2020-14281 이서용 HW7.

#1. True Softmax

$$V_\beta(x) = \frac{1}{\beta}\log\sum_{i=1}^{n}\exp(\beta x_i) \quad : \mathbb{R}^n \to \mathbb{R}$$

(a) $V_\beta(x) = \frac{1}{\beta}\log\left(e^{\beta x_1} + e^{\beta x_2} + \cdots + e^{\beta x_n}\right)$

Let $x_i = \max\{x_1, x_2, \ldots, x_n\}$. Then

$$V_\beta(x) = \frac{1}{\beta}\log\left(e^{\beta x_i}\left(\frac{1}{e^{\beta(x_i-x_1)}} + \frac{1}{e^{\beta(x_i-x_2)}} + \cdots + \frac{1}{e^{\beta(x_i-x_i)}} + \cdots + \frac{1}{e^{\beta(x_i-x_n)}}\right)\right)$$

$$\lim_{\beta\to\infty} V_\beta(x) = \lim_{\beta\to\infty}\frac{1}{\beta}\log\left(e^{\beta x_i}\left(0 + 0 + \cdots + 1 + 0 + \cdots 0\right)\right) = \lim_{\beta\to\infty}\frac{1}{\beta}\log(e^{\beta x_i}) = x_i$$

Thus it is clear that $V_\beta(x) \to \max\{x_1, \ldots, x_n\}$ as $\beta\to\infty$.

(b) $\frac{\partial V_1}{\partial x_i} = \frac{\partial \log\sum_{i=1}^{n}\exp(x_i)}{\partial x_i} = \frac{e^{x_i}}{\sum_{j=1}^{n}\exp(x_j)}$. Thus, $\nabla V_1 = \frac{1}{\sum_{j=1}^{n}\exp(x_j)}(e^{x_1}, e^{x_2}, \ldots, e^{x_n}) = \mu$.

(c) $$\frac{\partial V_\beta}{\partial x_i} = \frac{\partial \frac{1}{\beta}\log\sum_{i=1}^{n}\exp(\beta x_i)}{\partial x_i} = \frac{1}{\beta}\frac{\beta\exp(\beta x_i)}{\sum_{j=1}^{n}\exp(\beta x_j)} = \frac{\exp(\beta x_i)}{\sum_{j=1}^{n}\exp(\beta x_j)} = \frac{\exp(\beta(x_i - x_{i_{max}}))}{\sum_{j=1}^{n}\exp(\beta(x_j - x_{i_{max}}))}$$

If $i = i_{max}$, $\frac{\partial V_\beta}{\partial x_i} \to \frac{1}{1}$ as $\beta\to\infty$.

If $i \neq i_{max}$ $\frac{\partial V_\beta}{\partial x_i} \to \frac{0}{1} = 0$ as $\beta\to\infty$.

Thus, $\nabla V_\beta(x) \to e_{i_{max}}$ as $\beta\to\infty$.

#2. Are linear layers compute heavy?

$3\times227\times227$ image

Count addition/multiplication of linear/conv layers (in AlexNet given in counting-params.py)

- First Conv : filter : $\mathbb{R}^{64\times3\times11\times11}$ with stride 4.

  $(3\times11\times11)\times(64\times55\times55)$ 번의 곱셈

  $(3\times11\times11-1+1)\times64\times55\times55$ 번의 덧셈

  $\mathbb{R}^{64\times55\times55}$ $\Downarrow$ Max pool $\mathbb{R}^{64\times27\times27}$

- Second Conv filter : $\mathbb{R}^{192\times64\times5\times5}$ with padding 2

  $(64\times5\times5)\times(192\times27\times27)$ 번의 곱셈

  $(64\times5\times5-1+1)\times(192\times27\times27)$ 번의 덧셈

  $\mathbb{R}^{192\times27\times27}$ $\Downarrow$ Max pool $\mathbb{R}^{192\times13\times13}$

- Third Conv filter : $\mathbb{R}^{384\times192\times3\times3}$ with padding 1.

  $(192\times3\times3)\times(384\times13\times13)$ 번 곱셈

  $(192\times3\times3-1+1)\times(384\times13\times13)$ 번 덧셈

- Fourth Conv filter : $\mathbb{R}^{256\times384\times3\times3}$ with padding 1

  $(384\times3\times3)\times(256\times13\times13)$ 번 곱셈

  $(384\times3\times3-1+1)\times(256\times13\times13)$ 번 덧셈

- Last Conv : filter : $\mathbb{R}^{256\times256\times3\times3}$ with padding 1.

  $(256\times3\times3)\times(256\times13\times13)$ 번 곱셈

  $(256\times3\times3-1+1)\times(256\times13\times13)$ 번 덧셈.

655566528번의 곱셈, 덧셈

- First Linear : $\mathbb{R}^{256\times6\times6}\rightarrow\mathbb{R}^{4096}$

  $256\times6\times6\times4096$ 번 곱셈, $(256\times6\times6-1+1)\times4096$ 번의 덧셈

- Second Linear Layer : $\mathbb{R}^{4096}\rightarrow\mathbb{R}^{4096}$

  4096×4096 번 곱셈, $(4096-1+1)\times4096$ 번 덧셈

- Last Linear Layer : $\mathbb{R}^{4096}\rightarrow\mathbb{R}^{1000}$

  4096×1000 번 곱셈, $(4096-1+1)\times1000$ 번 덧셈.

58621952번의 곱셈, 덧셈

#4

(a) $$\left(\frac{\partial y_L}{\partial y_{L-1}}\right)_{1,i} = \frac{\partial y_L}{\partial (y_{L-1})_i} = \frac{\partial A_{w_L} y_{L-1} + b_L 1_{n_L}}{\partial (y_{L-1})_i} = (A_{w_L})_i \Rightarrow \frac{\partial y_L}{\partial y_{L-1}} = A_{w_L} \in \mathbb{R}^{1\times n_{L-1}}$$

$$\mathbb{R}^{n_\ell \times n_{\ell-1}} \ni \left(\frac{\partial y_\ell}{\partial y_{\ell-1}}\right)_{ij} = \frac{\partial (y_\ell)_i}{\partial (y_{\ell-1})_j} = \frac{\partial\left(\sigma(A_{w_\ell} y_{\ell-1} + b_\ell 1_{n_\ell})\right)_i}{\partial (y_{\ell-1})_j} = \left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell 1_{n_\ell})\right)_i (A_{w_\ell})_{ij}$$

$$\Rightarrow \left(\frac{\partial y_\ell}{\partial y_{\ell-1}}\right) = \mathrm{diag}\left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell 1_{n_\ell})\right) A_{w_\ell}$$

$$\mathbb{R}^{1\times n_{L-1}} \ni \left(\frac{\partial y_L}{\partial w_L}\right)_{1,i} = \frac{\partial (A_{w_L} y_{L-1} + b_L 1_{n_L})}{\partial (w_L)_i} = \sum_{j=1}^{n_{L-1}} \frac{\partial y_L}{\partial (A_{w_L})_{1j}} \frac{\partial (A_{w_L})_{1j}}{\partial (w_L)_i} = (y_{L-1})_i = 1 \times (y_{L-1})_i = V_L \times (y_{L-1})_i$$

Thus $$\frac{\partial y_L}{\partial w_L} = \left(C_{V_L^T} y_{L-1}\right)^T$$

$$\mathbb{R}^{1\times n_{\ell-1}-n_\ell+1} \ni \left(\frac{\partial y_L}{\partial w_\ell}\right)_{1,i} = \frac{\partial y_L}{\partial (w_\ell)_i} = \frac{\partial y_L}{\partial y_\ell} \cdot \frac{\partial y_\ell}{\partial (w_\ell)_i} = \sum_{j=1}^{n_\ell} \frac{\partial y_L}{\partial (y_\ell)_j} \frac{\partial (y_\ell)_j}{\partial (w_\ell)_i} = \sum_{j=1}^{n_\ell}\sum_{k=1}^{n_{\ell-1}} \frac{\partial y_L}{\partial (y_\ell)_j} \frac{\partial (y_\ell)_j}{\partial (A_{w_\ell})_{jk}} \frac{\partial (A_{w_\ell})_{jk}}{\partial (w_\ell)_i}$$

$$= \sum_{j=1}^{n_\ell}\sum_{k=1}^{n_{\ell-1}} \frac{\partial y_L}{\partial (y_\ell)_j} \sigma'(A_{w_\ell} y_{\ell-1} + b_\ell 1_{n_\ell})_j (y_{\ell-1})_k \, \alpha(i,j,k) \qquad * \ \alpha = \begin{cases} 1 & \text{if } k = i+j-1 \\ 0 & \text{else} \end{cases}$$

$$= \sum_{j=1}^{n_\ell} \frac{\partial y_L}{\partial (y_\ell)_j} \left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell 1_{n_\ell})\right)_j (y_{\ell-1})_{i+j-1}$$

Note we defined $$\mathbb{R}^{1\times n_\ell} \ni V_\ell = \frac{\partial y_L}{\partial y_\ell} \cdot \mathrm{diag}\left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell 1_{n_\ell})\right) = \left(\frac{\partial y_L}{\partial (y_\ell)_1}\left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell 1_{n_\ell})\right)_1 \cdots\cdots \frac{\partial y_L}{\partial (y_\ell)_{n_\ell}}\left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell 1_{n_\ell})\right)_{n_L}\right)$$

$$= \sum_{j=1}^{n_\ell} (V_\ell^T)_j (y_{\ell-1})_{i+j-1} = \left(C_{V_\ell^T} y_{\ell-1}\right)_i$$ Thus, $$\frac{\partial y_L}{\partial w_\ell} = \left(C_{V_\ell^T} y_{\ell-1}\right)^T$$

$$\mathbb{R} \ni \frac{\partial y_L}{\partial b_L} = \frac{\partial (A_{w_L} y_{L-1} + b_L)}{\partial b_L} = 1 = V_L 1_{n_L}$$

$$\mathbb{R} \ni \frac{\partial y_L}{\partial b_\ell} = \frac{\partial y_L}{\partial y_\ell} \cdot \frac{\partial y_\ell}{\partial b_\ell} = \sum_{j=1}^{n_\ell} \frac{\partial y_L}{\partial (y_\ell)_j} \frac{\partial (y_\ell)_j}{\partial b_\ell} = \sum_{j=1}^{n_\ell} \frac{\partial y_L}{\partial (y_\ell)_j} \cdot \left(\sigma'(A_{w_\ell} y_{\ell-1} + b_\ell 1_{n_\ell})\right)_j$$

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

(b) A_{w_ℓ}을 살피면 $A_{w_\ell} = \begin{pmatrix} k_1, & k_2, \dots, & k_{f_\ell}, & 0, & 0, \dots, & 0 \\ 0 & k_1, \dots & , k_{f_\ell} & & 0 & \\ & & \ddots & & & \\ & O & & k_1 & \cdots & k_{f_\ell} \end{pmatrix} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ 꼴임을 생각해볼 수 있다.

따라서 일단 forward pass에서는, $A_{w_\ell} y_{\ell-1}$ 등을 처리할 때, convolution을 이용해 A_{w_ℓ}을 fully construct 하지 않고 계산해야 효율적이다.

Backward pass의 경우, 일단 v_ℓ을 compute 해야 한다.

$$v_\ell = \frac{\partial y_L}{\partial y_{L-1}} \frac{\partial y_{L-1}}{\partial y_{L-2}} \cdots \frac{\partial y_{\ell+1}}{\partial y_\ell} \cdot \mathrm{diag}(\sigma'(\tilde{y}_\ell)) \qquad (* \, y_\ell = \sigma(\tilde{y}_\ell))$$

$$= A_{w_L} \cdot \mathrm{diag}(\sigma'(\tilde{y}_{L-1})) A_{w_{L-1}} \cdots \mathrm{diag}(\sigma'(\tilde{y}_{\ell+1})) A_{w_{\ell+1}} \mathrm{diag}(\sigma'(\tilde{y}_\ell))$$ 이므로,

v_ℓ을 compute 할 때도 A_{w_i}를 행렬로 곱하는 과정에서 convolution을 충분히 활용해줄 수 있다.

v_ℓ이 compute 되었다면 $\frac{\partial y_L}{\partial b_\ell} = v_\ell \mathbb{1}_{n_\ell}$은 그냥 계산해주면 되고,

$\frac{\partial y_L}{\partial w_\ell} = (C_{v_\ell^T} y_{\ell-1})^T$는 Transpose Convolution을 이용해서 계산해주면 총 $O(n_\ell \times n_{\ell-1})$개나 낭비되는 0 없이 계산 가능해진다.

P3

bn_remove.py

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


# 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')))
```

```python

# 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()



# 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-05
```

```python
# Initialize the following parameters
model_test.conv1[0].weight.data = model.conv1[0].weight * conv1_bn_gamma.reshape(-1,1,1,1) / (conv1_bn_var.reshape(-1,1,1,1)+eps)**0.5
model_test.conv1[0].bias.data = (model.conv1[0].bias-conv1_bn_mean)*conv1_bn_gamma / (conv1_bn_var+eps)**0.5 + conv1_bn_beta

model_test.conv2[0].weight.data = model.conv2[0].weight * conv2_bn_gamma.reshape(-1,1,1,1) / (conv2_bn_var.reshape(-1,1,1,1)+eps)**0.5
model_test.conv2[0].bias.data = (model.conv2[0].bias-conv2_bn_mean)*conv2_bn_gamma / (conv2_bn_var+eps)**0.5 + conv2_bn_beta

model_test.fc1[0].weight.data = model.fc1[0].weight * fc1_bn_gamma.reshape(-1,1) / (fc1_bn_var.reshape(-1,1)+eps)**0.5
model_test.fc1[0].bias.data = (model.fc1[0].bias-fc1_bn_mean)*fc1_bn_gamma / (fc1_bn_var+eps)**0.5 + fc1_bn_beta

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




# 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.



'''
For debugging purposes, you may want to match the output of conv1 first before
moving on working on conv2. To do so, you can replace the forward-evaluation
functions of the two models with
def forward(self, x) :
    x = self.conv1(x)
    return x
'''
```

결과는 다음과 같다

```
In [4]: runfile('C:/Users/sylee/OneDriv
화면/실수기/HW7')
Files already downloaded and verified
tensor(7.9200e-09)
```

잘나온다.

P5 : 다음과 같이 짠다

LNiN.py

```python
import torch.nn as nn
import torch
import torchvision


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


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),
        )
```

```python
        )

    ################################################################
    ### TODO: Complete initialization of self.classifier       ###
    ###        by filling in the ...                           ###
    ################################################################
        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):
                ####################################################
                ### TO DO: Correctly transfer weight of Net1     ###
                ####################################################
                if(i==0):
                    self.classifier[i].weight.copy_(net1.classifier[i].weight.view(4096, 256,18,18))
                elif(i==2):
                    self.classifier[i].weight.copy_(net1.classifier[i].weight.view(4096, 4096,1,1))
                else:
                    self.classifier[i].weight.copy_(net1.classifier[i].weight.view(10, 4096,1,1))
                self.classifier[i].bias.copy_(net1.classifier[i].bias)


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

```python



model1 = Net1() # model1 randomly initialized
model2 = Net2()
model2.copy_weights_from(model1)

test_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=False,
    transform=torchvision.transforms.ToTensor(),
    download = True
)

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

########################################################################


test_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    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):
        ########################################################
        ### TO DO: fill in ... to make out1 and out2 equal   ###
        ########################################################
        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()}")
```

(a) 번의 경우, linear layer와 동등한 conv layer들을 만들어놓고, weight의 size만 적절히 변환해서 넘겨주면 된다. (b)번의 경우, 어짜피 3*32*32짜리 image에 대해서 Net1이 돌기 때문에 이미지를 32*32 로 분할해서 넘겨주면(stride는 conv layer에서 전부 1을 사용하므로 1느낌으로) 그만이다.

결과는 다음과 같게 나온다.

```
In [26]: runfile('C:/Users/sylee/OneDrive/바탕 화
심수기/HW7')
Files already downloaded and verified
Average Pixel Difference: 8.350597394981668e-17
Files already downloaded and verified
Average Pixel Diff: 6.995657022774087e-17
```

거의 0이 나오는 것을 볼 수 있다. 즉 맞게 잘 했다.