MathDNN Homework 4

Department of Computer Science and Engineering
2021-16988 Jaewan Park

Problem 1

The parameters for convolution are $C_{\text{in}} = 1$, $C_{\text{out}} = 2$, $F = 3$, $S = 1$, $P = 1$. Then the relationship between X and Y becomes

$$Y_{l,i,j} = \sum_{\alpha=1}^{F}\sum_{\beta=1}^{F} w_{l,\alpha,\beta} X_{S(i-1)-P+\alpha,\, S(j-1)-P+\beta} + b_l$$

Therefore,

$$Y_{1,i,j} = \sum_{\alpha=1}^{3}\sum_{\beta=1}^{3} w_{1,\alpha,\beta} X_{i+\alpha-2,j+\beta-2} + b_1 = X_{i+1,j} - X_{i,j}$$

$$Y_{2,i,j} = \sum_{\alpha=1}^{3}\sum_{\beta=1}^{3} w_{2,\alpha,\beta} X_{i+\alpha-2,j+\beta-2} + b_2 = X_{i,j+1} - X_{i,j}$$

$$\therefore w_{1,\alpha,\beta} = \begin{cases} 1 & (\alpha = 3,\ \beta = 2) \\ -1 & (\alpha = 2,\ \beta = 2) \\ 0 & (\text{otherwise}) \end{cases},\quad w_{2,\alpha,\beta} = \begin{cases} 1 & (\alpha = 2,\ \beta = 3) \\ -1 & (\alpha = 2,\ \beta = 2) \\ 0 & (\text{otherwise}) \end{cases},\quad b = 0.$$

Problem 2

A common 2D convolution has the following relationship between X and Y. (Assume the batch size is 1.)

$$Y_{l,i,j} = \sum_{\gamma=1}^{C_{\text{in}}}\sum_{\alpha=1}^{f_1}\sum_{\beta=1}^{f_2} w_{l,\gamma,\alpha,\beta} X_{\gamma,S(i-1)+\alpha,S(j-1)+\beta} + b_l$$

In the case of the given average pooling operation, $C_{\text{in}} = C_{\text{out}} = C$, $f_1 = f_2 = S = k$, $b = 0$, $P = 0$, and

$$w_{l,\gamma,\alpha,\beta} = \begin{cases} \dfrac{1}{k^2} & (\gamma = l) \\ 0 & (\text{otherwise}) \end{cases}.$$

Substituting these to the above equation gives

$$\begin{aligned} Y_{l,i,j} &= \sum_{\gamma=1}^{C}\sum_{\alpha=1}^{k}\sum_{\beta=1}^{k} w_{l,\gamma,\alpha,\beta} X_{\gamma,k(i-1)+\alpha,k(j-1)+\beta} \\ &= \frac{1}{k^2}\sum_{\alpha=1}^{k}\sum_{\beta=1}^{k} X_{l,k(i-1)+\alpha,k(j-1)+\beta} \end{aligned}$$

which equals to the given relationship where indices α, β, l correspond to a, b, c. Also,

$$W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - f_1 + 2P}{S} + 1 \right\rfloor = \left\lfloor \frac{m - k + 0}{k} + 1 \right\rfloor = \frac{m}{k}$$

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} - f_2 + 2P}{S} + 1 \right\rfloor = \left\lfloor \frac{m - k + 0}{k} + 1 \right\rfloor = \frac{m}{k}$$

so the input and output dimensions match with the given conditions. In this way, average pooling can be represented as a convolution.

Problem 3

Consider an unbiased convolution and set parameters $C_{\text{in}} = 3$, $C_{\text{out}} = 1$, $F = 1$, $S = 1$, $P = 0$. Then substituting these parameters gives

$$\begin{aligned} Y_{i,j} &= \sum_{\gamma=1}^{C_{\text{in}}} \sum_{\alpha=1}^{F} \sum_{\beta=1}^{F} w_{\gamma,\alpha,\beta} X_{\gamma, S(i-1)+\alpha, S(j-1)+\beta} + b_l \\ &= w_{1,1,1} X_{1,i,j} + w_{2,1,1} X_{2,i,j} + w_{3,1,1} X_{3,i,j} \end{aligned}$$

Therefore $w_{1,1,1} = 0.299$, $w_{2,1,1} = 0.587$, $w_{3,1,1} = 0.114$.

Problem 4

Two functions σ and ρ are commute as the following.

$$\begin{aligned} [\sigma(\rho(X))]_{i,j} &= \sigma\Big([\rho(X)]_{i,j} \Big) \\ &= \sigma\left(\max_{1 \le p \le \frac{m}{k},\, 1 \le q \le \frac{n}{l}} X_{\frac{m}{k}(i-1)+p, \frac{n}{l}(j-1)+q} \right) \\ &= \max_{1 \le p \le \frac{m}{k},\, 1 \le q \le \frac{n}{l}} \sigma\Big(X_{\frac{m}{k}(i-1)+p, \frac{n}{l}(j-1)+q} \Big) \quad (\because \sigma \text{ is a nondecreasing function}) \\ &= \max_{1 \le p \le \frac{m}{k},\, 1 \le q \le \frac{n}{l}} [\sigma(X)]_{\frac{m}{k}(i-1)+p, \frac{n}{l}(j-1)+q} \\ &= [\rho(\sigma(X))]_{i,j} \end{aligned}$$

Problem 6

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

(a) Since $y_L = A_L y_{L-1} + b_L$, it is trivial that

$$\frac{\partial y_L}{\partial b_L} = 1, \quad \frac{\partial y_L}{\partial y_{L-1}} = A_L.$$

For $y_\ell = \sigma(A_\ell y_{\ell-1} + b_\ell)$, we can obtain the following.

$$\begin{aligned}
\frac{\partial y_\ell}{\partial b_\ell} &= \frac{\partial}{\partial b_\ell}\sigma(A_\ell y_{\ell-1} + b_\ell) \\
&= \left\{ \frac{\partial}{\partial [b_\ell]_j} [\sigma(A_\ell y_{\ell-1} + b_\ell)]_i \right\}_{i,j} = \left\{ \frac{\partial}{\partial [b_\ell]_j} \sigma([A_\ell y_{\ell-1} + b_\ell]_i) \right\}_{i,j} \\
&= \left\{ \frac{\partial}{\partial [b_\ell]_j} \sigma([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \right\}_{i,j} \\
&= \left\{ \sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \cdot \frac{\partial}{\partial [b_\ell]_j} ([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \right\}_{i,j} \\
&= \{\sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \cdot \delta_{ij}\}_{i,j} \quad (\delta_{ij}\text{: Kronecker Delta}) \\
&= \operatorname{diag}\Big(\sigma'(A_\ell y_{\ell-1} + b_\ell)\Big)
\end{aligned}$$

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

(b) Since $y_L = A_L y_{L-1} + b_L$,

$$\begin{aligned}
\left[\frac{\partial y_L}{\partial A_L}\right]_{1,j} &= \frac{\partial y_L}{\partial [A_L]_{1,j}} \\
&= \frac{\partial}{\partial [A_L]_{1,j}} (A_L y_{L-1} + b_L) = \frac{\partial}{\partial [A_L]_{1,j}} \left(\sum_{k=1}^{n_{\ell-1}} [A_L]_{1,k} [y_{L-1}]_k \right) \\
&= [y_{L-1}]_j
\end{aligned}$$

so $\dfrac{\partial y_L}{\partial A_L} = y_{L-1}^\intercal$. For $\ell = 1, \cdots, L-1$, we can obtain the following.

$$\begin{aligned}
\left[\frac{\partial y_L}{\partial A_\ell}\right]_{i,j} &= \frac{\partial y_L}{\partial [A_\ell]_{i,j}} = \frac{\partial y_L}{\partial y_\ell}\frac{\partial y_\ell}{\partial [A_\ell]_{i,j}} \\
&= \frac{\partial y_L}{\partial y_\ell}\left\{\frac{\partial [y_\ell]_k}{\partial [A_\ell]_{i,j}}\right\}_k \\
&= \frac{\partial y_L}{\partial y_\ell}\left\{\frac{\partial}{\partial [A_\ell]_{i,j}}\sigma([A_\ell y_{\ell-1}]_k + [b_\ell]_k)\right\}_k \\
&= \frac{\partial y_L}{\partial y_\ell}\left\{\sigma'([A_\ell y_{\ell-1}]_k + [b_\ell]_k)\left(\frac{\partial}{\partial [A_\ell]_{i,j}}([A_\ell y_{\ell-1}]_k + [b_\ell]_k)\right)\right\}_k \\
&= \frac{\partial y_L}{\partial y_\ell}\begin{bmatrix} 0 \\ \vdots \\ \sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i)[y_{\ell-1}]_j \\ \vdots \\ 0 \end{bmatrix} \quad \text{(All elements except the } i\text{-th element are 0.)} \\
&= \left[\frac{\partial y_L}{\partial y_\ell}\right]_i \sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i)[y_{\ell-1}]_j \\
&= [\sigma'(A_\ell y_{\ell-1} + b_\ell)]_i \left[\frac{\partial y_L}{\partial y_\ell}\right]_i [y_{\ell-1}]_j
\end{aligned}$$

Therefore $\dfrac{\partial y_L}{\partial A_\ell} = \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right)\left(\dfrac{\partial y_L}{\partial y_\ell}\right)^\intercal y_{\ell-1}^\intercal$.

Problem 5

[1]:

```
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
from random import shuffle
import matplotlib.pyplot as plt

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda
```

[2]:

```
'''
Step 1: Prepare dataset
'''
# Use data with only 4 and 9 as labels: which is hardest to classify
label_1, label_2 = 4, 9

# MNIST training data
train_set = datasets.MNIST(root='./mnist_data/', train=True, transform=transforms.
 ↪ToTensor(), download=True)

# Use data with two labels
idx = (train_set.targets == label_1) + (train_set.targets == label_2)
train_set.data = train_set.data[idx]
train_set.targets = train_set.targets[idx]
train_set.targets[train_set.targets == label_1] = -1
train_set.targets[train_set.targets == label_2] = 1

# MNIST testing data
test_set = datasets.MNIST(root='./mnist_data/', train=False, transform=transforms.
 ↪ToTensor())

# Use data with two labels
idx = (test_set.targets == label_1) + (test_set.targets == label_2)
test_set.data = test_set.data[idx]
test_set.targets = test_set.targets[idx]
test_set.targets[test_set.targets == label_1] = -1
test_set.targets[test_set.targets == label_2] = 1
```

[3]:

```
'''
Step 2: Define the neural network class
'''
class LR(nn.Module) :
    '''
    Initialize model
        input_dim : dimension of given input data
    '''
    # MNIST data is 28x28 images
    def __init__(self, input_dim=28*28) :
        super().__init__()
        self.linear = nn.Linear(input_dim, 1, bias=False)

    ''' forward given input x '''
    def forward(self, x) :
        return self.linear(x.float().view(-1, 28*28))
```

[4]:
```
'''
Step 3: Create the model, specify loss function and optimizer.
'''
model = LR().to(device)

def sum_of_squares_loss(output, target):
    return 1/2 * (1 - target) * ((1 - torch.sigmoid(-output))**2 + (torch.
 ↪sigmoid(output))**2) + 1/2 * (1 + target) * ((torch.sigmoid(-output))**2 + (1 -␣
 ↪torch.sigmoid(output))**2)

loss_function = sum_of_squares_loss
optimizer = torch.optim.SGD(model.parameters(), lr=255*1e-4)
```

[5]:
```
'''
Step 4: Train model with SGD
'''

# shuffled cyclic SGD
train_loader = DataLoader(dataset=train_set, batch_size=1, shuffle=True)

import time
start = time.time()
# Train the model (single epoch)
for image, label in train_loader :
    image, label = image.to(device), label.to(device)

    # Clear previously computed gradient
    optimizer.zero_grad()

    # then compute gradient with forward and backward passes
    train_loss = loss_function(model(image), label.float())
    train_loss.backward()

    # perform SGD step (parameter update)
    optimizer.step()
end = time.time()
print(f"Time ellapsed in training is: {end-start}")
```

```
Time ellapsed in training is: 18.5042941570282
```

[6]:
```
'''
Step 5: Test model (Evaluate the accuracy)
'''
test_loss, correct = 0, 0
misclassified_ind = []
```

```python
correct_ind = []

# Test data
test_loader = DataLoader(dataset=test_set, batch_size=1, shuffle=False)
# no need to shuffle test data

# Evaluate accuracy using test data
for ind, (image, label) in enumerate(test_loader) :
    image, label = image.to(device), label.to(device)

    # Forward pass
    output = model(image)

    # Calculate cumulative loss
    test_loss += loss_function(output, label.float()).item()

    # Make a prediction
    if output.item() * label.item() >= 0 :
        correct += 1
        correct_ind += [ind]
    else:
        misclassified_ind += [ind]

# Print out the results
print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /len(test_loader), correct, len(test_loader),
        100. * correct / len(test_loader)))
```

```
[Test set] Average loss: 0.0548, Accuracy: 1927/1991 (96.79%)
```

Now repeat steps 3 ˜ 5 using KL-divergence.

[7]:

```python
'''
Step 3: Create the model, specify loss function and optimizer.
'''
model = LR().to(device)

def logistic_loss(output, target):
    return -torch.nn.functional.logsigmoid(target*output)

loss_function = logistic_loss
optimizer = torch.optim.SGD(model.parameters(), lr=255*1e-4)
```

[8]:

```python
'''
Step 4: Train model with SGD
'''

# shuffled cyclic SGD
train_loader = DataLoader(dataset=train_set, batch_size=1, shuffle=True)

import time
start = time.time()
# Train the model (single epoch)
```

```
for image, label in train_loader :
    image, label = image.to(device), label.to(device)

    # Clear previously computed gradient
    optimizer.zero_grad()

    # then compute gradient with forward and backward passes
    train_loss = loss_function(model(image), label.float())
    train_loss.backward()

    # perform SGD step (parameter update)
    optimizer.step()
end = time.time()
print(f"Time ellapsed in training is: {end-start}")
```

```
Time ellapsed in training is: 9.621317386627197
```

[9]:

```
'''
Step 5: Test model (Evaluate the accuracy)
'''
test_loss, correct = 0, 0
misclassified_ind = []
correct_ind = []

# Test data
test_loader = DataLoader(dataset=test_set, batch_size=1, shuffle=False)
# no need to shuffle test data

# Evaluate accuracy using test data
for ind, (image, label) in enumerate(test_loader) :
    image, label = image.to(device), label.to(device)

    # Forward pass
    output = model(image)

    # Calculate cumulative loss
    test_loss += loss_function(output, label.float()).item()

    # Make a prediction
    if output.item() * label.item() >= 0 :
        correct += 1
        correct_ind += [ind]
    else:
        misclassified_ind += [ind]

# Print out the results
print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /len(test_loader), correct, len(test_loader),
        100. * correct / len(test_loader)))
```

```
[Test set] Average loss: 0.0935, Accuracy: 1928/1991 (96.84%)
```

Overall, using the two were similar in accuracy, but KL divergence showed better performance in terms of training time.

Problem 7

[10]:
```
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

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda
```

[11]:
```
'''
Step 1
'''

# MNIST dataset
train_dataset = datasets.MNIST(root='./mnist_data/',
                               train=True,
                               transform=transforms.ToTensor(),
                               download=True)

test_dataset = datasets.MNIST(root='./mnist_data/',
                              train=False,
                              transform=transforms.ToTensor())
```

[12]:
```
'''
Step 2: LeNet5
'''

# Modern LeNet uses this layer for C3
class C3_layer_full(nn.Module):
    def __init__(self):
        super(C3_layer_full, self).__init__()
        self.conv_layer = nn.Conv2d(6, 16, kernel_size=5)

    def forward(self, x):
        return self.conv_layer(x)

# Original LeNet uses this layer for C3
class C3_layer(nn.Module):
    def __init__(self):
        super(C3_layer, self).__init__()
        self.ch_in_3 = [[0, 1, 2],
                        [1, 2, 3],
                        [2, 3, 4],
                        [3, 4, 5],
                        [0, 4, 5],
                        [0, 1, 5]] # filter with 3 subset of input channels
        self.ch_in_4 = [[0, 1, 2, 3],
                        [1, 2, 3, 4],
                        [2, 3, 4, 5],
                        [0, 3, 4, 5],
                        [0, 1, 4, 5],
                        [0, 1, 2, 5],
                        [0, 1, 3, 4],
                        [1, 2, 4, 5],
                        [0, 2, 3, 5]] # filter with 4 subset of input channels
        # put implementation here
        self.ch_3_layers = nn.ModuleList([nn.Conv2d(3, 1, kernel_size=5) for _ in range(6)])
        self.ch_4_layers = nn.ModuleList([nn.Conv2d(4, 1, kernel_size=5) for _ in range(9)])
        self.ch_6_layer = nn.Conv2d(6, 1, kernel_size=5)

    def forward(self, x):
        ch_outputs = []
        ch_outputs += [self.ch_3_layers[i](x[:, self.ch_in_3[i], :, :]) for i in range(6)]
        ch_outputs += [self.ch_4_layers[i](x[:, self.ch_in_4[i], :, :]) for i in range(9)]
        ch_outputs += [self.ch_6_layer(x)]
        return torch.cat(ch_outputs, dim=1)

class LeNet(nn.Module) :
    def __init__(self) :
        super(LeNet, self).__init__()
        # padding=2 makes 28x28 image into 32x32
        self.C1_layer = nn.Sequential(
                nn.Conv2d(1, 6, kernel_size=5, padding=2),
                nn.Tanh()
                )
        self.P2_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C3_layer = nn.Sequential(
                # C3_layer_full(),
                C3_layer(),
                nn.Tanh()
                )
        self.P4_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C5_layer = nn.Sequential(
                nn.Linear(5*5*16, 120),
                nn.Tanh()
                )
        self.F6_layer = nn.Sequential(
                nn.Linear(120, 84),
                nn.Tanh()
                )
        self.F7_layer = nn.Linear(84, 10)
        self.tanh = nn.Tanh()

    def forward(self, x) :
        output = self.C1_layer(x)
        output = self.P2_layer(output)
        output = self.C3_layer(output)
        output = self.P4_layer(output)
        output = output.view(-1,5*5*16)
        output = self.C5_layer(output)
        output = self.F6_layer(output)
        output = self.F7_layer(output)
        return output
```

[13]:

```
'''
Step 3
'''
model = LeNet().to(device)
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# print total number of trainable parameters
param_ct = sum([p.numel() for p in model.parameters()])
print(f"Total number of trainable parameters: {param_ct}")
```

```
Total number of trainable parameters: 60806
```

[14]:

```
'''
Step 4
'''
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=100,␣
 ↪shuffle=True)

import time
start = time.time()
for epoch in range(10) :
    print("{}th epoch starting.".format(epoch))
    for images, labels in train_loader :
        images, labels = images.to(device), labels.to(device)

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

        optimizer.step()
end = time.time()
print("Time ellapsed in training is: {}".format(end - start))
```

```
0th epoch starting.
1th epoch starting.
2th epoch starting.
3th epoch starting.
```

```
4th epoch starting.
5th epoch starting.
6th epoch starting.
7th epoch starting.
8th epoch starting.
9th epoch starting.
Time ellapsed in training is: 97.80957221984863
```

[15]:

```
'''
Step 5
'''
test_loss, correct, total = 0, 0, 0

test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=100,␣
 ↪shuffle=False)

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

    output = model(images)
    test_loss += loss_function(output, labels).item()

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

    total += labels.size(0)

print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /total, correct, total,
        100. * correct / total))
```

```
[Test set] Average loss: 0.0005, Accuracy: 9826/10000 (98.26%)
```

Now try steps 2 ˜ 4 with the regular conv2d layer for C3.

[16]:

```
'''
Step 2: LeNet5
'''

# Modern LeNet uses this layer for C3
class C3_layer_full(nn.Module):
    def __init__(self):
        super(C3_layer_full, self).__init__()
        self.conv_layer = nn.Conv2d(6, 16, kernel_size=5)

    def forward(self, x):
        return self.conv_layer(x)

# Original LeNet uses this layer for C3
class C3_layer(nn.Module):
    def __init__(self):
        super(C3_layer, self).__init__()
        self.ch_in_3 = [[0, 1, 2],
                        [1, 2, 3],
                        [2, 3, 4],
                        [3, 4, 5],
                        [0, 4, 5],
                        [0, 1, 5]] # filter with 3 subset of input channels
        self.ch_in_4 = [[0, 1, 2, 3],
                        [1, 2, 3, 4],
                        [2, 3, 4, 5],
                        [0, 3, 4, 5],
                        [0, 1, 4, 5],
                        [0, 1, 2, 5],
                        [0, 1, 3, 4],
                        [1, 2, 4, 5],
                        [0, 2, 3, 5]] # filter with 4 subset of input channels
        # put implementation here
        self.ch_3_layers = nn.ModuleList([nn.Conv2d(3, 1, kernel_size=5) for _ in range␣
 ↪(6)])
        self.ch_4_layers = nn.ModuleList([nn.Conv2d(4, 1, kernel_size=5) for _ in range␣
 ↪(9)])
        self.ch_6_layer = nn.Conv2d(6, 1, kernel_size=5)

    def forward(self, x):
        ch_outputs = []
        ch_outputs += [self.ch_3_layers[i](x[:, self.ch_in_3[i], :, :]) for i in␣
 ↪range(6)]
        ch_outputs += [self.ch_4_layers[i](x[:, self.ch_in_4[i], :, :]) for i in␣
 ↪range(9)]
        ch_outputs += [self.ch_6_layer(x)]
        return torch.cat(ch_outputs, dim=1)

class LeNet(nn.Module) :
    def __init__(self) :
        super(LeNet, self).__init__()
        # padding=2 makes 28x28 image into 32x32
        self.C1_layer = nn.Sequential(
                nn.Conv2d(1, 6, kernel_size=5, padding=2),
                nn.Tanh()
                )
        self.P2_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C3_layer = nn.Sequential(
                C3_layer_full(),
                # C3_layer(),
                nn.Tanh()
                )
        self.P4_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C5_layer = nn.Sequential(
                nn.Linear(5*5*16, 120),
                nn.Tanh()
                )
        self.F6_layer = nn.Sequential(
                nn.Linear(120, 84),
                nn.Tanh()
                )
        self.F7_layer = nn.Linear(84, 10)
        self.tanh = nn.Tanh()

    def forward(self, x) :
        output = self.C1_layer(x)
        output = self.P2_layer(output)
        output = self.C3_layer(output)
        output = self.P4_layer(output)
        output = output.view(-1,5*5*16)
        output = self.C5_layer(output)
        output = self.F6_layer(output)
        output = self.F7_layer(output)
        return output

'''
Step 3
'''
model = LeNet().to(device)
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# print total number of trainable parameters
param_ct = sum([p.numel() for p in model.parameters()])
print(f"Total number of trainable parameters: {param_ct}")
```

```
Total number of trainable parameters: 61706
```

We can see that a total of 900 parameters are reduced. This can be calculated as

$$6\times(1\times5\times5+1)+6\times(3\times5\times5+1)+9\times(4\times5\times5+1)+1\times(6\times5\times5+1)+120\times(16\times5\times5+1)+84\times(120+1)+10\times(84+1) = 60806$$

for the original C3 model and

$$6 \times (1 \times 5 \times 5 + 1) + 16 \times (6 \times 5 \times 5 + 1) + 120 \times (16 \times 5 \times 5 + 1) + 84 \times (120 + 1) + 10 \times (84 + 1) = 61706$$

for complete C3 connections.

(The number of parameters needed between two layers is $C_{\text{out}} \times (C_{\text{in}} \times f_1 \times f_2 + b)$.)