MIS 583 Assignment 4: Self-supervised and transfer learning on CIFAR10

Before we start, please put your name and SID in following format: : LASTNAME Firstname, ?00000000 // e.g.) 李晨愷 M114020035

Your Answer:
Hi I'm 池品叡, B094020030.

Google Colab Setup

Next we need to run a few commands to set up our environment on Google Colab. If you are running this notebook on a local machine you can skip this section.

Run the following cell to mount your Google Drive. Follow the link, sign in to your Google account (the same account you used to store this notebook!) and copy the authorization code into the text box that appears below.

Data Setup (5 points)

The first thing to do is implement a dataset class to load rotated CIFAR10 images with matching labels. Since there is already a CIFAR10 dataset class implemented in `torchvision`, we will extend this class and modify the `__get_item__` method appropriately to load rotated images.

Each rotation label should be an integer in the set {0, 1, 2, 3} which correspond to rotations of 0, 90, 180, or 270 degrees respectively.

```
import torch
import torchvision
import torchvision.transforms as transforms
import numpy as np
import random


def rotate_img(img, rot):
    if rot == 0: # 0 degrees rotation
        return img
##########################################################################
#
    #        TODO: Implement rotate_img() - return the rotated img
#
##########################################################################
#
```

```
    elif rot == 1:
        return transforms.functional.rotate(img, 90)
    elif rot == 2:
        return transforms.functional.rotate(img, 180)
    elif rot == 3:
        return transforms.functional.rotate(img, 270)
    else:
        raise ValueError('rotation should be 0, 90, 180, or 270
degrees')


######################################################################
#
    #                          End of your code
#

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

class CIFAR10Rotation(torchvision.datasets.CIFAR10):

    def __init__(self, root, train, download, transform) -> None:
        super().__init__(root=root, train=train, download=download,
transform=transform)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index: int):
        image, cls_label = super().__getitem__(index)

        # randomly select image rotation
        rotation_label = random.choice([0, 1, 2, 3])
        image_rotated = rotate_img(image, rotation_label)

        rotation_label = torch.tensor(rotation_label).long()
        return image, image_rotated, rotation_label,
torch.tensor(cls_label).long()

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
```

```
0.2010)),
])

batch_size = 128

trainset = CIFAR10Rotation(root='./data', train=True,
                                        download=True,
transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset,
batch_size=batch_size,
                                          shuffle=True, num_workers=0)

testset = CIFAR10Rotation(root='./data', train=False,
                                       download=True,
transform=transform_test)
testloader = torch.utils.data.DataLoader(testset,
batch_size=batch_size,
                                         shuffle=False, num_workers=0)
```

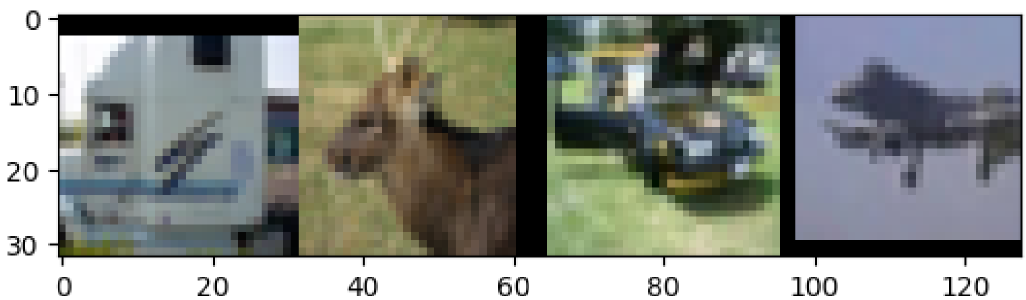
```
Files already downloaded and verified
Files already downloaded and verified
```

Show some example images and rotated images with labels:

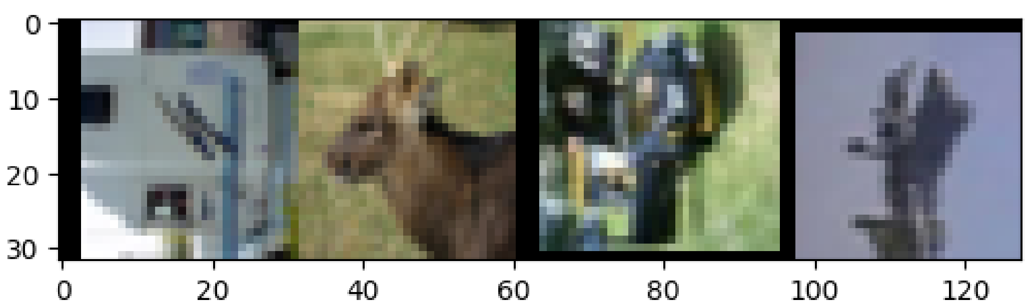
```
import matplotlib.pyplot as plt

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

rot_classes = ('0', '90', '180', '270')


def imshow(img):
    # unnormalize
    img = transforms.Normalize((0, 0, 0), (1/0.2023, 1/0.1994,
1/0.2010))(img)
    img = transforms.Normalize((-0.4914, -0.4822, -0.4465), (1, 1, 1))
(img)
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


dataiter = iter(trainloader)
images, rot_images, rot_labels, labels = next(dataiter)

# print images and rotated images
img_grid = imshow(torchvision.utils.make_grid(images[:4], padding=0))
print('Class labels: ', ' '.join(f'{classes[labels[j]]:5s}' for j in
range(4)))
```

```
img_grid = imshow(torchvision.utils.make_grid(rot_images[:4],
padding=0))
print('Rotation labels: ', ' '.join(f'{rot_classes[rot_labels[j]]:5s}'
for j in range(4)))
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).
```




```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).

Class labels:  truck deer  car   plane
```




```
Rotation labels:  90    0     90    270
```

Evaluation code

```
import time

def run_test(net, testloader, criterion, task):
    correct = 0
    total = 0
    avg_test_loss = 0.0
    # since we're not training, we don't need to calculate the
gradients for our outputs
    with torch.no_grad():
```

```
        for images, images_rotated, labels, cls_labels in testloader:
            if task == 'rotation':
              images, labels = images_rotated.to(device), 
labels.to(device)
            elif task == 'classification':
              images, labels = images.to(device), 
cls_labels.to(device)

######################################################################
#
            # TODO: Calculate outputs by running images through the 
network       #
            # The class with the highest energy is what we choose as 
prediction   #

######################################################################
#
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1) # Get the 
prediction result.
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

######################################################################
#
            #                           End of your code                  
#

######################################################################
#
            avg_test_loss += criterion(outputs, labels)  / 
len(testloader)
    print('TESTING:')
    print(f'Accuracy of the network on the 10000 test images: {100 * 
correct / total:.2f} %')
    print(f'Average loss on the 10000 test images: 
{avg_test_loss:.3f}')

def adjust_learning_rate(optimizer, epoch, init_lr, decay_epochs=30):
    """Sets the learning rate to the initial LR decayed by 10 every 30 
epochs"""
    lr = init_lr * (0.1 ** (epoch // decay_epochs))
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
```

Train a ResNet18 on the rotation task (9 points)

In this section, we will train a ResNet18 model **from scratch** on the rotation task. The input is a rotated image and the model predicts the rotation label. See the Data Setup section for details.

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

```
'cuda'
```

Notice: You should not use pretrained weights from ImageNet.

```
import torch.nn as nn
import torch.nn.functional as F

from torchvision.models import resnet18

net = resnet18(weights = None, num_classes=4) # Do not modify this
line.
net = net.to(device)
print(net) # print your model and check the num_classes is correct
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
```

```
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
```

```
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=4, bias=True)
)
```

```
import torch.nn as nn
import torch.optim as optim
######################################################################
##########
# TODO: Define loss and optmizer functions
#
# Try any loss or optimizer function and learning rate to get better
result    #
# hint: torch.nn and torch.optim
#
######################################################################
##########
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)
######################################################################
##########
#                              End of your code
#
######################################################################
##########
criterion = criterion.to(device)

# Both the self-supervised rotation task and supervised CIFAR10
classification are
# trained with the CrossEntropyLoss, so we can use the training loop
code.

def train(net, criterion, optimizer, num_epochs, decay_epochs,
init_lr, task):

    for epoch in range(num_epochs):  # loop over the dataset multiple
times

        running_loss = 0.0
        running_correct = 0.0
        running_total = 0.0
        start_time = time.time()

        net.train()

        for i, (imgs, imgs_rotated, rotation_label, cls_label) in
enumerate(trainloader, 0):
            adjust_learning_rate(optimizer, epoch, init_lr,
decay_epochs)

######################################################################
################################
            # TODO: Set the data to the correct device; Different task
will use different inputs and labels      #
            # TODO: Zero the parameter gradients
```

```
#
            # TODO: forward + backward + optimize
#
            # TODO: Get predicted results
#

######################################################################
################################
            inputs = torch.zeros(len(imgs))
            labels = torch.zeros(len(cls_label))
            if task == 'rotation':
              inputs = imgs_rotated.to(device)
              labels = rotation_label.to(device)

            elif task == 'classification':
              inputs = imgs.to(device)
              labels = cls_label.to(device)

            optimizer.zero_grad()
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            _, predicted = torch.max(outputs.data, 1)

######################################################################
################################
            #                              End of your code
#

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


            # print statistics
            print_freq = 100
            running_loss += loss.item()

            # calc acc
            running_total += labels.size(0)
            running_correct += (predicted == labels).sum().item()

            if i % print_freq == (print_freq - 1):    # print every
2000 mini-batches
                print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss
/ print_freq:.3f} acc: {100*running_correct / running_total:.2f} time:
{time.time() - start_time:.2f}')
                running_loss, running_correct, running_total = 0.0,
0.0, 0.0
                start_time = time.time()
```

```
################################################################
################################
        # TODO: Run the run_test() function after each epoch; Set the
model to the evaluation mode.          #

################################################################
################################
        run_test(net=net, testloader=testloader, criterion= criterion,
task = task)
        net.eval()

################################################################
################################
        #                              End of your code
#

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

    print('Finished Training')
```

Training Loop

```
net = torch.load('B094020030_rot_mdl.pt')

train(net, criterion, optimizer, num_epochs=45, decay_epochs=15,
init_lr=0.001, task='rotation')
################################
#     TODO: Save the model     #
################################
#torch.save(net.state_dict(), 'TRAINING.pt')
################################
#      End of your code        #
################################

[1,   100] loss: 1.274 acc: 43.03 time: 6.48
[1,   200] loss: 1.128 acc: 51.05 time: 7.10
[1,   300] loss: 1.107 acc: 51.54 time: 7.11
TESTING:
Accuracy of the network on the 10000 test images: 56.42 %
Average loss on the 10000 test images: 1.024
[2,   100] loss: 1.054 acc: 54.64 time: 6.24
[2,   200] loss: 1.025 acc: 56.62 time: 6.69
[2,   300] loss: 1.015 acc: 56.87 time: 7.19
TESTING:
Accuracy of the network on the 10000 test images: 58.61 %
Average loss on the 10000 test images: 0.986
[3,   100] loss: 0.984 acc: 58.45 time: 6.68
[3,   200] loss: 0.961 acc: 59.71 time: 6.98
```

```
[3,   300] loss: 0.957 acc: 59.40 time: 7.20
TESTING:
Accuracy of the network on the 10000 test images: 62.27 %
Average loss on the 10000 test images: 0.901
[4,   100] loss: 0.936 acc: 61.32 time: 6.68
[4,   200] loss: 0.932 acc: 61.03 time: 7.06
[4,   300] loss: 0.917 acc: 61.82 time: 7.03
TESTING:
Accuracy of the network on the 10000 test images: 63.25 %
Average loss on the 10000 test images: 0.880
[5,   100] loss: 0.915 acc: 61.82 time: 6.71
[5,   200] loss: 0.892 acc: 63.08 time: 6.83
[5,   300] loss: 0.889 acc: 63.19 time: 7.23
TESTING:
Accuracy of the network on the 10000 test images: 64.96 %
Average loss on the 10000 test images: 0.862
[6,   100] loss: 0.868 acc: 63.77 time: 6.42
[6,   200] loss: 0.858 acc: 64.73 time: 6.77
[6,   300] loss: 0.857 acc: 64.57 time: 6.88
TESTING:
Accuracy of the network on the 10000 test images: 66.12 %
Average loss on the 10000 test images: 0.827
[7,   100] loss: 0.852 acc: 65.11 time: 6.49
[7,   200] loss: 0.847 acc: 65.26 time: 6.90
[7,   300] loss: 0.823 acc: 65.94 time: 7.17
TESTING:
Accuracy of the network on the 10000 test images: 66.97 %
Average loss on the 10000 test images: 0.815
[8,   100] loss: 0.827 acc: 66.34 time: 6.60
[8,   200] loss: 0.823 acc: 66.41 time: 6.92
[8,   300] loss: 0.822 acc: 66.65 time: 7.16
TESTING:
Accuracy of the network on the 10000 test images: 66.99 %
Average loss on the 10000 test images: 0.810
[9,   100] loss: 0.817 acc: 66.62 time: 6.79
[9,   200] loss: 0.792 acc: 67.32 time: 7.03
[9,   300] loss: 0.798 acc: 67.69 time: 6.86
TESTING:
Accuracy of the network on the 10000 test images: 69.29 %
Average loss on the 10000 test images: 0.755
[10,   100] loss: 0.795 acc: 67.56 time: 6.63
[10,   200] loss: 0.787 acc: 67.86 time: 7.03
[10,   300] loss: 0.764 acc: 69.41 time: 6.81
TESTING:
Accuracy of the network on the 10000 test images: 69.68 %
Average loss on the 10000 test images: 0.752
[11,   100] loss: 0.757 acc: 69.28 time: 6.40
[11,   200] loss: 0.751 acc: 69.78 time: 6.81
[11,   300] loss: 0.760 acc: 69.62 time: 7.04
```

```
TESTING:
Accuracy of the network on the 10000 test images: 69.79 %
Average loss on the 10000 test images: 0.739
[12,   100] loss: 0.747 acc: 70.37 time: 6.78
[12,   200] loss: 0.750 acc: 69.99 time: 6.92
[12,   300] loss: 0.743 acc: 70.12 time: 7.26
TESTING:
Accuracy of the network on the 10000 test images: 71.46 %
Average loss on the 10000 test images: 0.724
[13,   100] loss: 0.731 acc: 70.88 time: 6.85
[13,   200] loss: 0.728 acc: 70.46 time: 6.77
[13,   300] loss: 0.716 acc: 71.45 time: 6.58
TESTING:
Accuracy of the network on the 10000 test images: 71.85 %
Average loss on the 10000 test images: 0.710
[14,   100] loss: 0.716 acc: 71.20 time: 6.87
[14,   200] loss: 0.709 acc: 71.94 time: 6.92
[14,   300] loss: 0.712 acc: 71.39 time: 6.49
TESTING:
Accuracy of the network on the 10000 test images: 72.40 %
Average loss on the 10000 test images: 0.698
[15,   100] loss: 0.700 acc: 72.43 time: 6.86
[15,   200] loss: 0.689 acc: 72.38 time: 6.91
[15,   300] loss: 0.685 acc: 72.81 time: 6.98
TESTING:
Accuracy of the network on the 10000 test images: 72.72 %
Average loss on the 10000 test images: 0.686
[16,   100] loss: 0.672 acc: 73.34 time: 7.06
[16,   200] loss: 0.624 acc: 75.57 time: 6.71
[16,   300] loss: 0.618 acc: 76.16 time: 6.78
TESTING:
Accuracy of the network on the 10000 test images: 74.93 %
Average loss on the 10000 test images: 0.622
[17,   100] loss: 0.613 acc: 75.87 time: 6.52
[17,   200] loss: 0.601 acc: 76.14 time: 6.99
[17,   300] loss: 0.608 acc: 76.51 time: 6.94
TESTING:
Accuracy of the network on the 10000 test images: 75.68 %
Average loss on the 10000 test images: 0.612
[18,   100] loss: 0.606 acc: 75.73 time: 6.77
[18,   200] loss: 0.601 acc: 76.63 time: 6.77
[18,   300] loss: 0.602 acc: 76.51 time: 7.06
TESTING:
Accuracy of the network on the 10000 test images: 75.81 %
Average loss on the 10000 test images: 0.614
[19,   100] loss: 0.598 acc: 76.31 time: 7.03
[19,   200] loss: 0.589 acc: 76.59 time: 6.69
[19,   300] loss: 0.597 acc: 76.46 time: 7.13
TESTING:
```

```
Accuracy of the network on the 10000 test images: 76.41 %
Average loss on the 10000 test images: 0.598
[20,   100] loss: 0.600 acc: 76.45 time: 6.66
[20,   200] loss: 0.595 acc: 76.41 time: 6.69
[20,   300] loss: 0.585 acc: 77.20 time: 6.53
TESTING:
Accuracy of the network on the 10000 test images: 76.41 %
Average loss on the 10000 test images: 0.598
[21,   100] loss: 0.587 acc: 77.18 time: 6.91
[21,   200] loss: 0.581 acc: 77.41 time: 6.94
[21,   300] loss: 0.590 acc: 76.82 time: 6.67
TESTING:
Accuracy of the network on the 10000 test images: 76.29 %
Average loss on the 10000 test images: 0.594
[22,   100] loss: 0.582 acc: 77.41 time: 7.08
[22,   200] loss: 0.596 acc: 76.69 time: 6.62
[22,   300] loss: 0.581 acc: 77.14 time: 6.51
TESTING:
Accuracy of the network on the 10000 test images: 76.82 %
Average loss on the 10000 test images: 0.587
[23,   100] loss: 0.597 acc: 76.62 time: 6.58
[23,   200] loss: 0.575 acc: 77.39 time: 6.81
[23,   300] loss: 0.563 acc: 78.23 time: 6.58
TESTING:
Accuracy of the network on the 10000 test images: 77.03 %
Average loss on the 10000 test images: 0.587
[24,   100] loss: 0.580 acc: 77.31 time: 6.58
[24,   200] loss: 0.581 acc: 77.19 time: 6.56
[24,   300] loss: 0.572 acc: 77.56 time: 6.53
TESTING:
Accuracy of the network on the 10000 test images: 76.98 %
Average loss on the 10000 test images: 0.581
[25,   100] loss: 0.576 acc: 77.30 time: 6.63
[25,   200] loss: 0.559 acc: 78.24 time: 6.76
[25,   300] loss: 0.573 acc: 77.84 time: 6.77
TESTING:
Accuracy of the network on the 10000 test images: 77.49 %
Average loss on the 10000 test images: 0.580
[26,   100] loss: 0.575 acc: 77.76 time: 6.93
[26,   200] loss: 0.557 acc: 78.19 time: 6.81
[26,   300] loss: 0.562 acc: 77.71 time: 6.89
TESTING:
Accuracy of the network on the 10000 test images: 77.26 %
Average loss on the 10000 test images: 0.580
[27,   100] loss: 0.569 acc: 77.77 time: 7.28
[27,   200] loss: 0.555 acc: 78.45 time: 7.00
[27,   300] loss: 0.569 acc: 77.61 time: 7.03
TESTING:
Accuracy of the network on the 10000 test images: 77.45 %
```

```
Average loss on the 10000 test images: 0.580
[28,   100] loss: 0.573 acc: 77.45 time: 6.98
[28,   200] loss: 0.561 acc: 77.92 time: 6.84
[28,   300] loss: 0.544 acc: 79.04 time: 6.96
TESTING:
Accuracy of the network on the 10000 test images: 76.75 %
Average loss on the 10000 test images: 0.590
[29,   100] loss: 0.569 acc: 77.85 time: 6.79
[29,   200] loss: 0.554 acc: 78.31 time: 6.74
[29,   300] loss: 0.556 acc: 78.25 time: 6.79
TESTING:
Accuracy of the network on the 10000 test images: 77.81 %
Average loss on the 10000 test images: 0.563
[30,   100] loss: 0.552 acc: 78.48 time: 6.61
[30,   200] loss: 0.548 acc: 78.64 time: 6.83
[30,   300] loss: 0.559 acc: 77.91 time: 7.00
TESTING:
Accuracy of the network on the 10000 test images: 77.62 %
Average loss on the 10000 test images: 0.574
[31,   100] loss: 0.549 acc: 78.59 time: 6.71
[31,   200] loss: 0.538 acc: 78.83 time: 6.63
[31,   300] loss: 0.547 acc: 78.62 time: 6.91
TESTING:
Accuracy of the network on the 10000 test images: 78.24 %
Average loss on the 10000 test images: 0.569
[32,   100] loss: 0.543 acc: 78.77 time: 6.73
[32,   200] loss: 0.541 acc: 78.64 time: 7.04
[32,   300] loss: 0.549 acc: 78.11 time: 6.91
TESTING:
Accuracy of the network on the 10000 test images: 77.93 %
Average loss on the 10000 test images: 0.564
[33,   100] loss: 0.545 acc: 78.62 time: 6.51
[33,   200] loss: 0.527 acc: 79.69 time: 6.92
[33,   300] loss: 0.551 acc: 78.41 time: 6.56
TESTING:
Accuracy of the network on the 10000 test images: 77.58 %
Average loss on the 10000 test images: 0.570
[34,   100] loss: 0.553 acc: 79.03 time: 6.67
[34,   200] loss: 0.536 acc: 79.05 time: 6.85
[34,   300] loss: 0.550 acc: 78.69 time: 6.92
TESTING:
Accuracy of the network on the 10000 test images: 77.70 %
Average loss on the 10000 test images: 0.566
[35,   100] loss: 0.535 acc: 79.23 time: 6.68
[35,   200] loss: 0.549 acc: 78.59 time: 6.70
[35,   300] loss: 0.552 acc: 78.34 time: 6.81
TESTING:
Accuracy of the network on the 10000 test images: 78.12 %
Average loss on the 10000 test images: 0.562
```

```
[36,   100] loss: 0.540 acc: 78.92 time: 6.44
[36,   200] loss: 0.548 acc: 78.51 time: 6.94
[36,   300] loss: 0.542 acc: 78.62 time: 7.10
TESTING:
Accuracy of the network on the 10000 test images: 77.95 %
Average loss on the 10000 test images: 0.556
[37,   100] loss: 0.546 acc: 78.68 time: 6.46
[37,   200] loss: 0.543 acc: 78.89 time: 6.81
[37,   300] loss: 0.542 acc: 78.81 time: 7.12
TESTING:
Accuracy of the network on the 10000 test images: 78.22 %
Average loss on the 10000 test images: 0.558
[38,   100] loss: 0.544 acc: 78.34 time: 6.71
[38,   200] loss: 0.548 acc: 78.51 time: 6.69
[38,   300] loss: 0.532 acc: 79.29 time: 6.95
TESTING:
Accuracy of the network on the 10000 test images: 78.09 %
Average loss on the 10000 test images: 0.563
[39,   100] loss: 0.540 acc: 78.96 time: 6.39
[39,   200] loss: 0.539 acc: 78.97 time: 6.93
[39,   300] loss: 0.537 acc: 79.38 time: 6.82
TESTING:
Accuracy of the network on the 10000 test images: 78.02 %
Average loss on the 10000 test images: 0.561
[40,   100] loss: 0.541 acc: 79.29 time: 7.11
[40,   200] loss: 0.544 acc: 78.74 time: 6.90
[40,   300] loss: 0.543 acc: 78.62 time: 6.88
TESTING:
Accuracy of the network on the 10000 test images: 78.38 %
Average loss on the 10000 test images: 0.562
[41,   100] loss: 0.537 acc: 79.09 time: 6.68
[41,   200] loss: 0.529 acc: 79.27 time: 6.87
[41,   300] loss: 0.540 acc: 78.89 time: 7.22
TESTING:
Accuracy of the network on the 10000 test images: 78.21 %
Average loss on the 10000 test images: 0.563
[42,   100] loss: 0.535 acc: 79.02 time: 6.69
[42,   200] loss: 0.560 acc: 77.89 time: 6.99
[42,   300] loss: 0.525 acc: 79.72 time: 7.18
TESTING:
Accuracy of the network on the 10000 test images: 78.08 %
Average loss on the 10000 test images: 0.563
[43,   100] loss: 0.532 acc: 79.41 time: 6.57
[43,   200] loss: 0.545 acc: 78.94 time: 6.77
[43,   300] loss: 0.535 acc: 79.05 time: 7.09
TESTING:
Accuracy of the network on the 10000 test images: 77.90 %
Average loss on the 10000 test images: 0.560
[44,   100] loss: 0.537 acc: 79.22 time: 6.58
```

```
[44,   200] loss: 0.536 acc: 79.27 time: 6.79
[44,   300] loss: 0.535 acc: 79.17 time: 7.10
TESTING:
Accuracy of the network on the 10000 test images: 78.06 %
Average loss on the 10000 test images: 0.561
[45,   100] loss: 0.537 acc: 79.30 time: 6.94
[45,   200] loss: 0.533 acc: 79.28 time: 6.95
[45,   300] loss: 0.541 acc: 78.96 time: 7.04
TESTING:
Accuracy of the network on the 10000 test images: 77.60 %
Average loss on the 10000 test images: 0.559
Finished Training

torch.save(net, 'B094020030_rot_mdl.pt')
```

Fine-tuning on the pre-trained model (9 points)

In this section, we will load the ResNet18 model pre-trained on the rotation task and fine-tune on the classification task. We will freeze all previous layers except for the 'layer4' block and 'fc' layer.

Then we will use the trained model from rotation task as the pretrained weights. Notice, you should not use the pretrained weights from ImageNet.

```
from sympy import true
import torch.nn as nn
import torch.nn.functional as F

from torchvision.models import resnet18

#####################################################
#     TODO: Load the pre-trained ResNet18 model     #
#####################################################
net = torch.load('B094020030_rot_mdl.pt')
print(net) # print your model and check the num_classes is correct
####################################################
#                End of your code                  #
####################################################

ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
```

```
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=4, bias=True)
)

######################################################################
###########################
#   TODO: Freeze all previous layers; only keep the 'layer4' block and
'fc' layer trainable     #
######################################################################
###########################
# 先凍結所有層
for params in net.parameters():
    params.requires_grad = False

# layer4 與 fc layer 依然可以更新權重
for params in net.layer4.parameters():
    params.requires_grad = True
for params in net.fc.parameters():
    params.requires_grad = True

# 修改 FC layer 架構
net.fc = nn.Linear(net.fc.in_features, 10) # 輸出 10 類別
net.to(device)

######################################################################
###########################
#                                           End of your code
#
######################################################################
###########################

ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

```python
# Print all the trainable parameters
params_to_update = net.parameters()
print("Params to learn:")
params_to_update = []
for name,param in net.named_parameters():
    if param.requires_grad == True:
        params_to_update.append(param)
        print("\t",name)
```

```
Params to learn:
	 layer4.0.conv1.weight
	 layer4.0.bn1.weight
	 layer4.0.bn1.bias
	 layer4.0.conv2.weight
	 layer4.0.bn2.weight
	 layer4.0.bn2.bias
	 layer4.0.downsample.0.weight
	 layer4.0.downsample.1.weight
	 layer4.0.downsample.1.bias
	 layer4.1.conv1.weight
	 layer4.1.bn1.weight
	 layer4.1.bn1.bias
	 layer4.1.conv2.weight
	 layer4.1.bn2.weight
	 layer4.1.bn2.bias
	 fc.weight
	 fc.bias
```

```python
# TODO: Define criterion and optimizer
# Note that your optimizer only needs to update the parameters that are trainable.
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam([
    {'params': net.layer4.parameters()},
    {'params': net.fc.parameters()}
], lr=0.001)
criterion = criterion.to(device)

train(net, criterion, optimizer, num_epochs=20, decay_epochs=20, init_lr=0.001, task='classification')
```

```
[1,   100] loss: 1.639 acc: 40.77 time: 6.17
[1,   200] loss: 1.379 acc: 49.68 time: 6.56
[1,   300] loss: 1.305 acc: 52.58 time: 6.66
TESTING:
Accuracy of the network on the 10000 test images: 55.96 %
Average loss on the 10000 test images: 1.231
[2,   100] loss: 1.227 acc: 55.88 time: 6.49
[2,   200] loss: 1.222 acc: 55.89 time: 6.55
[2,   300] loss: 1.209 acc: 56.29 time: 6.75
TESTING:
Accuracy of the network on the 10000 test images: 57.65 %
Average loss on the 10000 test images: 1.176
[3,   100] loss: 1.181 acc: 57.16 time: 6.44
[3,   200] loss: 1.160 acc: 58.10 time: 6.63
[3,   300] loss: 1.149 acc: 57.88 time: 6.93
TESTING:
Accuracy of the network on the 10000 test images: 59.25 %
Average loss on the 10000 test images: 1.134
[4,   100] loss: 1.117 acc: 59.50 time: 6.44
[4,   200] loss: 1.145 acc: 58.77 time: 6.63
[4,   300] loss: 1.130 acc: 58.95 time: 6.86
TESTING:
Accuracy of the network on the 10000 test images: 60.26 %
Average loss on the 10000 test images: 1.117
[5,   100] loss: 1.095 acc: 60.20 time: 6.62
[5,   200] loss: 1.097 acc: 60.33 time: 6.31
[5,   300] loss: 1.106 acc: 59.91 time: 6.45
TESTING:
Accuracy of the network on the 10000 test images: 61.19 %
Average loss on the 10000 test images: 1.096
[6,   100] loss: 1.084 acc: 60.95 time: 6.61
[6,   200] loss: 1.087 acc: 60.90 time: 6.48
[6,   300] loss: 1.081 acc: 60.90 time: 6.74
TESTING:
Accuracy of the network on the 10000 test images: 61.16 %
Average loss on the 10000 test images: 1.084
[7,   100] loss: 1.050 acc: 62.05 time: 6.55
```

```
[7,   200] loss: 1.053 acc: 62.30 time: 6.39
[7,   300] loss: 1.087 acc: 61.30 time: 7.00
TESTING:
Accuracy of the network on the 10000 test images: 61.56 %
Average loss on the 10000 test images: 1.070
[8,   100] loss: 1.034 acc: 62.53 time: 6.49
[8,   200] loss: 1.047 acc: 62.55 time: 6.42
[8,   300] loss: 1.053 acc: 61.93 time: 7.14
TESTING:
Accuracy of the network on the 10000 test images: 62.10 %
Average loss on the 10000 test images: 1.056
[9,   100] loss: 1.035 acc: 63.59 time: 6.56
[9,   200] loss: 1.024 acc: 63.16 time: 6.36
[9,   300] loss: 1.020 acc: 62.46 time: 6.92
TESTING:
Accuracy of the network on the 10000 test images: 62.56 %
Average loss on the 10000 test images: 1.044
[10,   100] loss: 1.013 acc: 63.13 time: 6.45
[10,   200] loss: 1.012 acc: 63.45 time: 6.41
[10,   300] loss: 1.023 acc: 63.02 time: 6.94
TESTING:
Accuracy of the network on the 10000 test images: 62.67 %
Average loss on the 10000 test images: 1.054
[11,   100] loss: 0.993 acc: 64.29 time: 6.55
[11,   200] loss: 0.994 acc: 63.86 time: 6.61
[11,   300] loss: 1.004 acc: 63.68 time: 7.02
TESTING:
Accuracy of the network on the 10000 test images: 63.35 %
Average loss on the 10000 test images: 1.030
[12,   100] loss: 0.995 acc: 64.72 time: 6.50
[12,   200] loss: 1.000 acc: 63.53 time: 6.59
[12,   300] loss: 0.985 acc: 64.69 time: 6.97
TESTING:
Accuracy of the network on the 10000 test images: 63.04 %
Average loss on the 10000 test images: 1.035
[13,   100] loss: 0.971 acc: 64.88 time: 6.53
[13,   200] loss: 0.999 acc: 63.59 time: 6.44
[13,   300] loss: 0.977 acc: 65.13 time: 6.97
TESTING:
Accuracy of the network on the 10000 test images: 64.08 %
Average loss on the 10000 test images: 1.018
[14,   100] loss: 0.976 acc: 64.76 time: 6.50
[14,   200] loss: 0.973 acc: 65.14 time: 6.45
[14,   300] loss: 0.970 acc: 64.96 time: 6.94
TESTING:
Accuracy of the network on the 10000 test images: 63.91 %
Average loss on the 10000 test images: 1.014
[15,   100] loss: 0.951 acc: 65.30 time: 6.54
[15,   200] loss: 0.969 acc: 65.38 time: 6.43
```

```
[15,   300] loss: 0.968 acc: 64.87 time: 6.98
TESTING:
Accuracy of the network on the 10000 test images: 63.57 %
Average loss on the 10000 test images: 1.026
[16,   100] loss: 0.944 acc: 66.29 time: 6.70
[16,   200] loss: 0.968 acc: 65.25 time: 6.40
[16,   300] loss: 0.966 acc: 65.29 time: 7.00
TESTING:
Accuracy of the network on the 10000 test images: 64.75 %
Average loss on the 10000 test images: 1.001
[17,   100] loss: 0.932 acc: 66.14 time: 6.60
[17,   200] loss: 0.942 acc: 65.91 time: 6.38
[17,   300] loss: 0.941 acc: 65.64 time: 6.96
TESTING:
Accuracy of the network on the 10000 test images: 63.51 %
Average loss on the 10000 test images: 1.022
[18,   100] loss: 0.939 acc: 66.84 time: 6.43
[18,   200] loss: 0.947 acc: 66.19 time: 6.41
[18,   300] loss: 0.942 acc: 66.16 time: 7.11
TESTING:
Accuracy of the network on the 10000 test images: 64.17 %
Average loss on the 10000 test images: 1.019
[19,   100] loss: 0.913 acc: 67.24 time: 6.44
[19,   200] loss: 0.940 acc: 66.19 time: 6.35
[19,   300] loss: 0.939 acc: 66.67 time: 7.09
TESTING:
Accuracy of the network on the 10000 test images: 64.75 %
Average loss on the 10000 test images: 1.001
[20,   100] loss: 0.920 acc: 66.70 time: 6.47
[20,   200] loss: 0.928 acc: 66.96 time: 6.38
[20,   300] loss: 0.931 acc: 66.42 time: 6.99
TESTING:
Accuracy of the network on the 10000 test images: 64.07 %
Average loss on the 10000 test images: 1.023
Finished Training

torch.save(net.state_dict(), 'FINETUNED_w.pth')
```

Fine-tuning on the randomly initialized model (9 points)

In this section, we will randomly initialize a ResNet18 model and fine-tune on the classification task. We will freeze all previous layers except for the 'layer4' block and 'fc' layer.

```
import torch.nn as nn
import torch.nn.functional as F

from torchvision.models import resnet18
##################################################
# TODO: Randomly initialize a ResNet18 model    #
```

```python
##################################################
net = resnet18(weights=None, num_classes=4) # 定義模型架構，no pre-trained wieght
net = net.to(device)
print(net) # print your model and check the num_classes is correctprint(net) # print your model and check the num_classes is correct
##################################################
#              End of your code                  #
##################################################
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
```

```
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
```

```
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=4, bias=True)
)

######################################################################
###########################
# TODO: Freeze all previous layers; only keep the 'layer4' block and
'fc' layer trainable       #
# To do this, you should set requires_grad=False for the frozen
layers.                           #
######################################################################
###########################
for params in net.parameters():
    params.requires_grad = False

for params in net.layer4.parameters():
```

```
    params.requires_grad = True
for params in net.fc.parameters():
    params.requires_grad = True

net.fc = nn.Linear(net.fc.in_features, 10) # 10 classes
net.to(device)
print(net)
######################################################################
###########################
#                                          End of your code
#
######################################################################
###########################
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
```

```
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)

# Print all the trainable parameters
params_to_update = net.parameters()
print("Params to learn:")
params_to_update = []
for name,param in net.named_parameters():
    if param.requires_grad == True:
        params_to_update.append(param)
        print("\t",name)
```

```
Params to learn:
	 layer4.0.conv1.weight
	 layer4.0.bn1.weight
	 layer4.0.bn1.bias
	 layer4.0.conv2.weight
	 layer4.0.bn2.weight
	 layer4.0.bn2.bias
	 layer4.0.downsample.0.weight
	 layer4.0.downsample.1.weight
	 layer4.0.downsample.1.bias
	 layer4.1.conv1.weight
	 layer4.1.bn1.weight
	 layer4.1.bn1.bias
	 layer4.1.conv2.weight
	 layer4.1.bn2.weight
	 layer4.1.bn2.bias
	 fc.weight
	 fc.bias
```

```
# TODO: Define criterion and optimizer
# Note that your optimizer only needs to update the parameters that
are trainable.
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(params=[
    {'params': net.layer4.parameters()},
    {'params': net.fc.parameters()}],
                       lr=0.001)
criterion = criterion.to(device)

train(net, criterion, optimizer, num_epochs=20, decay_epochs=10,
init_lr=0.001, task='classification')
```

```
[1,   100] loss: 2.037 acc: 26.94 time: 6.40
[1,   200] loss: 1.876 acc: 31.34 time: 6.30
[1,   300] loss: 1.840 acc: 33.59 time: 6.85
TESTING:
Accuracy of the network on the 10000 test images: 37.07 %
Average loss on the 10000 test images: 1.735
[2,   100] loss: 1.795 acc: 35.33 time: 6.57
[2,   200] loss: 1.776 acc: 35.18 time: 6.39
[2,   300] loss: 1.759 acc: 35.66 time: 7.01
TESTING:
Accuracy of the network on the 10000 test images: 40.29 %
Average loss on the 10000 test images: 1.665
[3,   100] loss: 1.733 acc: 37.01 time: 6.58
[3,   200] loss: 1.727 acc: 37.68 time: 6.26
[3,   300] loss: 1.738 acc: 36.58 time: 7.00
TESTING:
Accuracy of the network on the 10000 test images: 40.64 %
Average loss on the 10000 test images: 1.664
```

```
[4,   100] loss: 1.694 acc: 38.81 time: 6.65
[4,   200] loss: 1.725 acc: 37.46 time: 6.30
[4,   300] loss: 1.708 acc: 37.58 time: 6.95
TESTING:
Accuracy of the network on the 10000 test images: 40.54 %
Average loss on the 10000 test images: 1.643
[5,   100] loss: 1.689 acc: 38.89 time: 6.67
[5,   200] loss: 1.684 acc: 39.30 time: 6.26
[5,   300] loss: 1.685 acc: 39.07 time: 6.98
TESTING:
Accuracy of the network on the 10000 test images: 41.47 %
Average loss on the 10000 test images: 1.624
[6,   100] loss: 1.656 acc: 40.19 time: 6.61
[6,   200] loss: 1.665 acc: 40.05 time: 6.32
[6,   300] loss: 1.664 acc: 39.63 time: 7.07
TESTING:
Accuracy of the network on the 10000 test images: 41.95 %
Average loss on the 10000 test images: 1.618
[7,   100] loss: 1.650 acc: 40.43 time: 6.64
[7,   200] loss: 1.640 acc: 40.19 time: 6.42
[7,   300] loss: 1.650 acc: 40.37 time: 6.97
TESTING:
Accuracy of the network on the 10000 test images: 42.06 %
Average loss on the 10000 test images: 1.610
[8,   100] loss: 1.622 acc: 41.27 time: 6.60
[8,   200] loss: 1.633 acc: 40.82 time: 6.30
[8,   300] loss: 1.638 acc: 41.29 time: 7.00
TESTING:
Accuracy of the network on the 10000 test images: 41.95 %
Average loss on the 10000 test images: 1.612
[9,   100] loss: 1.621 acc: 41.36 time: 6.59
[9,   200] loss: 1.641 acc: 40.94 time: 6.36
[9,   300] loss: 1.614 acc: 41.95 time: 6.94
TESTING:
Accuracy of the network on the 10000 test images: 43.57 %
Average loss on the 10000 test images: 1.582
[10,   100] loss: 1.625 acc: 41.27 time: 6.55
[10,   200] loss: 1.623 acc: 41.34 time: 6.35
[10,   300] loss: 1.623 acc: 41.50 time: 6.97
TESTING:
Accuracy of the network on the 10000 test images: 43.37 %
Average loss on the 10000 test images: 1.590
[11,   100] loss: 1.589 acc: 43.30 time: 6.67
[11,   200] loss: 1.581 acc: 43.98 time: 6.37
[11,   300] loss: 1.572 acc: 43.66 time: 7.02
TESTING:
Accuracy of the network on the 10000 test images: 44.35 %
Average loss on the 10000 test images: 1.558
[12,   100] loss: 1.568 acc: 44.05 time: 6.86
```

```
[12,   200] loss: 1.568 acc: 43.66 time: 6.31
[12,   300] loss: 1.554 acc: 44.14 time: 6.97
TESTING:
Accuracy of the network on the 10000 test images: 44.98 %
Average loss on the 10000 test images: 1.546
[13,   100] loss: 1.554 acc: 44.31 time: 6.58
[13,   200] loss: 1.554 acc: 44.11 time: 6.33
[13,   300] loss: 1.561 acc: 44.03 time: 7.06
TESTING:
Accuracy of the network on the 10000 test images: 45.02 %
Average loss on the 10000 test images: 1.540
[14,   100] loss: 1.547 acc: 44.85 time: 6.68
[14,   200] loss: 1.553 acc: 44.34 time: 6.23
[14,   300] loss: 1.554 acc: 44.60 time: 7.03
TESTING:
Accuracy of the network on the 10000 test images: 45.23 %
Average loss on the 10000 test images: 1.537
[15,   100] loss: 1.557 acc: 44.30 time: 6.80
[15,   200] loss: 1.542 acc: 44.85 time: 6.17
[15,   300] loss: 1.542 acc: 44.27 time: 7.00
TESTING:
Accuracy of the network on the 10000 test images: 45.28 %
Average loss on the 10000 test images: 1.535
[16,   100] loss: 1.533 acc: 44.91 time: 6.76
[16,   200] loss: 1.532 acc: 45.69 time: 6.84
[16,   300] loss: 1.539 acc: 45.12 time: 7.00
TESTING:
Accuracy of the network on the 10000 test images: 45.54 %
Average loss on the 10000 test images: 1.533
[17,   100] loss: 1.530 acc: 44.97 time: 6.61
[17,   200] loss: 1.520 acc: 45.24 time: 6.31
[17,   300] loss: 1.543 acc: 44.69 time: 7.00
TESTING:
Accuracy of the network on the 10000 test images: 45.39 %
Average loss on the 10000 test images: 1.534
[18,   100] loss: 1.546 acc: 44.64 time: 6.56
[18,   200] loss: 1.540 acc: 45.39 time: 6.40
[18,   300] loss: 1.517 acc: 45.75 time: 7.15
TESTING:
Accuracy of the network on the 10000 test images: 45.72 %
Average loss on the 10000 test images: 1.532
[19,   100] loss: 1.539 acc: 44.33 time: 6.53
[19,   200] loss: 1.517 acc: 45.23 time: 6.46
[19,   300] loss: 1.532 acc: 45.12 time: 7.00
TESTING:
Accuracy of the network on the 10000 test images: 45.99 %
Average loss on the 10000 test images: 1.527
[20,   100] loss: 1.518 acc: 45.39 time: 6.62
[20,   200] loss: 1.524 acc: 44.91 time: 6.50
```

```
[20,   300] loss: 1.528 acc: 45.55 time: 7.03
TESTING:
Accuracy of the network on the 10000 test images: 45.89 %
Average loss on the 10000 test images: 1.525
Finished Training

torch.save(net.state_dict(), 'FINETUNED_rand.pth')
```

Supervised training on the pre-trained model (9 points)

In this section, we will load the ResNet18 model pre-trained on the rotation task and re-train the whole model on the classification task.

Then we will use the trained model from rotation task as the pretrained weights. Notice, you should not use the pretrained weights from ImageNet.

```
import torch.nn as nn
import torch.nn.functional as F

from torchvision.models import resnet18

#####################################################
#     TODO: Load the pre-trained ResNet18 model     #
#####################################################
net = torch.load("B094020030_rot_mdl.pt")
net.fc = nn.Linear(net.fc.in_features, 10) # 10 classes
net = net.to(device)
print(net) # print your model and check the num_classes is correct
#####################################################
#                End of your code                   #
#####################################################

ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
```

```
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
```

```
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

```
# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(params=net.parameters(),lr = 0.01)
criterion = criterion.to(device)
```

```
weights = torch.load("RETRAINED_w.pth")
net.load_state_dict(weights)
```

```
<All keys matched successfully>
```

```
train(net, criterion, optimizer, num_epochs=20, decay_epochs=10,
init_lr=0.01, task='classification')
```

```
[1,   100] loss: 1.576 acc: 41.07 time: 6.34
[1,   200] loss: 1.212 acc: 56.27 time: 6.72
[1,   300] loss: 1.101 acc: 60.99 time: 6.86
TESTING:
Accuracy of the network on the 10000 test images: 63.65 %
Average loss on the 10000 test images: 1.034
[2,   100] loss: 1.000 acc: 64.34 time: 6.43
[2,   200] loss: 0.931 acc: 67.43 time: 6.71
[2,   300] loss: 0.902 acc: 68.11 time: 6.65
TESTING:
Accuracy of the network on the 10000 test images: 67.53 %
Average loss on the 10000 test images: 0.931
[3,   100] loss: 0.855 acc: 70.34 time: 6.47
[3,   200] loss: 0.845 acc: 70.48 time: 6.51
[3,   300] loss: 0.826 acc: 71.20 time: 7.09
TESTING:
Accuracy of the network on the 10000 test images: 72.21 %
Average loss on the 10000 test images: 0.794
[4,   100] loss: 0.767 acc: 73.42 time: 6.73
[4,   200] loss: 0.757 acc: 73.59 time: 6.83
[4,   300] loss: 0.755 acc: 73.61 time: 6.93
TESTING:
Accuracy of the network on the 10000 test images: 73.58 %
Average loss on the 10000 test images: 0.769
[5,   100] loss: 0.715 acc: 75.17 time: 6.50
[5,   200] loss: 0.707 acc: 74.99 time: 6.56
```

```
[5,   300] loss: 0.706 acc: 75.39 time: 6.65
TESTING:
Accuracy of the network on the 10000 test images: 74.12 %
Average loss on the 10000 test images: 0.761
[6,   100] loss: 0.658 acc: 77.25 time: 6.72
[6,   200] loss: 0.663 acc: 76.49 time: 6.60
[6,   300] loss: 0.668 acc: 76.84 time: 6.90
TESTING:
Accuracy of the network on the 10000 test images: 76.62 %
Average loss on the 10000 test images: 0.682
[7,   100] loss: 0.621 acc: 78.45 time: 6.69
[7,   200] loss: 0.624 acc: 78.33 time: 6.78
[7,   300] loss: 0.642 acc: 77.94 time: 7.12
TESTING:
Accuracy of the network on the 10000 test images: 77.01 %
Average loss on the 10000 test images: 0.670
[8,   100] loss: 0.607 acc: 78.88 time: 6.41
[8,   200] loss: 0.607 acc: 78.64 time: 6.83
[8,   300] loss: 0.613 acc: 79.12 time: 6.77
TESTING:
Accuracy of the network on the 10000 test images: 76.98 %
Average loss on the 10000 test images: 0.670
[9,   100] loss: 0.564 acc: 80.40 time: 6.59
[9,   200] loss: 0.580 acc: 80.09 time: 6.54
[9,   300] loss: 0.587 acc: 79.83 time: 6.84
TESTING:
Accuracy of the network on the 10000 test images: 76.82 %
Average loss on the 10000 test images: 0.688
[10,   100] loss: 0.555 acc: 80.83 time: 6.88
[10,   200] loss: 0.544 acc: 81.19 time: 6.71
[10,   300] loss: 0.555 acc: 80.90 time: 6.85
TESTING:
Accuracy of the network on the 10000 test images: 78.66 %
Average loss on the 10000 test images: 0.623
[11,   100] loss: 0.466 acc: 83.62 time: 6.62
[11,   200] loss: 0.449 acc: 84.49 time: 6.83
[11,   300] loss: 0.443 acc: 84.62 time: 6.95
TESTING:
Accuracy of the network on the 10000 test images: 80.49 %
Average loss on the 10000 test images: 0.567
[12,   100] loss: 0.435 acc: 85.36 time: 6.20
[12,   200] loss: 0.411 acc: 85.56 time: 6.87
[12,   300] loss: 0.423 acc: 85.05 time: 7.06
TESTING:
Accuracy of the network on the 10000 test images: 80.45 %
Average loss on the 10000 test images: 0.572
[13,   100] loss: 0.403 acc: 85.76 time: 6.52
[13,   200] loss: 0.422 acc: 85.28 time: 6.76
[13,   300] loss: 0.403 acc: 86.04 time: 7.04
```

```
TESTING:
Accuracy of the network on the 10000 test images: 80.51 %
Average loss on the 10000 test images: 0.571
[14,   100] loss: 0.404 acc: 86.15 time: 6.31
[14,   200] loss: 0.410 acc: 85.79 time: 6.87
[14,   300] loss: 0.399 acc: 86.22 time: 7.08
TESTING:
Accuracy of the network on the 10000 test images: 80.96 %
Average loss on the 10000 test images: 0.562
[15,   100] loss: 0.394 acc: 86.46 time: 6.66
[15,   200] loss: 0.390 acc: 86.36 time: 6.79
[15,   300] loss: 0.392 acc: 86.56 time: 7.04
TESTING:
Accuracy of the network on the 10000 test images: 81.46 %
Average loss on the 10000 test images: 0.550
[16,   100] loss: 0.379 acc: 86.90 time: 6.53
[16,   200] loss: 0.386 acc: 86.77 time: 6.76
[16,   300] loss: 0.395 acc: 86.27 time: 7.06
TESTING:
Accuracy of the network on the 10000 test images: 81.22 %
Average loss on the 10000 test images: 0.563
[17,   100] loss: 0.383 acc: 86.94 time: 6.60
[17,   200] loss: 0.373 acc: 86.98 time: 6.81
[17,   300] loss: 0.374 acc: 87.05 time: 7.17
TESTING:
Accuracy of the network on the 10000 test images: 81.47 %
Average loss on the 10000 test images: 0.555
[18,   100] loss: 0.364 acc: 87.58 time: 6.57
[18,   200] loss: 0.370 acc: 87.18 time: 6.77
[18,   300] loss: 0.373 acc: 87.12 time: 7.05
TESTING:
Accuracy of the network on the 10000 test images: 81.27 %
Average loss on the 10000 test images: 0.551
[19,   100] loss: 0.372 acc: 86.88 time: 6.55
[19,   200] loss: 0.362 acc: 87.55 time: 6.72
[19,   300] loss: 0.371 acc: 86.81 time: 7.03
TESTING:
Accuracy of the network on the 10000 test images: 81.28 %
Average loss on the 10000 test images: 0.553
[20,   100] loss: 0.359 acc: 87.73 time: 6.57
[20,   200] loss: 0.356 acc: 87.62 time: 6.72
[20,   300] loss: 0.365 acc: 87.61 time: 7.09
TESTING:
Accuracy of the network on the 10000 test images: 81.15 %
Average loss on the 10000 test images: 0.550
Finished Training
```

```python
torch.save(net.state_dict(), 'RETRAINED_w.pth')
```

Supervised training on the randomly initialized model (9 points)

In this section, we will randomly initialize a ResNet18 model and re-train the whole model on the classification task.

```
import torch.nn as nn
import torch.nn.functional as F

from torchvision.models import resnet18

#################################################
# TODO: Randomly initialize a ResNet18 model    #
#################################################
net = resnet18(weights=None)
net.fc = nn.Linear(net.fc.in_features, 10)

net.to(device)
print(net) # print your model and check the num_classes is correct
#################################################
#              End of your code                 #
#################################################
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
```

```
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
```

```
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
```

```
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

```python
# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(params=net.parameters(),lr=0.01)
criterion = criterion.to(device)

train(net, criterion, optimizer, num_epochs=20, decay_epochs=10,
init_lr=0.01, task='classification')
```

```
[1,   100] loss: 2.320 acc: 19.63 time: 6.68
[1,   200] loss: 1.896 acc: 31.17 time: 6.72
[1,   300] loss: 1.760 acc: 34.65 time: 6.95
TESTING:
Accuracy of the network on the 10000 test images: 43.24 %
Average loss on the 10000 test images: 1.541
[2,   100] loss: 1.584 acc: 41.49 time: 6.94
[2,   200] loss: 1.499 acc: 45.49 time: 6.79
[2,   300] loss: 1.452 acc: 46.80 time: 6.28
TESTING:
Accuracy of the network on the 10000 test images: 49.02 %
Average loss on the 10000 test images: 1.373
[3,   100] loss: 1.319 acc: 51.61 time: 6.57
[3,   200] loss: 1.263 acc: 54.35 time: 6.76
[3,   300] loss: 1.207 acc: 56.52 time: 6.47
TESTING:
Accuracy of the network on the 10000 test images: 61.76 %
Average loss on the 10000 test images: 1.088
[4,   100] loss: 1.122 acc: 59.72 time: 6.67
[4,   200] loss: 1.080 acc: 61.77 time: 6.76
[4,   300] loss: 1.052 acc: 62.30 time: 6.36
TESTING:
Accuracy of the network on the 10000 test images: 64.74 %
Average loss on the 10000 test images: 1.003
[5,   100] loss: 1.007 acc: 64.51 time: 6.53
[5,   200] loss: 0.968 acc: 65.34 time: 6.80
[5,   300] loss: 0.967 acc: 66.00 time: 6.15
TESTING:
Accuracy of the network on the 10000 test images: 68.85 %
Average loss on the 10000 test images: 0.912
[6,   100] loss: 0.918 acc: 67.42 time: 6.44
[6,   200] loss: 0.891 acc: 68.47 time: 6.62
[6,   300] loss: 0.885 acc: 69.08 time: 6.88
TESTING:
Accuracy of the network on the 10000 test images: 71.11 %
Average loss on the 10000 test images: 0.835
[7,   100] loss: 0.835 acc: 70.54 time: 6.64
[7,   200] loss: 0.825 acc: 71.23 time: 6.84
[7,   300] loss: 0.829 acc: 70.38 time: 6.92
```

```
TESTING:
Accuracy of the network on the 10000 test images: 71.18 %
Average loss on the 10000 test images: 0.836
[8,   100] loss: 0.800 acc: 72.40 time: 6.59
[8,   200] loss: 0.776 acc: 72.95 time: 6.72
[8,   300] loss: 0.788 acc: 72.41 time: 7.05
TESTING:
Accuracy of the network on the 10000 test images: 73.02 %
Average loss on the 10000 test images: 0.787
[9,   100] loss: 0.739 acc: 74.66 time: 7.11
[9,   200] loss: 0.729 acc: 74.73 time: 6.41
[9,   300] loss: 0.736 acc: 74.23 time: 6.48
TESTING:
Accuracy of the network on the 10000 test images: 74.80 %
Average loss on the 10000 test images: 0.728
[10,   100] loss: 0.706 acc: 75.68 time: 5.91
[10,   200] loss: 0.704 acc: 75.73 time: 6.51
[10,   300] loss: 0.691 acc: 75.91 time: 7.29
TESTING:
Accuracy of the network on the 10000 test images: 74.40 %
Average loss on the 10000 test images: 0.743
[11,   100] loss: 0.608 acc: 79.06 time: 6.57
[11,   200] loss: 0.577 acc: 80.04 time: 6.84
[11,   300] loss: 0.576 acc: 79.76 time: 7.04
TESTING:
Accuracy of the network on the 10000 test images: 78.20 %
Average loss on the 10000 test images: 0.637
[12,   100] loss: 0.529 acc: 81.39 time: 7.04
[12,   200] loss: 0.545 acc: 81.02 time: 7.03
[12,   300] loss: 0.539 acc: 81.66 time: 6.75
TESTING:
Accuracy of the network on the 10000 test images: 79.04 %
Average loss on the 10000 test images: 0.621
[13,   100] loss: 0.517 acc: 81.72 time: 6.62
[13,   200] loss: 0.527 acc: 81.44 time: 6.62
[13,   300] loss: 0.509 acc: 82.05 time: 6.75
TESTING:
Accuracy of the network on the 10000 test images: 79.43 %
Average loss on the 10000 test images: 0.618
[14,   100] loss: 0.530 acc: 81.52 time: 6.37
[14,   200] loss: 0.502 acc: 82.52 time: 6.42
[14,   300] loss: 0.498 acc: 82.60 time: 7.07
TESTING:
Accuracy of the network on the 10000 test images: 79.00 %
Average loss on the 10000 test images: 0.617
[15,   100] loss: 0.476 acc: 83.38 time: 6.41
[15,   200] loss: 0.491 acc: 82.71 time: 6.29
[15,   300] loss: 0.509 acc: 82.50 time: 6.60
TESTING:
```

```
Accuracy of the network on the 10000 test images: 79.51 %
Average loss on the 10000 test images: 0.605
[16,   100] loss: 0.471 acc: 83.52 time: 6.66
[16,   200] loss: 0.494 acc: 82.63 time: 6.87
[16,   300] loss: 0.479 acc: 83.01 time: 6.91
TESTING:
Accuracy of the network on the 10000 test images: 79.75 %
Average loss on the 10000 test images: 0.601
[17,   100] loss: 0.471 acc: 83.34 time: 6.97
[17,   200] loss: 0.461 acc: 83.91 time: 6.36
[17,   300] loss: 0.470 acc: 83.43 time: 6.58
TESTING:
Accuracy of the network on the 10000 test images: 79.97 %
Average loss on the 10000 test images: 0.605
[18,   100] loss: 0.475 acc: 83.36 time: 6.63
[18,   200] loss: 0.449 acc: 83.94 time: 6.67
[18,   300] loss: 0.457 acc: 83.82 time: 6.90
TESTING:
Accuracy of the network on the 10000 test images: 80.04 %
Average loss on the 10000 test images: 0.596
[19,   100] loss: 0.444 acc: 84.50 time: 6.66
[19,   200] loss: 0.451 acc: 84.32 time: 6.89
[19,   300] loss: 0.449 acc: 84.30 time: 7.16
TESTING:
Accuracy of the network on the 10000 test images: 79.99 %
Average loss on the 10000 test images: 0.600
[20,   100] loss: 0.439 acc: 84.63 time: 6.52
[20,   200] loss: 0.443 acc: 84.51 time: 6.77
[20,   300] loss: 0.445 acc: 84.52 time: 7.02
TESTING:
Accuracy of the network on the 10000 test images: 80.51 %
Average loss on the 10000 test images: 0.592
Finished Training

torch.save(net.state_dict(), 'RETRAINED_rand.pth')
```

Write report (37 points)

本次作業主要有 3 個 tasks 需要大家完成，在 A4.pdf 中有希望大家達成的 baseline **(**不能低於 **baseline 最多 2%**，沒有達到不會給全部分數**)**，report 的撰寫請大家根據以下要求完成，就請大家將嘗試的結果寫在 report 裡，祝大家順利！

1. (13 points) Train a ResNet18 on the Rotation task and report the test performance. Discuss why such a task helps in learning features that are generalizable to other visual tasks.

2. (12 points) Initializing from the Rotation model or from random weights, fine-tune only the weights of the final block of convolutional layers and linear layer on the

supervised CIFAR10 classification task. Report the test results and compare the performance of these two models. Provide your observations and insights. You can also discuss how the performance of pre-trained models affects downstream tasks, the performance of fine-tuning different numbers of layers, and so on.

3. (12 points) Initializing from the Rotation model or from random weights, train the full network on the supervised CIFAR10 classification task. Report the test results and compare the performance of these two models. Provide your observations and insights.

Extra Credit (13 points)

上面基本的 code 跟 report 最高可以拿到 87 分，這個加分部分並沒有要求同學們一定要做，若同學們想要獲得更高的分數可以根據以下的加分要求來獲得加分。

- In Figure 5(b) from the Gidaris et al. paper, the authors show a plot of CIFAR10 classification performance vs. number of training examples per category for a supervised CIFAR10 model vs. a RotNet model with the final layers fine-tuned on CIFAR10. The plot shows that pre-training on the Rotation task can be advantageous when only a small amount of labeled data is available. Using your RotNet fine-tuning code and supervised CIFAR10 training code from the main assignment, try to create a similar plot by performing supervised fine-tuning/training on only a subset of CIFAR10.
- Use a more advanced model than ResNet18 to try to get higher accuracy on the rotation prediction task, as well as for transfer to supervised CIFAR10 classification.
- If you have a good amount of compute at your disposal, try to train a rotation prediction model on the larger ImageNette dataset (still smaller than ImageNet, though).