MathDNN Homework 5

Department of Computer Science and Engineering
2021-16988 Jaewan Park

Problem 2

When $i \neq L$, we can calculate the following.

$$\begin{aligned}
\frac{\partial y_L}{\partial b_i} &= \frac{\partial y_L}{\partial y_i}\frac{\partial y_i}{\partial b_i} \\
&= \frac{\partial y_L}{\partial y_{L-1}}\frac{\partial y_{L-1}}{\partial y_{L-2}}\cdots\frac{\partial y_{i+1}}{\partial y_i}\frac{\partial y_i}{\partial b_i} \\
&= A_L(\mathrm{diag}(\sigma'(\tilde{y_{L-1}}))A_{L-1})\cdots(\mathrm{diag}(\sigma'(\tilde{y_{i+1}}))A_{i+1})\,\mathrm{diag}(\sigma'(\tilde{y_i})) \\
\frac{\partial y_L}{\partial A_i} &= \mathrm{diag}(\sigma'(\tilde{y_i}))\left(\frac{\partial y_L}{\partial y_i}\right)^{\intercal} y_{i-1}^{\intercal} \\
&= \mathrm{diag}(\sigma'(\tilde{y_i}))A_{i+1}^{\intercal}\mathrm{diag}(\sigma'(\tilde{y_{i+1}}))^{\intercal}\cdots A_{L-1}^{\intercal}\mathrm{diag}(\sigma'(\tilde{y_{L-1}}))^{\intercal}A_L^{\intercal}y_{i-1}^{\intercal}
\end{aligned}$$

First, suppose A_j is small for some $j \in \{l+1, \cdots, L\}$. Then for any $i \in \{1, \cdots, l\}$, $i \neq L$ and A_j must exist among $A_{i+1}, \cdots, A_L$. Therefore one small matrix exists in the chain of matrix multiplication in the above calculations. Also, since $0 \leq \sigma'(z) \leq 0.25$ for all z where σ is the sigmoid activation function, all outputs of the function are relatively 'not too large' numbers and consequently $\mathrm{diag}(\sigma'(\tilde{y_k}))$ are all not too large matrices. Therefore the above calculations only contain not too large matrices and at least one small matrix, thus the results become small.

Next, suppose $\tilde{y_j}$ has large absolute value for some $j \in \{l+1, \cdots, L-1\}$. Then for any $i \in \{1, \cdots, l\}$, $i \neq L$ and at $\tilde{y_j}$ must exist among $\tilde{y_{i+1}}, \cdots, \tilde{y_{L-1}}$. Since $\sigma'(z) \to 0$ as $z \to \pm\infty$ where σ is the sigmoid activation function, $\mathrm{diag}(\sigma'(\tilde{y_j}))$ is absolutely 'small' as $\tilde{y_j}$ has absolute large value. Other values of A_k or $\mathrm{diag}(\sigma'(\tilde{y_k}))$ are all not too large. Therefore the above calculations only contain not too large matrices and at least one small matrix, thus the results become small.

Problem 3

To prevent confusion, notate the points calculated from Form I θ_{I}^k, and those from Form II θ_{II}^k.

Calculations of θ^1 from the two forms are identical.

$$\begin{aligned}
\theta_{\mathrm{I}}^1 &= \theta^0 - \alpha g^0 + \beta\big(\theta^0 - \theta^0\big) = \theta^0 - \alpha g^0 \\
\theta_{\mathrm{II}}^1 &= \theta^0 - \alpha v^1 = \theta^0 - \alpha\big(g^0 + \beta v^0\big) = \theta^0 - \alpha g^0
\end{aligned}$$

Now suppose calculations of $\theta^0, \cdots, \theta^k$ from the two forms are all identical. Then

$$\begin{aligned}
\theta_{\mathrm{II}}^{k+1} &= \theta_{\mathrm{II}}^k - \alpha v^{k+1} \\
&= \theta_{\mathrm{I}}^k - \alpha\big(g^k + \beta v^k\big) = \theta_{\mathrm{I}}^k - \alpha g^k + \beta\big(-\alpha v^k\big) \\
&= \theta_{\mathrm{I}}^k - \alpha g^k + \beta\big(\theta_{\mathrm{II}}^k - \theta_{\mathrm{II}}^{k-1}\big) \\
&= \theta_{\mathrm{I}}^k - \alpha g^k + \beta\big(\theta_{\mathrm{I}}^k - \theta_{\mathrm{I}}^{k-1}\big) \\
&= \theta_{\mathrm{I}}^{k+1}.
\end{aligned}$$

Therefore $\theta_{\mathrm{I}}^{k+1} = \theta_{\mathrm{II}}^{k+1}$, and by using mathematical induction, we can claim that Forms I and II produce the same $\theta^1, \theta^2, \cdots$ sequence.

Problem 4

Let the output of the first, third, and fourth convolutional layers y_4, y_5, y_6. Then $y_4[k,i,j]$ depends on $X[:, i-1 : i+1, j-1 : j+1]$, $y_1[k,i,j]$ depends on $y_4[:, i-1 : i+1, j-1 : j+1]$, $y_2[k,i,j]$ depends on $y_1[:, 2i-1 : 2i, 2j-1 : 2j]$, $y_5[k,i,j]$ depends on $y_2[:, i-1 : i+1, j-1 : j+1]$, $y_6[k,i,j]$ depends on $y_5[:, i-1 : i+1, j-1 : j+1]$, and $y_3[k,i,j]$ depends on $y_6[:, 2i-1 : 2i, 2j-1 : 2j]$.

As a result, $y_1[k,i,j]$ depends on $X[:, i-2 : i+2, j-2 : j+2]$, $y_2[k,i,j]$ depends on $X[:, 2i-3 : 2i+2; 2j-3 : 2j+2]$, and $y_3[k,i,j]$ depends on $X[:, 4i-9 : 4i+6, 4j-9 : 4j+6]$.

Problem 5

The number of trainable parameters between two layers can be calculated as $C_{\text{out}} \times (C_{\text{in}} \times F \times F + 1)$, where the addition of 1 is made due to the bias. The number of additions and multiplications are equal, both calculated as $C_{\text{out}} \times m_{\text{out}} \times n_{\text{out}} \times C_{\text{in}} \times F \times F$, where $m_{\text{out}} \times n_{\text{out}}$ is the output dimension. Additions are made between multiplied values, so the number of it should be one less than that of multiplications, but adding the bias makes them equal. The number of activation function evaluations can be calculated as $C_{\text{out}} \times m_{\text{out}} \times n_{\text{out}}$, since the function is applies after the convolution. Therefore the counts for each models are the following.

The First Model
Number of Trainable Parameters : $192 \times (256 \times 3 \times 3 + 1) + 96 \times (256 \times 5 \times 5 + 1) = 1057056$
Number of Additions : $192 \times 32 \times 32 \times 256 \times 3 \times 3 + 96 \times 32 \times 32 \times 256 \times 3 \times 3 = 679477248$
Number of Multiplications : $192 \times 32 \times 32 \times 256 \times 3 \times 3 + 96 \times 32 \times 32 \times 256 \times 3 \times 3 = 679477248$
Number of Activation Function Evaluations : $192 \times 32 \times 32 + 96 \times 32 \times 32 = 294912$

The Second Model
Number of Trainable Parameters : $64 \times (256 \times 1 \times 1 + 1) + 192 \times (64 \times 3 \times 3 + 1) + 64 \times (256 \times 1 \times 1 + 1) + 96 \times (64 \times 5 \times 5 + 1) = 297376$

Number of Additions : $64 \times 32 \times 32 \times 256 \times 1 \times 1 + 192 \times 32 \times 32 \times 64 \times 3 \times 3 + 64 \times 32 \times 32 \times 256 \times 1 \times 1 + 96 \times 32 \times 32 \times 64 \times 5 \times 5 = 304087040$
Number of Multiplications : $64 \times 32 \times 32 \times 256 \times 1 \times 1 + 192 \times 32 \times 32 \times 64 \times 3 \times 3 + 64 \times 32 \times 32 \times 256 \times 1 \times 1 + 96 \times 32 \times 32 \times 64 \times 5 \times 5 = 304087040$
Number of Activation Function Evaluations : $64 \times 32 \times 32 + 192 \times 32 \times 32 + 64 \times 32 \times 32 + 96 \times 32 \times 32 = 425984$

The second model has advantage in number of trainable parameters, while the number of additions, multiplications, and activation function evaluations are larger.

Problem 1

[1]:
```
import torch
from torch import nn
```

[2]:
```
def sigma(x):
    return torch.sigmoid(x)
def sigma_prime(x):
    return sigma(x)*(1-sigma(x))
```

[3]:
```
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 1: directly use PyTorch's autograd feature
for A in A_list:
    A.requires_grad = True
for b in b_list:
    b.requires_grad = True

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])

# backward pass in pytorch
loss=torch.square(y-Y_data)/2
loss.backward()

print(A_list[0].grad)
print(b_list[0].grad.T)
```

```
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]])
tensor([[4.8247e-05, 6.8722e-05, 4.9245e-05, 4.0678e-05]])
```

[4]:
```
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):
```

```python
    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 2: construct a NN model and use backprop
class MLP(nn.Module) :
    def __init__(self) :
        super().__init__()
        self.linear = nn.ModuleList([nn.Linear(4,4) for _ in range(L-1)])
        self.linear.append(nn.Linear(4,1))
        for ell in range(L):
            self.linear[ell].weight.data = A_list[ell]
            self.linear[ell].bias.data = b_list[ell].squeeze()

    def forward(self, x) :
        x = x.squeeze()
        for ell in range(L-1):
            x = sigma(self.linear[ell](x))
        x = self.linear[-1](x)
        return x

model = MLP()

loss = torch.square(model(X_data)-Y_data)/2
loss.backward()

print(model.linear[0].weight.grad)
print(model.linear[0].bias.grad)
```

```
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]])
tensor([4.8247e-05, 6.8722e-05, 4.9245e-05, 4.0678e-05])
```

[5]:

```python
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
dyL = torch.tensor([[1.]])  # dy_L/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.diag(S(A@y+b).reshape(-1))                          # dloss/
↪db_ell
    dA = (y_list[-1]-Y_data)*torch.diag(S(A@y+b).reshape(-1))@dyL.T@y.T   # dloss/
↪dA_ell
    dy = dy@torch.diag(S(A@y+b).reshape(-1))@A                        # dloss/
↪dy_{ell-1}
    dyL = dyL@torch.diag(S(A@y+b).reshape(-1))@A                      # dy_L/
↪dy_{ell-1}

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

print(dA_list[0])
print(db_list[0])
```

```
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]])
tensor([[4.8247e-05, 6.8722e-05, 4.9245e-05, 4.0678e-05]])
```

Results from the three methods are identical.

Problem 6

[6]:
```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets
from torchvision.transforms import transforms
import matplotlib.pyplot as plt
import random
```

[7]:
```
# MNIST dataset
mnist_dataset = datasets.MNIST(root='./mnist_data/',
                               train=True,
                               transform=transforms.ToTensor(),
                               download=True)

train_dataset = Subset(mnist_dataset, torch.randperm(60000)[:6000])
train_dataset.targets = torch.randint(10, (len(train_dataset),))
```

[8]:
```python
# (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
```

[9]:
```python
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_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=False)
```

[10]:

```
accuracy_list = []
loss_list = []

import time
tick = time.time()
for epoch in range(epochs):
    print(f"\nEpoch {epoch + 1} / {epochs}")

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

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

        optimizer.step()

    train_loss, correct = 0, 0

    for images, labels in test_loader :
        images, labels = images.to(device), labels.to(device)

        output = model(images)
        train_loss += loss_function(output, labels).item()
        for i in range(len(output)):
            if torch.argmax(output[i]).item() == labels[i].item():
                correct += 1

    train_accuracy = correct / 6000

    accuracy_list.append(train_accuracy)
    loss_list.append(train_loss)

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

```
Epoch 1 / 150

Epoch 2 / 150

Epoch 3 / 150

Epoch 4 / 150

Epoch 5 / 150

Epoch 6 / 150

Epoch 7 / 150

Epoch 8 / 150

Epoch 9 / 150
```

Epoch 10 / 150

Epoch 11 / 150

Epoch 12 / 150

Epoch 13 / 150

Epoch 14 / 150

Epoch 15 / 150

Epoch 16 / 150

Epoch 17 / 150

Epoch 18 / 150

Epoch 19 / 150

Epoch 20 / 150

Epoch 21 / 150

Epoch 22 / 150

Epoch 23 / 150

Epoch 24 / 150

Epoch 25 / 150

Epoch 26 / 150

Epoch 27 / 150

Epoch 28 / 150

Epoch 29 / 150

Epoch 30 / 150

Epoch 31 / 150

Epoch 32 / 150

Epoch 33 / 150

Epoch 34 / 150

Epoch 35 / 150

Epoch 36 / 150

```
Epoch 37 / 150

Epoch 38 / 150

Epoch 39 / 150

Epoch 40 / 150

Epoch 41 / 150

Epoch 42 / 150

Epoch 43 / 150

Epoch 44 / 150

Epoch 45 / 150

Epoch 46 / 150

Epoch 47 / 150

Epoch 48 / 150

Epoch 49 / 150

Epoch 50 / 150

Epoch 51 / 150

Epoch 52 / 150

Epoch 53 / 150

Epoch 54 / 150

Epoch 55 / 150

Epoch 56 / 150

Epoch 57 / 150

Epoch 58 / 150

Epoch 59 / 150

Epoch 60 / 150

Epoch 61 / 150

Epoch 62 / 150

Epoch 63 / 150
```

```
Epoch 64 / 150

Epoch 65 / 150

Epoch 66 / 150

Epoch 67 / 150

Epoch 68 / 150

Epoch 69 / 150

Epoch 70 / 150

Epoch 71 / 150

Epoch 72 / 150

Epoch 73 / 150

Epoch 74 / 150

Epoch 75 / 150

Epoch 76 / 150

Epoch 77 / 150

Epoch 78 / 150

Epoch 79 / 150

Epoch 80 / 150

Epoch 81 / 150

Epoch 82 / 150

Epoch 83 / 150

Epoch 84 / 150

Epoch 85 / 150

Epoch 86 / 150

Epoch 87 / 150

Epoch 88 / 150

Epoch 89 / 150

Epoch 90 / 150
```

```
Epoch 91 / 150

Epoch 92 / 150

Epoch 93 / 150

Epoch 94 / 150

Epoch 95 / 150

Epoch 96 / 150

Epoch 97 / 150

Epoch 98 / 150

Epoch 99 / 150

Epoch 100 / 150

Epoch 101 / 150

Epoch 102 / 150

Epoch 103 / 150

Epoch 104 / 150

Epoch 105 / 150

Epoch 106 / 150

Epoch 107 / 150

Epoch 108 / 150

Epoch 109 / 150

Epoch 110 / 150

Epoch 111 / 150

Epoch 112 / 150

Epoch 113 / 150

Epoch 114 / 150

Epoch 115 / 150

Epoch 116 / 150

Epoch 117 / 150
```

```
Epoch 118 / 150

Epoch 119 / 150

Epoch 120 / 150

Epoch 121 / 150

Epoch 122 / 150

Epoch 123 / 150

Epoch 124 / 150

Epoch 125 / 150

Epoch 126 / 150

Epoch 127 / 150

Epoch 128 / 150

Epoch 129 / 150

Epoch 130 / 150

Epoch 131 / 150

Epoch 132 / 150

Epoch 133 / 150

Epoch 134 / 150

Epoch 135 / 150

Epoch 136 / 150

Epoch 137 / 150

Epoch 138 / 150

Epoch 139 / 150

Epoch 140 / 150

Epoch 141 / 150

Epoch 142 / 150

Epoch 143 / 150

Epoch 144 / 150
```

```
Epoch 145 / 150

Epoch 146 / 150

Epoch 147 / 150

Epoch 148 / 150

Epoch 149 / 150

Epoch 150 / 150
Total training time: 752.191534280777
```

[11]:

```
fig, ax1 = plt.subplots()
ax1.set_xlabel('Epochs')
ax1.set_ylabel('Accuracy', color='tab:red')
ax1.plot(range(epochs), accuracy_list, color='tab:red', label='Train Accuracy')
ax1.tick_params(axis='y', labelcolor='tab:red')
ax1.legend(bbox_to_anchor=(0.8, 1.22), loc="upper left")
ax2 = ax1.twinx()
ax2.set_ylabel('Loss', color='tab:blue')
ax2.plot(range(epochs), loss_list, color='tab:blue', label='Train Loss')
ax2.tick_params(axis ='y', labelcolor='tab:blue')
ax2.legend(bbox_to_anchor=(0.8, 1.13), loc="upper left")
plt.title('Training with Randomized Label')
plt.show()
```