Summary Report [Week 9]. Classification

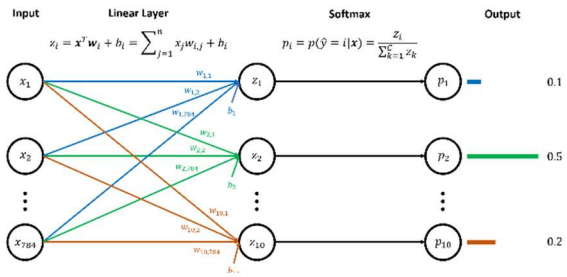
20190106 KimByungjun (김병준)

● Exercises

이번 Exercise 에선 자필로 작성된 digits image samples 이 담긴 MNIST dataset 을 single linear layer 과 softmax 가 결합된 model 과 activation function 이 동반된 multi-layer model 을 이용해 분류하는 것이 목적이다. 이 때, loss function 으로 cross entropy loss 를 사용하며, SGD(stochastic gradient descent) optimizer 를 model 에 적용한다.

주어진 MNIST dataset 내 각 sample 들은 28×28 의 2D size 를 가지며, 이를 size 784의 1D vector 로 flatten 했음을 알린다. 단일 vector 내 각 element 는 0(black)에서 1(white) 사이의 값을 가진다. 분류 output 으로 0~9 까지의 digit, 총 10 개의 classes 를 가진다.

첫째로, **[Fig 1]**과 같이 단일 linear layer 을 통과 후 softmax 를 이용해 각 label 에 대한 probability 를 계산하여 digit 을 추정하는 model 을 구현하고자 한다. 이 때, input layer 의 output z_i는 $z_i = \Sigma_{j=1}^{784} x_j w_{i,j} + b_i (i = 1,2,\dots,10)$ 이며, *softmax 의 결과로 $p_i = p(\hat{y} = i|\mathbf{x}) = \frac{z_i}{\Sigma_{k=1}^{10} z_k} (i = 1,2,\dots,10)$ 의 확률이 계산된다.



[Fig 1] Linear Layer model with Softmax function

```python
# Define a model
class LinearClassificationModel(nn.Module):
    def __init__(self, input_unit, output_unit):
        # input_unit: the number of input units (= input size)
        # output_unit: the number of output units (= the number of classes)
        super(LinearClassificationModel, self).__init__()

        """ ### Implement the code yourself ### """
        self.linear = nn.Linear(input_unit, output_unit)
        """ ######################################## """

    def forward(self, x):
        """ ### Implement the code yourself ### """
        out = self.linear(x)
        """ ######################################## """
        return out

model = LinearClassificationModel(input_unit=784, output_unit=10).to(device)
```

[Fig 2] Code for Exercise 1

1) Implement *linear* and implement the forward process of the model.

<Explanation>

[Fig 1]의 linear layer 을 구현한 code 가 **[Fig 2]**와 같다. 생성자 __init__에서 torch.nn 내 nn.Linear 함수를 호출한다. 이 때, 변수로 input_unit 과 output_unit 을 받으며 **[Fig 2]** 맨 아래를 보듯이 각각 784(= 28×28)와 10 을 넣게 된다. 추가로, linear layer 내 연산을 담당하는 forward 함수에선 self.linear(x) (x: input tensor) 을 통해 위에서 언급한 z_i를 계산한다.

```python
def cross_entropy_manual(input, target):
    prob = softmax(input)
    cross_entropy_loss = -torch.mean(torch.log(prob[range(target.shape[0]), target]))
    return cross_entropy_loss
```

[Fig 3] Cross entropy loss function

앞서 말했듯이 cross entropy loss function 을 사용하며 $L(x) = -\Sigma_{i=1}^{10} y_i \log \frac{exp(z_i)}{\Sigma_{j=1}^{10} \exp(z_j)} (z_i = \mathbf{x}^T \mathbf{w}_i + b_i)$로 정의된다. 이에 대해서 **[Fig 3]**과 같이 식을 표현할 수 있지만 nn module 내 nn.CrossEntropyLoss() 함수를 이용하여 진행한다.

SGD optimizer 를 이용해 최적의 weight 와 bias 를 구하고, 구한 parameter 들을 이용해 training MNIST set 과 test MNIST set 을 분류한 결과, 아래와 **[Fig 4]**와 같다.

```
Epoch:   1 | Iter: 390 | Average Loss: 1.354 | Elapsed Time:    8.0 s
Epoch:   1 | Validation Accuracy: 83.000 %

Epoch:   2 | Iter: 390 | Average Loss: 0.763 | Elapsed Time:   14.4 s
Epoch:   2 | Validation Accuracy: 85.120 %
Epoch:  19 | Iter: 390 | Average Loss: 0.368 | Elapsed Time:  123.0 s
Epoch:  19 | Validation Accuracy: 90.200 %

Epoch:  20 | Iter: 390 | Average Loss: 0.365 | Elapsed Time:  129.2 s
Epoch:  20 | Validation Accuracy: 90.290 %
```

```python
# Test the model
total = 0
correct = 0

model.eval()
with torch.no_grad():                                   # No gradient computation
    for data in dataloaders['test']:
        x, y_true = data
        x_flat = x.reshape(-1, width * height).to(device)
        y_true = y_true.to(device)
        y_pred = model(x_flat)
        loss = loss_function(input=y_pred, target=y_true)
        y_pred_argmax = torch.max(y_pred, dim=1)        # y_pred_argmax = predicted label,

        total += y_true.size(0)                         # total number of prediction
        correct += (y_pred_argmax.indices == y_true).sum().item()   # correct prediction

# Print test accuracy
print('Test Accuracy: {:5.3f} %'.format(correct / total * 100))
```

```
Test Accuracy: 90.680 %
```

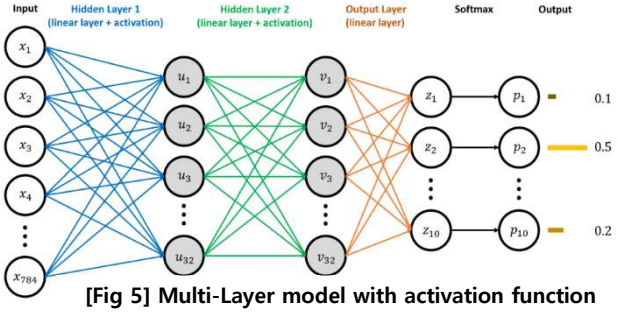
[Fig 4] (Above) Training Accuracy for first two/last two epochs

(Right) Test Accuracy for Single linear layer model

*softmax 함수도 torch 라이브러리 내 softmax 함수를 호출해 사용한다.

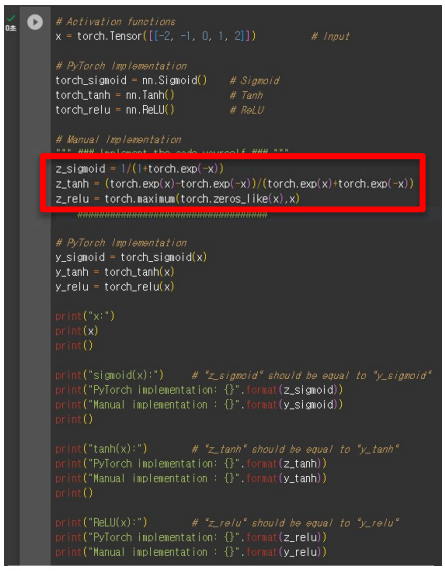
Training accuracy 는 첫 epoch(iteration=390)에서 83%를 시작으로, 마지막 epoch 에서 90.29%의 정확도를 보였으며, test accuracy 는 90.68%의 정확도를 보였다. 보다 더 많은 iteration 을 적용 시, 정확도가 높아질 거라 추측되며 이는 아래 Discussion 에서 논하겠다.

둘째로, **[Fig 5]**과 같이 input 이 linear layer 과 activation function 조합의 hidden layer 2 개와 single linear layer 을 거친 후, 이전 model 처럼 softmax 를 통해 digit 을 추정하는 multi-layer (perceptron) model 을 구현하고자 한다.



[Fig 5] Multi-Layer model with activation function (two Hidden layers + single output layer + softmax)

2) Implement the sigmoid, tanh, and ReLU functions yourself using the mathematical definition of each function.



```
# Activation functions
x = torch.Tensor([[-2, -1, 0, 1, 2]])          # Input

# PyTorch implementation
torch_sigmoid = nn.Sigmoid()     # Sigmoid
torch_tanh = nn.Tanh()           # Tanh
torch_relu = nn.ReLU()           # ReLU

# Manual implementation
""" ### Implement the code yourself ### """
z_sigmoid = 1/(1+torch.exp(-x))
z_tanh = (torch.exp(x)-torch.exp(-x))/(torch.exp(x)+torch.exp(-x))
z_relu = torch.maximum(torch.zeros_like(x),x)
    ##########################################

# PyTorch implementation
y_sigmoid = torch_sigmoid(x)
y_tanh = torch_tanh(x)
y_relu = torch_relu(x)

print("x:")
print(x)
print()

print("sigmoid(x):")     # "z_sigmoid" should be equal to "y_sigmoid"
print("PyTorch implementation: {}".format(z_sigmoid))
print("Manual implementation : {}".format(y_sigmoid))
print()

print("tanh(x):")        # "z_tanh" should be equal to "y_tanh"
print("PyTorch implementation: {}".format(z_tanh))
print("Manual implementation : {}".format(y_tanh))
print()

print("ReLU(x):")       # "z_relu" should be equal to "y_relu"
print("PyTorch implementation: {}".format(z_relu))
print("Manual implementation : {}".format(y_relu))
```

[Fig 6] Code for Exercise 2

```
x:
tensor([[-2., -1.,  0.,  1.,  2.]])

sigmoid(x):
PyTorch implementation: tensor([[0.1192, 0.2689, 0.5000, 0.7311, 0.8808]])
Manual implementation : tensor([[0.1192, 0.2689, 0.5000, 0.7311, 0.8808]])

tanh(x):
PyTorch implementation: tensor([[-0.9640, -0.7616,  0.0000,  0.7616,  0.9640]])
Manual implementation : tensor([[-0.9640, -0.7616,  0.0000,  0.7616,  0.9640]])

ReLU(x):
PyTorch implementation: tensor([[0., 0., 0., 1., 2.]])
Manual implementation : tensor([[0., 0., 0., 1., 2.]])
```

[Fig 7] Result of Exercise 2

<Explanation>

각 activation function 의 수학적 정의는 $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$, $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$, $\text{ReLU}(x) = \max\{0, x\}$ 이다. 이 activation function 들은 non-linear 하게 작동하며 model 이 보다 더 complex 하고 accurate 하게 기능할 수 있게 해준다. 해당 activation function 들은 모두 tensor input 에 대한 연산을 해야 하므로, sigmoid 와 tanh 내 exponential term 은 torch.exp(x)로, ReLU 에 대해서는 x 와 size 가 동일한 0 으로만 구성된 tensor 를 만들고, 그 tensor 와 x 간의 torch.maximum 을 통해 **[Fig 6]**과 같이 activation function 을 구현하였다.

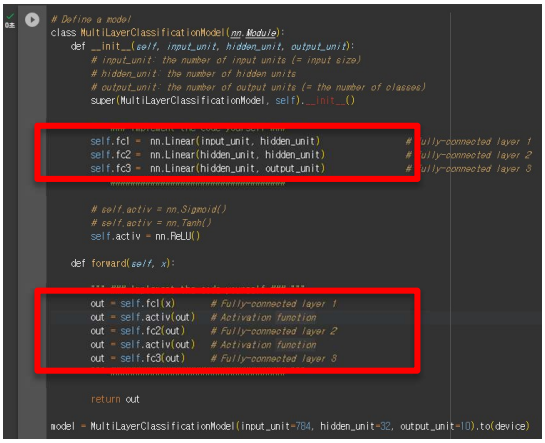
Result. x가 -2,-1,0,1,2 일 때 각 activation function 의 output 은 과 같이 나오며, 위에서 말한 수학적 정의의 결과와 일치한다.

Note. x 자리에 들어가는 것은 각 hidden layer 내 linear layer 의 output tensor 이며, [Fig 5]를 기준으로 모두 size 32 의 tensor 를 activation function 이 input 으로 받게 된다.

3) Implement *fc1*, *fc2*, and *fc3*. Also, implement the forward process of the model. (Use ReLU for activation)

<Explanation>

[Fig 5]와 같이 두 hidden layer 내 linear layer fc1, fc2 와 마지막 single linear layer fc3 를 구현한다. 이에 대한 code 는 **[Fig 8]**과 같다.



```
# Define a model
class MultiLayerClassificationModel(nn.Module):
    def __init__(self, input_unit, hidden_unit, output_unit):
        # input_unit: the number of input units (= input size)
        # hidden_unit: the number of hidden units
        # output_unit: the number of output units (= the number of classes)
        super(MultiLayerClassificationModel, self).__init__()

        ######################################################
        self.fc1 = nn.Linear(input_unit, hidden_unit)          # Fully-connected layer 1
        self.fc2 = nn.Linear(hidden_unit, hidden_unit)         # Fully-connected layer 2
        self.fc3 = nn.Linear(hidden_unit, output_unit)         # Fully-connected layer 3
        ######################################################

        # self.activ = nn.Sigmoid()
        # self.activ = nn.Tanh()
        self.activ = nn.ReLU()

    def forward(self, x):

        ######################################################
        out = self.fc1(x)       # Fully-connected layer 1
        out = self.activ(out)   # Activation function
        out = self.fc2(out)     # Fully-connected layer 2
        out = self.activ(out)   # Activation function
        out = self.fc3(out)     # Fully-connected layer 3
        ######################################################

        return out

model = MultiLayerClassificationModel(input_unit=784, hidden_unit=32, output_unit=10).to(device)
```

[Fig 8] Code for Exercise 3

눈 여겨 볼 점은, fc1 의 경우 (input_size, output_size) = (784, 32), fc2 의 경우 (32,32)의 Linear layer 를 가지고 있으며 이전 model 과 달리 hidden_unit 이라는 새로운 parameter 을 가지고 있다. fc1 적용 후 ReLU, fc2 적용 후 ReLU activation function 이 적용되도록 forward process 를 그림과 같이 구현하였고, 마지막엔 activation 없이 단일 fc3 layer 만을 적용시켜 **[Fig 5]**의 multi-layer model 을 구현하였다.

동일한 loss function(cross entropy)와 optimizer(SGD)을 이용해 해당 model 을 training 및 test(이전 model 과 똑같은 iteration, epoch 를 적용)하면 **[Fig 9]**와 같은 result 을 얻으며, 이전 linear classifier 보다 조금 더 높은 test accuracy 를 보임을 알 수 있다 (92.46% > 90.68%).

```
Epoch:   1 | Iter: 390 | Average Loss: 2.225 | Elapsed Time:    5.3 s
Epoch:   1 | Validation Accuracy: 55.490 %

Epoch:   2 | Iter: 390 | Average Loss: 1.649 | Elapsed Time:   12.3 s
Epoch:   2 | Validation Accuracy: 72.100 %

Epoch:  19 | Iter: 390 | Average Loss: 0.279 | Elapsed Time:  127.0 s
Epoch:  19 | Validation Accuracy: 92.020 %

Epoch:  20 | Iter: 390 | Average Loss: 0.274 | Elapsed Time:  134.0 s
Epoch:  20 | Validation Accuracy: 92.270 %
```

```
# Test the model
total = 0
correct = 0

model.eval()
with torch.no_grad():                                   # No gradient computation
    for data in dataloaders['test']:
        x, y_true = data
        x_flat = x.reshape(-1, width * height).to(device)
        y_true = y_true.to(device)
        y_pred = model(x_flat)
        loss = loss_function(input=y_pred, target=y_true)
        y_pred_argmax = torch.max(y_pred, dim=1)        # y_pred_argmax = predicted label

        total += y_true.size(0)                                 # total number of pred
        correct += (y_pred_argmax.indices == y_true).sum().item()   # correct prediction

# Print test accuracy
print('Test Accuracy: {:5.3f} %'.format(correct / total * 100))
```

```
Test Accuracy: 92.460 %
```

[Fig 9] (Above) Training Accuracy for first two/last two epochs

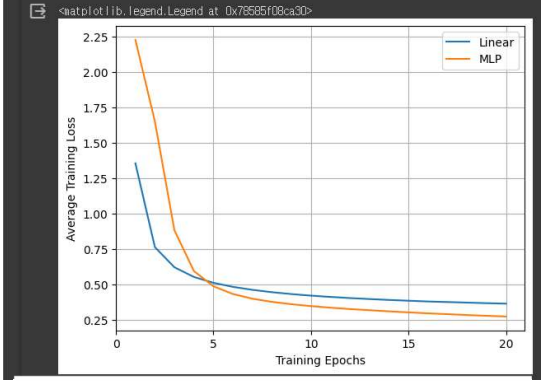
(Right) Test Accuracy for MLP model

● Discussion

1) Compare linear classifier and MLP classifier

```
# Plot training loss for each epoch
epoch_list = [i+1 for i in range(20)]
plt.plot(epoch_list, average_loss_list_linear, label='Linear')
plt.plot(epoch_list, average_loss_list_mlp, label='MLP')
plt.xlabel('Training Epochs')
plt.ylabel('Average Training Loss')
plt.xticks([0, 5, 10 ,15, 20])
plt.grid(visible=True)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x78585f08ca30>
```



[Fig 10] Average Training Loss Graph between Linear(Blue) and MLP(Orange) classifier

[Fig 10]은 위 Exercise 에서 구현한 Linear model 과 MLP model 의 training 중 매 epoch 마다 생기는 training loss 를 계산하는 code 및 결과 graph 이다.

Epoch 5 를 기점으로 그 전까지는 Linear classifier 의 training loss 평균이 더 낮다. 초반 training 에서 MLP classifier 는 linear model 보다 크게 높은 loss 를 보였으나, epoch 5 이후부터는 MLP classifier 가 마지막 epoch 까지 더 낮은 training loss 평균을 보인다. 이와 더불어, 이전에 언급했듯이 **MLP classifier 가 test 에서 더 높은 test accuracy** 를 보였는데, 이는 MLP classifier 가 activation function 과 함께 더 많은 layer 를 소지하면서 보다 더 precise 한 분류를 가능하게 만들었다고 볼 수 있다. 추가로, Linear model 에서 보다 MLP model 에서는, Linear 하지 않은 data distribution 에 대해서는 보다 더 높은 accuracy 를 기대할 수 있다.

2) More Iterations make higher accuracy.

```
Epoch:  19 | Iter: 390 | Average Loss: 0.091 | Elapsed Time:  123.3 s
Epoch:  19 | Validation Accuracy: 96.150 %

Epoch:  20 | Iter: 390 | Average Loss: 0.090 | Elapsed Time:  130.1 s
Epoch:  20 | Validation Accuracy: 96.220 %

Test Accuracy: 96.100 %
```

[Fig 11] Additional training MLP classifier & accuracy

해당 Exercise 에서는 1 cycle 의 training 당 Epoch 를 20 (각 epoch 당 390 회 가량의 iterations)으로 설정하였다. MLP classifier 에서 그 training 을 추가로 3 cycles 를 더 하게 되면 model 이 얼마나 정교해는 지 확인하고자 한다.

그 결과, **[Fig 11]**과 같이 average training loss 도 0.09 로 줄어들었으며, training accuracy 역시 96% 이상의 높은 수치를 보이고 있음을 알 수 있다. 이것이 단지 training set 에만 overfit 된 model 인 지 아닌 지 확인을 위해 test accuracy 역시 확인해본 결과, test accuracy 또한 96% 이상의 높은 수치를 보였다. 결론적으로, 3 회의 추가 training 을 통해 더 정교한 MLP classifier 가 구현되었음을 알 수 있다. 더불어, 만일 과하게 training cycle 을 돌리게 된다면, training accuracy 는 계속 높아지지만 test accuracy 는 낮아지게 되는 overfitting 에 도달할 수 있어 그 적절한 training process 를 찾는 것이 중요할 것이다.