MathDNN Homework 7

Department of Computer Science and Engineering
2021-16988 Jaewan Park

Problem 1

(a) WLOG, assume $x_1 \geq x_2 \geq \cdots \geq x_n$. Then

$$\begin{aligned}\nu_\beta(x) &= \frac{1}{\beta}\log\sum_{i=1}^{n}\exp(\beta x_i) = \frac{1}{\beta}\log\left(\exp(\beta x_1)+\exp(\beta x_2)+\cdots+\exp(\beta x_n)\right)\\ &\leq \frac{1}{\beta}\log\left(n\exp(\beta x_1)\right) = x_1 + \frac{\log n}{\beta}\\ \nu_\beta(x) &= \frac{1}{\beta}\log\sum_{i=1}^{n}\exp(\beta x_i) = \frac{1}{\beta}\log\left(\exp(\beta x_1)+\exp(\beta x_2)+\cdots+\exp(\beta x_n)\right)\\ &\geq \frac{1}{\beta}\log\left(\exp(\beta x_1)\right) = x_1\end{aligned}$$

so $\lim_{\beta\to\infty} x_1 \leq \lim_{\beta\to\infty}\nu_\beta(x) \leq \lim_{\beta\to\infty}\left(x_1+\frac{\log n}{\beta}\right)$, therefore

$$\lim_{\beta\to\infty}\nu_\beta(x) = x_1 = \max\{x_1,\cdots,x_n\}.$$

(b) The partial derivatives of $\nu_1(x)$ are

$$\frac{\partial\nu_1}{\partial x_j} = \frac{\partial}{\partial x_j}\log\sum_{i=1}^{n}\exp(x_i) = \frac{\exp(x_j)}{\sum_{i=1}^{n}\exp(x_i)} = \mu_j.$$

Therefore $\nabla\nu_1 = \mu$.

(c) The partial derivatives of $\nu_\beta(x)$ are

$$\frac{\partial\nu_\beta}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{1}{\beta}\log\sum_{i=1}^{n}\exp(\beta x_i)\right) = \frac{\exp(\beta x_j)}{\sum_{i=1}^{n}\exp(\beta x_i)}.$$

When $\beta\to\infty$, we obtain

$$\begin{aligned}\lim_{\beta\to\infty}\frac{\partial\nu_\beta}{\partial x_j} &= \lim_{\beta\to\infty}\frac{\exp(\beta x_j)}{\sum_{i=1}^{n}\exp(\beta x_i)} = \lim_{\beta\to\infty}\frac{1}{\sum_{i=1}^{n}\exp(\beta x_i-\beta x_j)}\\ &= \begin{cases}1 & (j=i_{\max})\\ 0 & (j\neq i_{\max})\end{cases}.\end{aligned}$$

Therefore $\lim_{\beta\to\infty}\nabla\nu_\beta(x) = \mathbf{e}_{i_{\max}}$.

Problem 2

The number of additions and multiplications in convolutional layers are both $C_{\text{in}} \times F^2 \times C_{\text{out}} \times h_{\text{out}}^2$, where $F \times F$ is the kernel size and $h_{\text{out}} \times h_{\text{out}}$ is the output dimension. The number of additions and multiplications in linear layers are both $n_{\text{in}} \times n_{\text{out}}$, where n_{in} and n_{out} are each input and output vector sizes. Additions are made between multiplied values, so the number of it should be one less than that of multiplications, but the process of adding the bias makes them equal.

Therefore the number of operations needed in the convolutional layers is

$$\begin{aligned}2 \times (3 \times 11^2 \times 64 \times 55^2 + 64 \times 5^2 \times 192 \times 27^2 + 192 \times 3^2 \times 384 \times 13^2 \\ + 384 \times 3^2 \times 256 \times 13^2 + 256 \times 3^2 \times 256 \times 13^2) = 1311133056\end{aligned}$$

and the number of operations needed in the linear layers is

$$2 \times (256 \times 6^2 \times 4096 + 4096 \times 4096 + 4096 \times 1000) = 117243904.$$

Problem 4

Notation For a matrix or vector X, the notation $[X]_{i,j}$ refers to the element of X at that index, and $\{f(i,j)\}_{i,j}$ refers to the matrix of which element at index (i,j) is $f(i,j)$.

(a) It is obvious that

$$\frac{\partial y_L}{\partial y_{L-1}} = \frac{\partial}{\partial y_{L-1}}(A_{w_L} y_{L-1} + b_L \mathbf{1}_{n_L}) = A_{w_L}.$$

For $\ell = 2, \cdots, L-1$, we obtain

$$\begin{aligned}\frac{\partial y_\ell}{\partial y_{\ell-1}} &= \frac{\partial}{\partial y_{\ell-1}} \sigma(A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell}) \\ &= \left\{ \frac{\partial}{\partial [y_{\ell-1}]_j} [\sigma(A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell})]_i \right\}_{i,j} = \left\{ \frac{\partial}{\partial [y_{\ell-1}]_j} \sigma\big([A_{w_\ell} y_{\ell-1}]_i + [b_\ell \mathbf{1}_{n_\ell}]_i\big) \right\}_{i,j} \\ &= \left\{ \sigma'\big([A_{w_\ell} y_{\ell-1}]_i + [b_\ell \mathbf{1}_{n_\ell}]_i\big) \frac{\partial}{\partial [y_{\ell-1}]_j} \big([A_{w_\ell} y_{\ell-1}]_i + [b_\ell \mathbf{1}_{n_\ell}]_i\big) \right\}_{i,j} \\ &= \left\{ \sigma'\big([A_{w_\ell} y_{\ell-1}]_i + [b_\ell \mathbf{1}_{n_\ell}]_i\big) \frac{\partial}{\partial [y_{\ell-1}]_j} \left(\sum_{k=1}^{n_{\ell-1}} [A_{w_\ell}]_{i,k} [y_{\ell-1}]_k + b_\ell \right) \right\}_{i,j} \\ &= \left\{ \sigma'\big([A_{w_\ell} y_{\ell-1}]_i + [b_\ell \mathbf{1}_{n_\ell}]_i\big) [A_{w_\ell}]_{i,j} \right\}_{i,j} \\ &= \operatorname{diag}(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell})) A_{w_\ell}.\end{aligned}$$

A_{w_ℓ} will be in the form of the following.

$$A_{w_\ell} = \begin{bmatrix} [w_\ell]_1 & \cdots & [w_\ell]_{f_\ell} & 0 & 0 & \cdots & 0 \\ 0 & [w_\ell]_1 & \cdots & [w_\ell]_{f_\ell} & 0 & \cdots & 0 \\ \vdots & & \ddots & & \ddots & & \vdots \\ 0 & \cdots & 0 & [w_\ell]_1 & \cdots & [w_\ell]_{f_\ell} & 0 \\ 0 & \cdots & 0 & 0 & [w_\ell]_1 & \cdots & [w_\ell]_{f_\ell} \end{bmatrix}$$

Then using the chain rule, for $\ell = 1, \cdots, L$, we obtain the following.

$$\begin{aligned}
\frac{\partial y_L}{\partial w_\ell} &= \frac{\partial y_L}{\partial y_\ell}\frac{\partial y_\ell}{\partial w_\ell} = \frac{\partial y_L}{\partial y_\ell}\left\{\frac{\partial [y_\ell]_i}{\partial [w_\ell]_j}\right\}_{i,j} \\
&= \frac{\partial y_L}{\partial y_\ell}\left\{\sigma'\big([A_{w_\ell}y_{\ell-1}]_i + [b_\ell \mathbf{1}_{n_\ell}]_i\big)\frac{\partial}{\partial [w_\ell]_j}\left(\sum_{k=1}^{n_{\ell-1}} [A_{w_\ell}]_{i,k}[y_{\ell-1}]_k + b_\ell\right)\right\}_{i,j} \\
&= \frac{\partial y_L}{\partial y_\ell}\left\{\sigma'\big([A_{w_\ell}y_{\ell-1}]_i + [b_\ell \mathbf{1}_{n_\ell}]_i\big)\sum_{k=1}^{n_{\ell-1}}\frac{\partial [A_{w_\ell}]_{i,k}}{\partial [w_\ell]_j}[y_{\ell-1}]_k\right\}_{i,j} \\
&= \frac{\partial y_L}{\partial y_\ell}\Big\{\sigma'\big([A_{w_\ell}y_{\ell-1}]_i + [b_\ell \mathbf{1}_{n_\ell}]_i\big)[y_{\ell-1}]_{i+j-1}\Big\}_{i,j} \\
&= \left\{\sum_{k=1}^{n_\ell}\left[\frac{\partial y_L}{\partial y_\ell}\right]_{i,k}\sigma'\big([A_{w_\ell}y_{\ell-1}]_k + [b_\ell \mathbf{1}_{n_\ell}]_k\big)[y_{\ell-1}]_{k+j-1}\right\}_{i,j} \\
&= \begin{bmatrix} \sum_{k=1}^{n_\ell}\left[\frac{\partial y_L}{\partial y_\ell}\right]_{1,k}\sigma'\big([A_{w_\ell}y_{\ell-1}]_k + [b_\ell \mathbf{1}_{n_\ell}]_k\big)[y_{\ell-1}]_k \\ \sum_{k=1}^{n_\ell}\left[\frac{\partial y_L}{\partial y_\ell}\right]_{1,k}\sigma'\big([A_{w_\ell}y_{\ell-1}]_k + [b_\ell \mathbf{1}_{n_\ell}]_k\big)[y_{\ell-1}]_{k+1} \\ \vdots \\ \sum_{k=1}^{n_\ell}\left[\frac{\partial y_L}{\partial y_\ell}\right]_{1,k}\sigma'\big([A_{w_\ell}y_{\ell-1}]_k + [b_\ell \mathbf{1}_{n_\ell}]_k\big)[y_{\ell-1}]_{k+f_\ell-1} \end{bmatrix}^\intercal \\
&= \left(\begin{bmatrix} [v_\ell^\intercal]_1 & \cdots & [v_\ell^\intercal]_{n_\ell} & 0 & 0 & \cdots & 0 \\ 0 & [v_\ell^\intercal]_1 & \cdots & [v_\ell^\intercal]_{n_\ell} & 0 & \cdots & 0 \\ \vdots & & \ddots & & \ddots & & \vdots \\ 0 & \cdots & 0 & [v_\ell^\intercal]_1 & \cdots & [v_\ell^\intercal]_{n_\ell} & 0 \\ 0 & \cdots & 0 & 0 & [v_\ell^\intercal]_1 & \cdots & [v_\ell^\intercal]_{n_\ell} \end{bmatrix}\begin{bmatrix} [y_{\ell-1}]_1 \\ [y_{\ell-1}]_2 \\ \vdots \\ [y_{\ell-1}]_{n_{\ell-1}-1} \\ [y_{\ell-1}]_{n_{\ell-1}} \end{bmatrix}\right)^\intercal \\
&= \left(\mathcal{C}_{v_\ell^\intercal}y_{\ell-1}\right)^\intercal \quad \left(\because\ [v_\ell^\intercal]_k = \left[\frac{\partial y_L}{\partial y_\ell}\right]_{1,k}\sigma'\big([A_{w_\ell}y_{\ell-1}]_k + [b_\ell \mathbf{1}_{n_\ell}]_k\big)\right)
\end{aligned}$$

$$\begin{aligned}
\frac{\partial y_L}{\partial b_\ell} &= \frac{\partial y_L}{\partial y_\ell}\frac{\partial y_\ell}{\partial b_\ell} \\
&= \frac{\partial y_L}{\partial y_\ell}\mathrm{diag}(\sigma'(A_{w_\ell}y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell}))\left(\frac{\partial}{\partial b_\ell}(A_{w_\ell}y_{\ell-1} + b_\ell \mathbf{1}_{n_\ell})\right) \\
&= v_\ell \mathbf{1}_{n_\ell}
\end{aligned}$$

(b) Performing multiplications $A_{w_i}v$ or vA_{w_i}, where v is a vector of appropriate size, should be computed by performing convolution each. vA_{w_i} can be computed according to the equation $vA_{w_i} = \left(A_{w_i}^{\intercal} v^{\intercal}\right)^{\intercal}$, so it should be computed by transpose-convolution. Instead of considering A_{w_i} or $A_{w_i}^{\intercal}$ as matrices each, we can perform the process by applying convolution to the target vector with respect to the convolutional filter w_i. During backprop, applying convolutions are performed sequentially regarding the chain rule.

Problem 3

[1]:
```
import torch.nn as nn
from torch.utils.data import DataLoader
import torch
import torchvision
import torchvision.transforms as transforms
```

[2]:
```
# Instantiate model with BN and load trained parameters

class smallNetTrain(nn.Module) :
    # CIFAR-10 data is 32*32 images with 3 RGB channels
    def __init__(self, input_dim=3*32*32) :
        super().__init__()

        self.conv1 = nn.Sequential(
                            nn.Conv2d(3, 16, kernel_size=3, padding=1),
                            nn.BatchNorm2d(16),
                            nn.ReLU()
                            )
        self.conv2 = nn.Sequential(
                            nn.Conv2d(16, 16, kernel_size=3, padding=1),
                            nn.BatchNorm2d(16),
                            nn.ReLU()
                            )
        self.fc1 = nn.Sequential(
                            nn.Linear(16*32*32, 32*32),
                            nn.BatchNorm1d(32*32),
                            nn.ReLU()
                            )
        self.fc2 = nn.Sequential(
                            nn.Linear(32*32, 10),
                            nn.ReLU()
                            )
    def forward(self, x) :
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.float().view(-1, 16*32*32)
        x = self.fc1(x)
        x = self.fc2(x)

        return x

model = smallNetTrain()
model.load_state_dict(torch.load("./smallNetSaved",map_location=torch.device('cpu')))
```

[2]:
```
<All keys matched successfully>
```

[3]:
```
# Instantiate model without BN

class smallNetTest(nn.Module) :
    # CIFAR-10 data is 32*32 images with 3 RGB channels
    def __init__(self, input_dim=3*32*32) :
```

```
        super().__init__()

        self.conv1 = nn.Sequential(
                            nn.Conv2d(3, 16, kernel_size=3, padding=1),
                            nn.ReLU()
                            )
        self.conv2 = nn.Sequential(
                            nn.Conv2d(16, 16, kernel_size=3, padding=1),
                            nn.ReLU()
                            )
        self.fc1 = nn.Sequential(
                            nn.Linear(16*32*32, 32*32),
                            nn.ReLU()
                            )
        self.fc2 = nn.Sequential(
                            nn.Linear(32*32, 10),
                            nn.ReLU()
                            )
    def forward(self, x) :
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.float().view(-1, 16*32*32)
        x = self.fc1(x)
        x = self.fc2(x)

        return x

model_test = smallNetTest()
```

[4]:
```
# Initialize weights of model without BN

conv1_bn_beta, conv1_bn_gamma = model.conv1[1].bias, model.conv1[1].weight
conv1_bn_mean, conv1_bn_var = model.conv1[1].running_mean, model.conv1[1].running_var
conv2_bn_beta, conv2_bn_gamma = model.conv2[1].bias, model.conv2[1].weight
conv2_bn_mean, conv2_bn_var = model.conv2[1].running_mean, model.conv2[1].running_var
fc1_bn_beta, fc1_bn_gamma = model.fc1[1].bias, model.fc1[1].weight
fc1_bn_mean, fc1_bn_var = model.fc1[1].running_mean, model.fc1[1].running_var
eps = 1e-5

model_test.conv1[0].weight.data = model.conv1[0].weight.data * (conv1_bn_gamma / torch.
 ↪sqrt(conv1_bn_var + eps)).view([model.conv1[0].out_channels, 1, 1, 1])
model_test.conv1[0].bias.data = (model.conv1[0].bias.data - conv1_bn_mean) / torch.
 ↪sqrt(conv1_bn_var + eps) * conv1_bn_gamma + conv1_bn_beta

model_test.conv2[0].weight.data = model.conv2[0].weight.data * (conv2_bn_gamma / torch.
 ↪sqrt(conv2_bn_var + eps)).view([model.conv2[0].out_channels, 1, 1, 1])
model_test.conv2[0].bias.data = (model.conv2[0].bias.data - conv2_bn_mean) / torch.
 ↪sqrt(conv2_bn_var + eps) * conv2_bn_gamma + conv2_bn_beta

model_test.fc1[0].weight.data = model.fc1[0].weight.data * (fc1_bn_gamma / torch.
 ↪sqrt(fc1_bn_var + eps)).view([model.fc1[0].out_features, 1])
model_test.fc1[0].bias.data = (model.fc1[0].bias.data - fc1_bn_mean) / torch.
 ↪sqrt(fc1_bn_var + eps) * fc1_bn_gamma + fc1_bn_beta
```

```
model_test.fc2[0].weight.data = model.fc2[0].weight.data
model_test.fc2[0].bias.data = model.fc2[0].bias.data
```

[5]:
```
# Verify difference between model and model_test

model.eval()
# model_test.eval()  # not necessary since model_test has no BN or dropout

test_dataset = torchvision.datasets.CIFAR10(root='./cifar_10data/',
                                train=False,
                                transform=transforms.ToTensor(), download = True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=100,␣
 ↪shuffle=False)

diff = []
with torch.no_grad():
    for images, _ in test_loader:
        diff.append(torch.norm(model(images) - model_test(images))**2)

print(max(diff)) # If less than 1e-08, you got the right answer.
```

```
Files already downloaded and verified
tensor(5.1554e-09)
```

Problem 5

(a)

[6]:
```
class Net1(nn.Module):
    def __init__(self, num_classes=10):
        super(Net1, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=1),
            nn.ReLU(),
            nn.Conv2d(64, 192, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(192, 384, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(384, 256, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1),
        )
        self.classifier = nn.Sequential(
            nn.Linear(256 * 18 * 18, 4096),
            nn.ReLU(),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Linear(4096, num_classes)
        )

    def forward(self, x):
```

```
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x

model1 = Net1() # model1 randomly initialized
```

[7]:
```
class Net2(nn.Module):
    def __init__(self, num_classes=10):
        super(Net2, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=1),
            nn.ReLU(),
            nn.Conv2d(64, 192, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(192, 384, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(384, 256, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1),
        )
        self.classifier = nn.Sequential(
            nn.Conv2d(256, 4096, kernel_size=18),
            nn.ReLU(),
            nn.Conv2d(4096, 4096, kernel_size=1),
            nn.ReLU(),
            nn.Conv2d(4096, num_classes, kernel_size=1)
        )

    def copy_weights_from(self, net1):
        with torch.no_grad():
            for i in range(0, len(self.features), 2):
                self.features[i].weight.copy_(net1.features[i].weight)
                self.features[i].bias.copy_(net1.features[i].bias)

            for i in range(0, len(self.classifier), 2):
                self.classifier[i].weight.copy_(net1.classifier[i].weight.view(self.
 ↪classifier[i].weight.shape))
                self.classifier[i].bias.copy_(net1.classifier[i].bias)

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

model2 = Net2()
model2.copy_weights_from(model1)
```

[8]:
```
test_dataset = torchvision.datasets.CIFAR10(
    root='./cifar_10data/',
    train=False,
    transform=torchvision.transforms.ToTensor()
)
```

```
test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=10
)

imgs, _ = next(iter(test_loader))
diff = torch.mean((model1(imgs) - model2(imgs).squeeze()) ** 2)
print(f"Average Pixel Difference: {diff.item()}") # should be small
```

```
Average Pixel Difference: 7.309216708446808e-17
```

(b)

[9]:
```
test_dataset = torchvision.datasets.CIFAR10(
    root='./cifar_10data/',
    train=False,
    transform=torchvision.transforms.Compose([
        torchvision.transforms.Resize((36, 38)),
        torchvision.transforms.ToTensor()
        ]),
    download=True
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=10,
    shuffle=False
)

images, _ = next(iter(test_loader))
b, w, h = images.shape[0], images.shape[-1], images.shape[-2]
out1 = torch.empty((b, 10, h - 31, w - 31))
for i in range(h - 31):
    for j in range(w - 31):
        out1[:, :, i, j] = model1(images[:, :, i:i+32, j:j+32])
out2 = model2(images)
diff = torch.mean((out1 - out2) ** 2)

print(f"Average Pixel Diff: {diff.item()}")
```

```
Files already downloaded and verified
Average Pixel Diff: 7.132095501731034e-17
```