)
```

Figure 4. Error Occurred by shape or dimension of input data X

Task#2: MLP with Pytorch

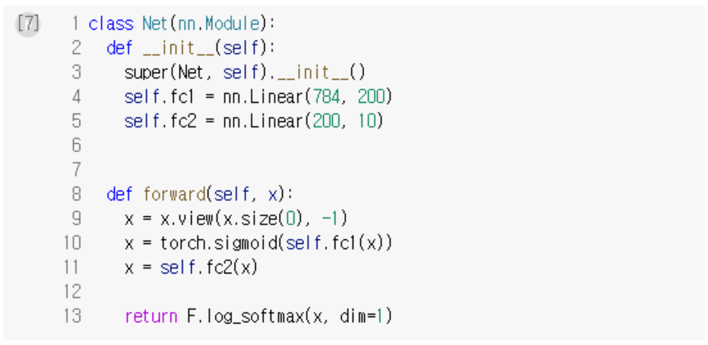
When implementing MLP with Pytoch, batch was used to learn train data efficiently (Figure 5).



```
[37]
batch_size = 64

train_loader = DataLoader(dataset=mnist_train, batch_size=batch_size, shuffle=True, drop_last=True)
test_loader = DataLoader(dataset=mnist_test, batch_size=100, shuffle=False, drop_last=False)
```

Figure 5. Declaring the batch size and data loader with data loader library

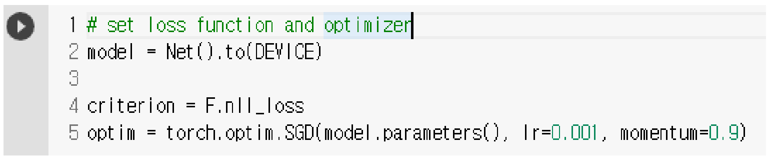
In the case of Network, as can be seen in Figure 6, it is implemented in the same way as MLP implemented on my own.



```
[7]
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(784, 200)
    self.fc2 = nn.Linear(200, 10)


  def forward(self, x):
    x = x.view(x.size(0), -1)
    x = torch.sigmoid(self.fc1(x))
    x = self.fc2(x)

    return F.log_softmax(x, dim=1)
```

Figure 6. Declaring the Network

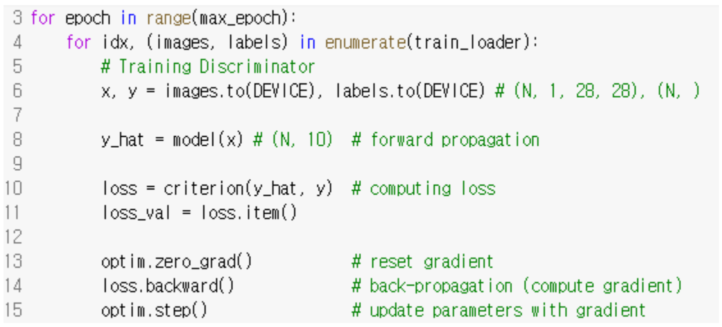
Stochastic gradient design (SGD) was used as the optimization method. This does not update weight using the entire dataset, but updates weight only with batch.



```
# set loss function and optimizer
model = Net().to(DEVICE)

criterion = F.nll_loss
optim = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

Figure 7. Declaring the loss function and Optimization Method

Figure 8 shows the training part. This part is implemented very similar to the tutorial.



```
for epoch in range(max_epoch):
    for idx, (images, labels) in enumerate(train_loader):
        # Training Discriminator
        x, y = images.to(DEVICE), labels.to(DEVICE) # (N, 1, 28, 28), (N, )

        y_hat = model(x) # (N, 10)  # forward propagation

        loss = criterion(y_hat, y)  # computing loss
        loss_val = loss.item()

        optim.zero_grad()           # reset gradient
        loss.backward()             # back-propagation (compute gradient)
        optim.step()                # update parameters with gradient
```

Figure 8. The Implementation about Training Part with PyTorch

Result

First, the parameters used in the imprinted MLP and MLP with PyTorch will be referred to in table 1. In the case of MLP with PyTorch, since it uses batch, it is somewhat far from comparing the Implemented MLP model that updates weight using the entire dataset with epoch alone, so I tried to compare it with # of steps.

I know that the scratch code says not to decrease the epoch below 10000. However, the training time took too long to set the epoch to 10000 and proceed with the test, so the epoch was arbitrarily set to 3000.

In addition, it was judged that the CPU performance of the PC currently in use was better than that of colab, so the experiment was conducted on the local PC.

Table 1. Used Parameters

	Implemented MLP	MLP with PyTorch
Learning Rate	0.01	0.001
# of steps(epoch)	3000	3000

Table 2 shows the performance of the two models. First of all, the model with a slightly lower loss is the Implemented MLP. However, in the case of Implemented MLP, which took more than 16 times the learning time, the accuracy was 88.77%, which is lower than 89.82% of MLP with PyTorch.

Table 2. Performance

	Implemented MLP	**MLP with PyTorch**
Loss for Test	**0.378297**	0.5413635
Accuracy for Test	88.77%	**89.82%**
Executed Time	3310.987s	**193.664s**

Figure 9 is an image obtained by plotting the loss value measured for each step. Since the initial loss value of the Implemented MLP was greater than that of the MLP with PyTorch, it can be seen that the slope is relatively steep. In addition, it can be seen that the Implemented MLP is smaller even in the case of loss of the test conducted on the test data.

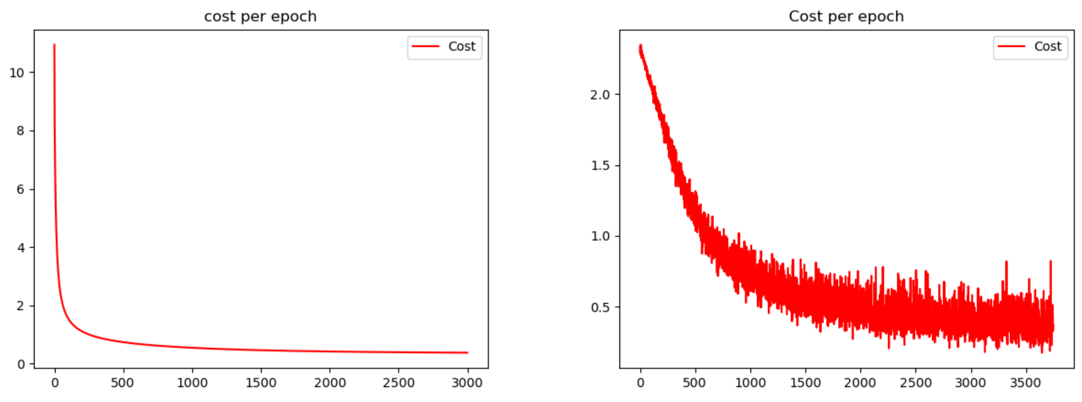


Figure 9. Loss Value per epoch (Left: Implemented MLP, Right: MLP with PyTorch)

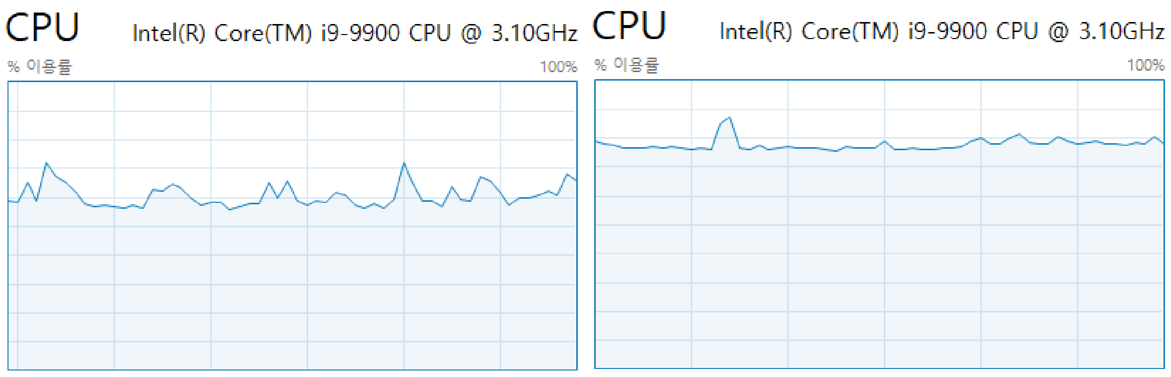


Figure 10. CPU Utilization (Left: Implemented MLP, Right: MLP with PyTorch)

Figure 11 is attached to analyze the learning speed with respect to CPU utilization. First, the Utilization of CPU is about 60% when training using the Implemented MLP Model. On the other hand, it can be seen that the MLP with PyTorch Model is about 80%. Therefore, it is assumed that MLP with PyTorch's learning speed processing would have been faster due to the difference in CPU usage of 20%.

Also, since the PyTorch Library is a public library used by many people, it is considered to be the most refined code through many optimizations. Therefore, it is assumed that there was a difference in the overall optimization of the code because the function implemented in the most effective way was used.

In addition, in the case of MLP with PyTorch using Batch, it is inevitable to be faster than the implemented MLP Model learned 3000 times using the existing 50,000 train datasets.

Conclusion

I have experience implementing MLP using PyTorch in the other class. At that time, it was a Black Box that I didn't know how weight changed (learned) while doing Network modeling. However, it is believed that a more mature understanding was achieved because it was implemented directly through this assignment.

And when using the MLP with PyTorch Model, it was confirmed that if the batch size is 50000, the utilization of CPU drops to 40%. I wonder why this phenomenon appears.

Also, without a kind scratch code and tutorial, this task would have given me more difficulty than the last assignment #10. Through this, I learned that if someone has a guide in studying, I can learn faster.

Link about MLP with PyTorch on Colab:
https://colab.research.google.com/drive/1BQnNOTFy9jlg-NA5FYrvl1ThZKnFW09K?usp=sharing