HW 5

2020-14281 이서용

#1

아래와 같이 짜면 된다.

```
mlp_backprop.py

import torch
from torch import nn

def sigma(x):
    return torch.sigmoid(x)
def sigma_prime(x):
    return sigma(x)*(1-sigma(x))


torch.manual_seed(0)
L = 6
X_data = torch.rand(4, 1)
Y_data = torch.rand(1, 1)

A_list,b_list = [],[]
for _ in range(L-1):
    A_list.append(torch.rand(4, 4))
    b_list.append(torch.rand(4, 1))
A_list.append(torch.rand(1, 4))
b_list.append(torch.rand(1, 1))
```

```
# Option 3: implement backprop yourself
y_list = [X_data]
y = X_data
for ell in range(L):
    S = sigma if ell<L-1 else lambda x: x
    y = S(A_list[ell]@y+b_list[ell])
    y_list.append(y)


dA_list = []
db_list = []
dy = y-Y_data  # dloss/dy_L
for ell in reversed(range(L)):
    S = sigma_prime if ell<L-1 else lambda x: torch.ones(x.shape)
    A, b, y = A_list[ell], b_list[ell], y_list[ell]

    db = dy @ torch.diagflat(S(b+A@y))    # dloss/db_ell
    dA = torch.diagflat(S(b+A@y))@dy.T@y.T    # dloss/dA_ell
    dy = dy @ torch.diagflat(S(b+A@y))@A    # dloss/dy_{ell-1}

    dA_list.insert(0, dA)
    db_list.insert(0, db)

print(dA_list[0])
```

Forward path를 거치면서 y_i 들을 계산하고 backward pass 에서 이를 사용해 계산한다. 마지막 layer에는 activation function이 없다는 사실을 고려해주면서 HW 4의 6번 결과를 그대로 적어주면 된다. 결과는 아래와 같다.

```
In [10]: runfile('C:/Users/sylee/OneDrive/바탕 화면/심심개/HW5/
mlp_backprop.py', wdir='C:/Users/sylee/OneDrive/바탕 화면/심심개/HW5')
pytorch autograd
tensor([[2.3943e-05, 3.7064e-05, 4.2687e-06, 6.3700e-06],
        [3.4104e-05, 5.2794e-05, 6.0804e-06, 9.0735e-06],
        [2.4438e-05, 3.7831e-05, 4.3571e-06, 6.5019e-06],
        [2.0187e-05, 3.1250e-05, 3.5991e-06, 5.3707e-06]])

In [11]: runfile('C:/Users/sylee/OneDrive/바탕 화면/심심개/HW5/
mlp_backprop.py', wdir='C:/Users/sylee/OneDrive/바탕 화면/심심개/HW5')
my implementation
tensor([[2.3943e-05, 3.7064e-05, 4.2687e-06, 6.3700e-06],
        [3.4104e-05, 5.2794e-05, 6.0804e-06, 9.0735e-06],
        [2.4438e-05, 3.7831e-05, 4.3571e-06, 6.5019e-06],
        [2.0187e-05, 3.1250e-05, 3.5991e-06, 5.3707e-06]])
```

Pytorch autograd를 사용하나 직접 implement하나 같은 결과를 얻게 된다.

#2.

$$\frac{\partial y_L}{\partial b_i} = \frac{\partial y_L}{\partial y_{L-1}} \cdots \frac{\partial y_{i+1}}{\partial y_i} \cdot \frac{\partial y_i}{\partial b_i} = A_L \cdot \mathrm{diag}(\sigma'(\tilde{y}_{L-1})) A_{L-1}\, \mathrm{diag}(\sigma'(\tilde{y}_{L-2})) A_{L-2} \cdots \mathrm{diag}(\sigma'(\tilde{y}_{i+1})) A_{i+1} \frac{\partial y_i}{\partial b_i}$$

$$= (A_L)\left(\mathrm{diag}(\sigma'(\tilde{y}_{L-1})) A_{L-1}\right)\left(\mathrm{diag}(\sigma'(\tilde{y}_{L-2})) A_{L-2}\right) \cdots \left(\mathrm{diag}(\sigma'(\tilde{y}_{i+1})) A_{i+1}\right)\left(\mathrm{diag}(\sigma'(\tilde{y}_i))\right)$$

$\sigma'(x) = \sigma(x)(1-\sigma(x))$, $0 \le \sigma'(x) \le \frac{1}{4}$.

$\therefore$ If A_j $j \in \{l+1 \dots L\}$ is small, $\frac{\partial y_L}{\partial b_i}$ is a product of $\mathrm{diag}(\sigma'(\tilde{y}_k))$, A_k, and A_j which is a product of ("not too large"), ("not too large"), and ("small") matrices.

This means $\frac{\partial y_L}{\partial b_i}$ is "small.

Also, if $\tilde{y}_j$ has large absolute value, $\sigma'(\tilde{y}_j)$ becomes "small". Therefore, the matrix $\mathrm{diag}\,\sigma'(\tilde{y}_j)$ is small, and the multiplication of matrices containing $\mathrm{diag}\,\sigma'(\tilde{y}_j)$ also becomes "small".

$$\frac{\partial y_L}{\partial A_i} = \mathrm{diag}(\sigma'(\tilde{y}_i)) \cdot \left(\frac{\partial y_L}{\partial y_{L-1}} \cdots \frac{\partial y_{i+1}}{\partial y_i}\right)^T y_{i-1}^T$$

$$= \mathrm{diag}(\sigma'(\tilde{y}_i)) \cdot (A_L)\left(\mathrm{diag}(\sigma'(\tilde{y}_{L-1})) A_{L-1}\right) \cdots \left(\mathrm{diag}(\sigma'(\tilde{y}_{i+1})) A_{i+1}\right) y_{i-1}^T$$

Similarly, if A_j $j \in \{l+1 \dots L\}$ is small, or if $\tilde{y}_j$ $j \in \{l+1 \dots L\}$ has large absolute value (thus $\mathrm{diag}(\sigma'(\tilde{y}_j))$ become small), $\frac{\partial y_L}{\partial A_i}$ is a product of "small" and "not too large" matrices. Therefore, $\frac{\partial y_L}{\partial A_i}$ becomes "small".

#3. $\theta^0, g^0, g^1, \dots$ given. Let's show that θ^i produced by Form I and II is equivalent.

we use mathematical induction.

<i> Case where $n=1$

From form I, $\theta^1 = \theta^0 - \alpha g^0 + \beta(\theta^0 - \theta^{-1}) = \underline{\theta^0 - \alpha g^0}$

From Form II, $\theta^1 = \theta^0 - \alpha v^1$, $v^1 = g^0 + \beta v^0 = g^0$ $\Rightarrow \theta^1 = \underline{\theta^0 - \alpha g^0}$

$\Rightarrow$ θ^1 obtained for the two forms are equivalent.

<ii> Assume, θ^i obtained from Form 1 (θ^i) and from Form 2 ($\bar{\theta}^i$) is equivalent. ($\forall i \le n$)

Then, From Form I, $\theta^{n+1} = \theta^n - \alpha g^n + \beta(\theta^n - \theta^{n-1})$

From Form II,

$$\left.\begin{aligned} v^{n+1} &= g^n + \beta v^n \\ \bar{\theta}^{n+1} &= \bar{\theta}^n - \alpha v^{n+1} \end{aligned}\right\} \Rightarrow \bar{\theta}^{n+1} = \bar{\theta}^n - \alpha g^n - \alpha\beta v^n.$$

We also know that $\bar{\theta}^n = \bar{\theta}^{n-1} - \alpha v^n = \theta^n$ from assumption

$$\begin{aligned} \therefore\ \bar{\theta}^{n+1} &= \bar{\theta}^n - \alpha g^n - \alpha v^n \beta = \bar{\theta}^n - \alpha g^n + \beta(\bar{\theta}^n - \bar{\theta}^{n-1}) \\ &= \theta^n - \alpha g^n + \beta(\theta^n - \theta^{n-1}) \\ &= \theta^{n+1}. \end{aligned}$$

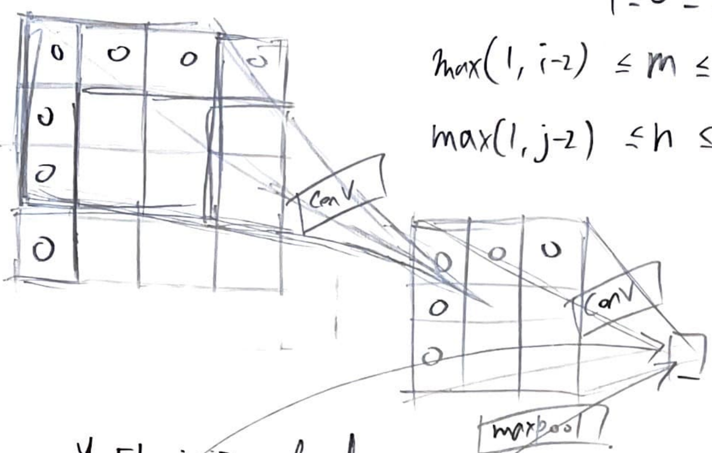
By <i>, <ii>, and mathematical induction, we proved that Form I and II produces same θ^i ($\forall i \in \mathbb{N}$) sequence.

#4

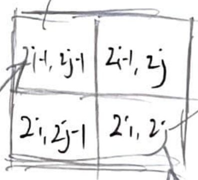
* 모든 index는 1에서 시작한다고 생각
(zero padding 제외하고..)

$y_1[k, i, j]$ depends on. $X[c, m, n]$

$$1 \le c \le 3$$
$$\max(1, i-2) \le m \le \min(224, i+2)$$
$$\max(1, j-2) \le n \le \min(224, j+2)$$



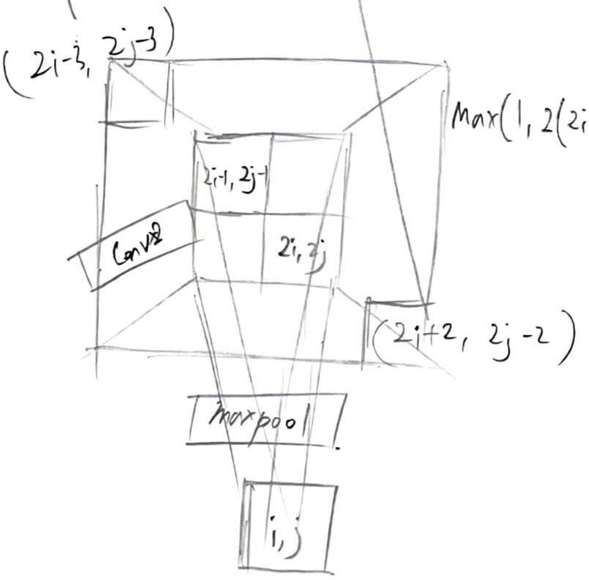
$y_2[k, i, j]$ depends on $X[c, m, n]$



$$1 \le c \le 3$$
$$\max(1, 2i-3) \le m \le \min(224, 2i+2)$$
$$\max(1, 2j-3) \le n \le \min(224, 2j+2)$$

$y_3[k, i, j]$ depends on $X[c, m, n]$

$$1 \le c \le 3$$
$$\max(1, 2(2i-3)-3) \le m \le \min(224, 2\times(2i+2)+2)$$
$$\Rightarrow \max(1, 4i-9) \le m \le \min(224, 4i+6)$$
$$\max(1, 4j-9) \le n \le \min(224, 4j+6)$$



#5 ① Naive inception

<i> Trainable parameters :

$$128 \times (256 \times 1 \times 1 + 1) + 192 \times (256 \times 3 \times 3 + 1) + 96 \times (256 \times 5 \times 5 + 1) \text{ 개.}$$

$$= 1089952 \text{ 개}$$

<ii> 연산.

$$\text{addition} = (256) \times 128 \times 32^2 + (9 \times 256) \times 192 \times 32^2 + (25 \times 256) \times 96 \times 32^2$$

Multiplication : addition 과 동일.

$$= 1115684864 \text{ 번}$$

$$\text{activation fn} : 128 \times 32^2 + 192 \times 32^2 + 96 \times 32^2 = 425984 \text{ 번.}$$

② Inception with 1x1 bottleneck convolution

<i> Trainable parameters :

$$128 \times (256 \times 1 \times 1 + 1) + 64 \times (256 \times 1 \times 1 + 1) + 192 \times (64 \times 3 \times 3 + 1)$$

$$+ 64 \times (256 \times 1 \times 1 + 1) + 96 \times (64 \times 5 \times 5 + 1)$$

$$+ 64 \times (256 \times 1 \times 1 + 1) = 346720 \text{ 개}$$

<ii> 연산

$$\text{Addition} : 256 \times 128 \times 32^2 + 256 \times 64 \times 32^2 + 9 \times 64 \times 192 \times 32^2$$

$$+ 256 \times 64 \times 32^2 + 25 \times 64 \times 96 \times 32^2$$

$$+ 256 \times 64 \times 32^2 = 354418688 \text{ 번}$$

Multiplication : addition 과 동일.

$$\text{Activation fn} : 128 \times 32^2 + 64 \times 32^2 + 192 \times 32^2 + 64 \times 32^2 + 96 \times 32^2 + 64 \times 32^2$$

$$= 622592 \text{ 번}$$

1x1 Bottleneck convolution 을 사용한 경우, trainable parameter 와 addition/multiplication 이 크게 줄어드는 것을 볼 수 있다.

In [6]:

```python
import torch
import torch.nn as nn
from torch.optim import Optimizer
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import transforms

import numpy as np
import time
import matplotlib.pyplot as plt
# Make sure to use only 10% of the available MNIST data.
# Otherwise, experiment will take quite long (around 90 minutes).

# Prepare data
trainset = datasets.MNIST(root = './mnist_data/', train=True, transform = transforms.ToTensor(),download=True)
# random label
n = len(trainset)
trainset.targets = torch.randint(0,10,(n,) )
# use only 10%
idx = np.random.permutation(np.arange(n))[:int(n/10)]
trainset.data = trainset.data[idx]
trainset.targets = trainset.targets[idx]

# (Modified version of AlexNet)
class AlexNet(nn.Module):
    def __init__(self, num_class=10):
        super(AlexNet, self).__init__()

        self.conv_layer1 = nn.Sequential(
            nn.Conv2d(1, 96, kernel_size=4),
            nn.ReLU(inplace=True),
            nn.Conv2d(96, 96, kernel_size=3),
            nn.ReLU(inplace=True)
        )
        self.conv_layer2 = nn.Sequential(
            nn.Conv2d(96, 256, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )
        self.conv_layer3 = nn.Sequential(
            nn.Conv2d(256, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )

        self.fc_layer1 = nn.Sequential(
            nn.Dropout(),
            nn.Linear(6400, 800),
            nn.ReLU(inplace=True),
            nn.Linear(800, 10)
        )

    def forward(self, x):
        output = self.conv_layer1(x)
        output = self.conv_layer2(output)
        output = self.conv_layer3(output)
        output = torch.flatten(output, 1)
        output = self.fc_layer1(output)
        return output


learning_rate = 0.1
batch_size = 64
epochs = 150

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = AlexNet().to(device)
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

train_loss = []
train_accuracy = []
train_loader = DataLoader(dataset = trainset, batch_size = batch_size, shuffle = True)


tick = time.time()
for epoch in range(epochs):
    print(f"\nEpoch {epoch + 1} / {epochs}")
    total =0
    correct =0
    lossval =0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        output = model(images)
        loss = loss_function(output, labels)
        loss.backward()
        optimizer.step()

        lossval += loss.item()
        total+=1
        pred = output.max(1, keepdim=True)[1]
        correct += pred.eq(labels.view_as(pred)).sum().item()

    train_loss.append(lossval/total)
    train_accuracy.append(correct/total/batch_size)
    print(train_accuracy[-1])

tock = time.time()
print(f"Total training time: {tock - tick}")

#plot
xaxis = [i for i in range(epochs)]
fig = plt.figure()
plt.plot(xaxis, train_loss, label='train loss')
plt.plot(xaxis, train_accuracy, label='train accuracy')
fig.legend()
plt.show()
```

```
Epoch 1 / 150
0.10056515957446809

Epoch 2 / 150
0.10887632978723404

Epoch 3 / 150
0.10887632978723404

Epoch 4 / 150
0.10887632978723404

Epoch 5 / 150
0.10887632978723404

Epoch 6 / 150
0.10638297872340426

Epoch 7 / 150
0.10887632978723404

Epoch 8 / 150
0.10887632978723404

Epoch 9 / 150
0.10887632978723404

Epoch 10 / 150
0.10887632978723404

Epoch 11 / 150
0.10887632978723404

Epoch 12 / 150
0.10887632978723404

Epoch 13 / 150
0.10887632978723404

Epoch 14 / 150
0.10887632978723404

Epoch 15 / 150
0.10605053191489362

Epoch 16 / 150
0.10887632978723404

Epoch 17 / 150
0.10887632978723404

Epoch 18 / 150
0.10887632978723404

Epoch 19 / 150
0.10588430851063829

Epoch 20 / 150
0.10887632978723404

Epoch 21 / 150
0.10887632978723404

Epoch 22 / 150
0.10887632978723404

Epoch 23 / 150
0.10887632978723404

Epoch 24 / 150
0.10887632978723404

Epoch 25 / 150
0.10787898936170212

Epoch 26 / 150
0.1077127659574468

Epoch 27 / 150
0.10887632978723404

Epoch 28 / 150
0.10887632978723404

Epoch 29 / 150
0.10887632978723404

Epoch 30 / 150
0.10887632978723404

Epoch 31 / 150
0.10887632978723404

Epoch 32 / 150
0.10887632978723404

Epoch 33 / 150
0.10887632978723404

Epoch 34 / 150
0.10887632978723404

Epoch 35 / 150
0.10887632978723404

Epoch 36 / 150
0.10871010638297872

Epoch 37 / 150
0.10887632978723404

Epoch 38 / 150
0.10821143617021277

Epoch 39 / 150
0.10887632978723404

Epoch 40 / 150
0.10887632978723404

Epoch 41 / 150
0.10871010638297872

Epoch 42 / 150
0.10887632978723404

Epoch 43 / 150
0.10787898936170212

Epoch 44 / 150
0.10970744680851063

Epoch 45 / 150
0.10887632978723404

Epoch 46 / 150
0.10904255319148937

Epoch 47 / 150
0.10871010638297872

Epoch 48 / 150
0.109375

Epoch 49 / 150
0.10970744680851063

Epoch 50 / 150
0.109375

Epoch 51 / 150
0.1085438829787234

Epoch 52 / 150
0.11236702127659574

Epoch 53 / 150
0.10837765957446809

Epoch 54 / 150
0.11037234042553191

Epoch 55 / 150
0.11419547872340426

Epoch 56 / 150
0.11868351063829788

Epoch 57 / 150
0.11785239361702128

Epoch 58 / 150
0.11319813829787234

Epoch 59 / 150
0.11419547872340426

Epoch 60 / 150
0.1180186170212766

Epoch 61 / 150
0.12300531914893617

Epoch 62 / 150
0.12483377659574468

Epoch 63 / 150
0.11951462765957446

Epoch 64 / 150
0.11951462765957446

Epoch 65 / 150
0.12982047872340424

Epoch 66 / 150
0.12566489361702127

Epoch 67 / 150
0.12516622340425532

Epoch 68 / 150
0.12566489361702127

Epoch 69 / 150
0.1363031914893617

Epoch 70 / 150
0.13696808510638298

Epoch 71 / 150
0.14012632978723405

Epoch 72 / 150
0.14045877659574468

Epoch 73 / 150
0.14378324468085107

Epoch 74 / 150
0.1519281914893617

Epoch 75 / 150
0.14611037234042554

Epoch 76 / 150
0.14627659574468085

Epoch 77 / 150
0.15226063829787234

Epoch 78 / 150
0.1612367021276596

Epoch 79 / 150
0.17054521276595744

Epoch 80 / 150
0.16705452127659576

Epoch 81 / 150
0.17220744680851063

Epoch 82 / 150
0.1825132978723404

Epoch 83 / 150
0.1918218085106383

Epoch 84 / 150
0.19664228723404256

Epoch 85 / 150
0.2066156914893617

Epoch 86 / 150
0.21742021276595744

Epoch 87 / 150
0.23470744680851063

Epoch 88 / 150
0.25615026595744683

Epoch 89 / 150
0.2677859042553192

Epoch 90 / 150
0.3020279255319149

Epoch 91 / 150
0.3249667553191489

Epoch 92 / 150
0.3533909574468085

Epoch 93 / 150
0.38148271276595747

Epoch 94 / 150
0.421875

Epoch 95 / 150
0.45478723404255317

Epoch 96 / 150
0.5094747340425532

Epoch 97 / 150
0.5477061170212766

Epoch 98 / 150
0.5861037234042553

Epoch 99 / 150
0.6377992021276596

Epoch 100 / 150
0.667220744680851

Epoch 101 / 150
0.6894946808510638

Epoch 102 / 150
0.7292220744680851

Epoch 103 / 150
0.7672872340425532

Epoch 104 / 150
0.7903922872340425

Epoch 105 / 150
0.8016954787234043

Epoch 106 / 150
0.8236369680851063

Epoch 107 / 150
0.8342752659574468

Epoch 108 / 150
0.8459109042553191

Epoch 109 / 150
0.8558843085106383

Epoch 110 / 150
0.866688829787234

Epoch 111 / 150
0.8824800531914894

Epoch 112 / 150
0.8929521276595744

Epoch 113 / 150
0.8956117021276596

Epoch 114 / 150
0.9084109042553191

Epoch 115 / 150
0.9114029255319149

Epoch 116 / 150
0.9208776595744681

Epoch 117 / 150
0.9243683510638298

Epoch 118 / 150
0.925531914893617

Epoch 119 / 150
0.937998670212766

Epoch 120 / 150
0.937998670212766

Epoch 121 / 150
0.9403257978723404

Epoch 122 / 150
0.9429853723404256

Epoch 123 / 150
0.9436502659574468

Epoch 124 / 150
0.9479720744680851

Epoch 125 / 150
0.9446476063829787

Epoch 126 / 150
0.949966755319149

Epoch 127 / 150
0.9549534574468085

Epoch 128 / 150
0.9574468085106383

Epoch 129 / 150
0.9479720744680851

Epoch 130 / 150
0.9627659574468085

Epoch 131 / 150
0.9609375

Epoch 132 / 150
0.9649268617021277

Epoch 133 / 150
0.9660904255319149

Epoch 134 / 150
0.9679188829787234

Epoch 135 / 150
0.9705784574468085

Epoch 136 / 150
0.9722406914893617

Epoch 137 / 150
0.9700797872340425

Epoch 138 / 150
0.9765625

Epoch 139 / 150
0.9737367021276596

Epoch 140 / 150
0.9705784574468085

Epoch 141 / 150
0.973404255319149

Epoch 142 / 150
0.9755651595744681

Epoch 143 / 150
0.9755651595744681

Epoch 144 / 150
0.9762300531914894

Epoch 145 / 150
0.9744015957446809

Epoch 146 / 150
0.9765625

Epoch 147 / 150
0.9765625

Epoch 148 / 150
0.9747340425531915

Epoch 149 / 150
0.9780585106382979

Epoch 150 / 150
0.9780585106382979
Total training time: 528.9760839939117
```