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 游雅淇, B104020012.

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.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

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.

In [ ]:
```
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, transfo

    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
```

In [ ]:

```
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_
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = CIFAR10Rotation(root='./data', train=False,
                                       download=True, transform=transform_te
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
```

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

Out[ ]:

```
<torch.utils.data.dataloader.DataLoader at 0x780f91f8f9a0>
```

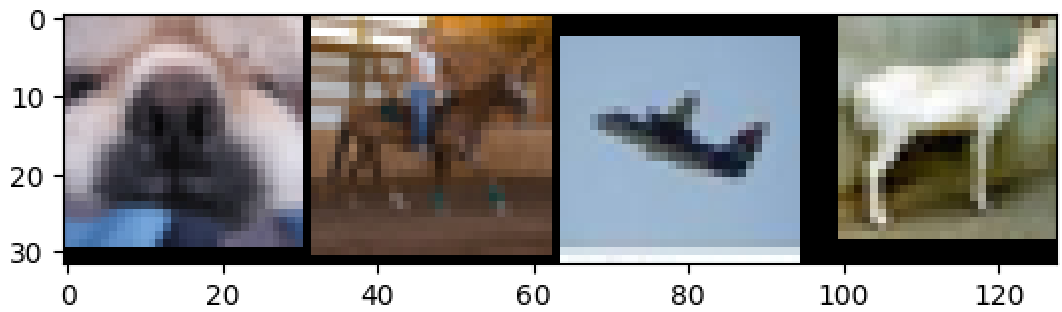
Show some example images and rotated images with labels:

In [ ]:

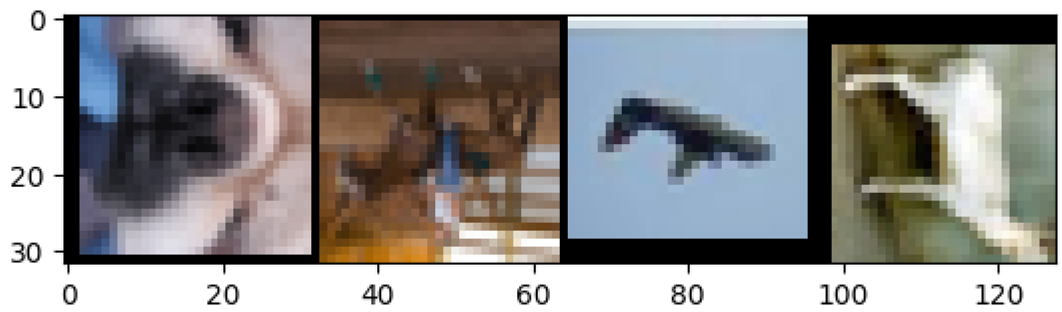
```
import matplotlib.pyplot as plt

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))(im
    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
```

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



```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow
with RGB data ([0..1] for floats or [0..255] for integers).
Class labels:  dog   horse plane deer
```



```
Rotation labels:  270   180   180   270
```

Evaluation code

In [ ]:
```
device = 'cuda'
```

In [ ]:
```
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
    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 predict
            #################################################################
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            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
    print(f'Average loss on the 10000 test images: {avg_test_loss:.3f}')
```

In [ ]:

```
def adjust_learning_rate(optimizer, epoch, init_lr, decay_epochs=30):
    """Sets the learning rate to the initial LR decayed by 10 every 30 epoch
    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.

In [ ]:

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

Out[ ]:

```
'cuda'
```

Notice: You should not use pretrained weights from ImageNet.

In [ ]:

```
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_runnin
g_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_ru
nning_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_ru
nning_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_ru
nning_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_ru
nning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
unning_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_r
unning_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_r
unning_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=4, bias=True)
)
```

In [ ]:

```
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(params=net.parameters(), lr=0.001)
################################################################################
#                               End of your code
################################################################################
criterion = criterion.to(device)
```

In [ ]:
```
device
```

Out[ ]:
```
'cuda'
```

In [ ]:
```
# Both the self-supervised rotation task and supervised CIFAR10 classificati
# trained with the CrossEntropyLoss, so we can use the training loop code.

def train(net, criterion, optimizer, num_epochs, decay_epochs, init_lr, tasl

    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
            adjust_learning_rate(optimizer, epoch, init_lr, decay_epochs)
            ############################################################################
            # TODO: Set the data to the correct device; Different task will
            # TODO: Zero the parameter gradients
            # TODO: forward + backward + optimize
            # TODO: Get predicted results
            ############################################################################
            if task == 'rotation':
                inputs, labels = imgs_rotated.to(device), rotation_label.to
            elif task == 'classification':
                inputs, labels = imgs.to(device), 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 mi
                print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / pri
                running_loss, running_correct, running_total = 0.0, 0.0, 0.
                start_time = time.time()
        ###################################################################
        # TODO: Run the run_test() function after each epoch; Set the model
        ###################################################################
        net.eval()
        run_test(net, testloader, criterion, task)
        net.train()
        ###################################################################
        #                               End of your code
        ###################################################################

    print('Finished Training')
```

In [ ]:

```
train(net, criterion, optimizer, num_epochs=45, decay_epochs=15, init_lr=0.
################################
#     TODO: Save the model     #
################################
torch.save(net.state_dict(), "best_model.pth")
################################
#      End of your code        #
################################
```

```
[1,   100] loss: 1.316 acc: 47.24 time: 10.08
[1,   200] loss: 1.002 acc: 56.99 time: 9.40
[1,   300] loss: 0.940 acc: 60.54 time: 8.98
TESTING:
Accuracy of the network on the 10000 test images: 62.62 %
Average loss on the 10000 test images: 0.900
[2,   100] loss: 0.867 acc: 64.29 time: 8.79
[2,   200] loss: 0.836 acc: 65.74 time: 9.68
[2,   300] loss: 0.827 acc: 66.30 time: 11.31
TESTING:
Accuracy of the network on the 10000 test images: 66.52 %
Average loss on the 10000 test images: 0.820
[3,   100] loss: 0.785 acc: 67.85 time: 10.76
[3,   200] loss: 0.774 acc: 68.52 time: 7.75
[3,   300] loss: 0.771 acc: 68.79 time: 10.48
TESTING:
Accuracy of the network on the 10000 test images: 70.21 %
Average loss on the 10000 test images: 0.747
[4,   100] loss: 0.744 acc: 70.12 time: 8.06
[4,   200] loss: 0.739 acc: 70.74 time: 10.68
[4,   300] loss: 0.735 acc: 70.42 time: 7.76
TESTING:
Accuracy of the network on the 10000 test images: 73.04 %
Average loss on the 10000 test images: 0.677
[5,   100] loss: 0.721 acc: 71.05 time: 10.81
[5,   200] loss: 0.717 acc: 71.38 time: 7.95
[5,   300] loss: 0.690 acc: 72.61 time: 10.60
TESTING:
Accuracy of the network on the 10000 test images: 72.78 %
Average loss on the 10000 test images: 0.684
[6,   100] loss: 0.690 acc: 72.90 time: 9.52
[6,   200] loss: 0.700 acc: 72.06 time: 9.72
[6,   300] loss: 0.688 acc: 72.56 time: 8.80
TESTING:
Accuracy of the network on the 10000 test images: 73.21 %
Average loss on the 10000 test images: 0.672
[7,   100] loss: 0.669 acc: 73.45 time: 8.66
[7,   200] loss: 0.670 acc: 73.90 time: 9.78
[7,   300] loss: 0.651 acc: 74.34 time: 8.86
TESTING:
Accuracy of the network on the 10000 test images: 73.91 %
Average loss on the 10000 test images: 0.661
[8,   100] loss: 0.668 acc: 73.71 time: 10.88
[8,   200] loss: 0.653 acc: 74.21 time: 7.61
[8,   300] loss: 0.651 acc: 74.55 time: 10.48
TESTING:
Accuracy of the network on the 10000 test images: 75.51 %
Average loss on the 10000 test images: 0.632
[9,   100] loss: 0.647 acc: 74.66 time: 11.69
[9,   200] loss: 0.653 acc: 74.01 time: 9.63
[9,   300] loss: 0.644 acc: 74.39 time: 9.28
TESTING:
Accuracy of the network on the 10000 test images: 75.21 %
Average loss on the 10000 test images: 0.626
[10,   100] loss: 0.608 acc: 76.12 time: 10.79
[10,   200] loss: 0.643 acc: 74.96 time: 7.84
[10,   300] loss: 0.636 acc: 75.20 time: 10.27
TESTING:
Accuracy of the network on the 10000 test images: 76.71 %
Average loss on the 10000 test images: 0.600
[11,   100] loss: 0.622 acc: 75.43 time: 7.86
[11,   200] loss: 0.610 acc: 76.09 time: 10.38
[11,   300] loss: 0.618 acc: 76.09 time: 7.75
TESTING:
```

```
Accuracy of the network on the 10000 test images: 76.13 %
Average loss on the 10000 test images: 0.603
[12,   100] loss: 0.597 acc: 76.43 time: 10.38
[12,   200] loss: 0.613 acc: 76.03 time: 7.90
[12,   300] loss: 0.634 acc: 75.34 time: 10.47
TESTING:
Accuracy of the network on the 10000 test images: 76.72 %
Average loss on the 10000 test images: 0.590
[13,   100] loss: 0.599 acc: 76.67 time: 9.40
[13,   200] loss: 0.588 acc: 77.35 time: 9.88
[13,   300] loss: 0.609 acc: 76.41 time: 8.37
TESTING:
Accuracy of the network on the 10000 test images: 78.20 %
Average loss on the 10000 test images: 0.565
[14,   100] loss: 0.576 acc: 77.38 time: 9.12
[14,   200] loss: 0.583 acc: 77.26 time: 9.44
[14,   300] loss: 0.595 acc: 76.98 time: 9.15
TESTING:
Accuracy of the network on the 10000 test images: 77.82 %
Average loss on the 10000 test images: 0.567
[15,   100] loss: 0.574 acc: 77.66 time: 10.74
[15,   200] loss: 0.570 acc: 78.08 time: 8.14
[15,   300] loss: 0.626 acc: 75.72 time: 13.37
TESTING:
Accuracy of the network on the 10000 test images: 77.49 %
Average loss on the 10000 test images: 0.583
[16,   100] loss: 0.540 acc: 79.65 time: 8.07
[16,   200] loss: 0.516 acc: 79.99 time: 10.46
[16,   300] loss: 0.523 acc: 79.57 time: 7.79
TESTING:
Accuracy of the network on the 10000 test images: 80.74 %
Average loss on the 10000 test images: 0.502
[17,   100] loss: 0.505 acc: 80.60 time: 9.47
[17,   200] loss: 0.507 acc: 80.54 time: 9.13
[17,   300] loss: 0.500 acc: 80.82 time: 10.02
TESTING:
Accuracy of the network on the 10000 test images: 80.70 %
Average loss on the 10000 test images: 0.494
[18,   100] loss: 0.492 acc: 81.12 time: 10.27
[18,   200] loss: 0.489 acc: 81.34 time: 8.50
[18,   300] loss: 0.485 acc: 81.27 time: 9.33
TESTING:
Accuracy of the network on the 10000 test images: 81.31 %
Average loss on the 10000 test images: 0.482
[19,   100] loss: 0.485 acc: 81.25 time: 7.98
[19,   200] loss: 0.484 acc: 81.27 time: 10.39
[19,   300] loss: 0.487 acc: 80.95 time: 7.92
TESTING:
Accuracy of the network on the 10000 test images: 81.63 %
Average loss on the 10000 test images: 0.479
[20,   100] loss: 0.488 acc: 81.63 time: 10.77
[20,   200] loss: 0.482 acc: 81.65 time: 7.69
[20,   300] loss: 0.475 acc: 81.50 time: 10.40
TESTING:
Accuracy of the network on the 10000 test images: 81.48 %
Average loss on the 10000 test images: 0.480
[21,   100] loss: 0.475 acc: 81.95 time: 8.68
[21,   200] loss: 0.473 acc: 81.96 time: 10.23
[21,   300] loss: 0.474 acc: 81.79 time: 7.80
TESTING:
Accuracy of the network on the 10000 test images: 81.38 %
Average loss on the 10000 test images: 0.479
[22,   100] loss: 0.475 acc: 81.80 time: 12.38
[22,   200] loss: 0.465 acc: 82.20 time: 7.73
```

```
[22,   300] loss: 0.478 acc: 81.61 time: 10.51
TESTING:
Accuracy of the network on the 10000 test images: 81.69 %
Average loss on the 10000 test images: 0.480
[23,   100] loss: 0.468 acc: 82.12 time: 8.87
[23,   200] loss: 0.481 acc: 81.55 time: 10.27
[23,   300] loss: 0.468 acc: 82.20 time: 7.56
TESTING:
Accuracy of the network on the 10000 test images: 81.74 %
Average loss on the 10000 test images: 0.469
[24,   100] loss: 0.466 acc: 82.05 time: 9.46
[24,   200] loss: 0.464 acc: 82.57 time: 8.90
[24,   300] loss: 0.463 acc: 82.40 time: 9.88
TESTING:
Accuracy of the network on the 10000 test images: 82.07 %
Average loss on the 10000 test images: 0.466
[25,   100] loss: 0.463 acc: 82.16 time: 10.32
[25,   200] loss: 0.460 acc: 82.38 time: 8.65
[25,   300] loss: 0.467 acc: 82.02 time: 9.59
TESTING:
Accuracy of the network on the 10000 test images: 81.87 %
Average loss on the 10000 test images: 0.471
[26,   100] loss: 0.466 acc: 82.23 time: 8.25
[26,   200] loss: 0.465 acc: 81.91 time: 10.25
[26,   300] loss: 0.454 acc: 82.59 time: 8.13
TESTING:
Accuracy of the network on the 10000 test images: 81.94 %
Average loss on the 10000 test images: 0.465
[27,   100] loss: 0.462 acc: 82.46 time: 10.67
[27,   200] loss: 0.457 acc: 82.23 time: 7.70
[27,   300] loss: 0.434 acc: 83.73 time: 11.01
TESTING:
Accuracy of the network on the 10000 test images: 82.23 %
Average loss on the 10000 test images: 0.456
[28,   100] loss: 0.454 acc: 82.93 time: 8.27
[28,   200] loss: 0.452 acc: 82.60 time: 10.60
[28,   300] loss: 0.452 acc: 82.77 time: 10.85
TESTING:
Accuracy of the network on the 10000 test images: 82.35 %
Average loss on the 10000 test images: 0.461
[29,   100] loss: 0.446 acc: 83.03 time: 10.92
[29,   200] loss: 0.451 acc: 82.82 time: 7.82
[29,   300] loss: 0.440 acc: 83.09 time: 10.36
TESTING:
Accuracy of the network on the 10000 test images: 82.37 %
Average loss on the 10000 test images: 0.454
[30,   100] loss: 0.446 acc: 83.27 time: 8.31
[30,   200] loss: 0.459 acc: 81.92 time: 10.55
[30,   300] loss: 0.441 acc: 83.06 time: 7.91
TESTING:
Accuracy of the network on the 10000 test images: 82.39 %
Average loss on the 10000 test images: 0.458
[31,   100] loss: 0.436 acc: 83.50 time: 9.93
[31,   200] loss: 0.441 acc: 83.07 time: 8.48
[31,   300] loss: 0.455 acc: 82.20 time: 10.23
TESTING:
Accuracy of the network on the 10000 test images: 82.61 %
Average loss on the 10000 test images: 0.452
[32,   100] loss: 0.441 acc: 82.99 time: 9.69
[32,   200] loss: 0.438 acc: 82.97 time: 9.03
[32,   300] loss: 0.435 acc: 83.67 time: 8.88
TESTING:
Accuracy of the network on the 10000 test images: 82.38 %
Average loss on the 10000 test images: 0.458
```

```
[33,   100] loss: 0.443 acc: 83.11 time: 8.21
[33,   200] loss: 0.441 acc: 82.98 time: 10.16
[33,   300] loss: 0.440 acc: 83.02 time: 9.13
TESTING:
Accuracy of the network on the 10000 test images: 82.50 %
Average loss on the 10000 test images: 0.451
[34,   100] loss: 0.448 acc: 83.25 time: 10.86
[34,   200] loss: 0.436 acc: 83.06 time: 7.83
[34,   300] loss: 0.433 acc: 83.30 time: 10.32
TESTING:
Accuracy of the network on the 10000 test images: 82.72 %
Average loss on the 10000 test images: 0.446
[35,   100] loss: 0.456 acc: 82.46 time: 8.64
[35,   200] loss: 0.438 acc: 83.01 time: 9.89
[35,   300] loss: 0.425 acc: 83.83 time: 8.69
TESTING:
Accuracy of the network on the 10000 test images: 82.59 %
Average loss on the 10000 test images: 0.454
[36,   100] loss: 0.438 acc: 83.30 time: 9.89
[36,   200] loss: 0.438 acc: 83.36 time: 8.19
[36,   300] loss: 0.440 acc: 83.16 time: 9.17
TESTING:
Accuracy of the network on the 10000 test images: 82.83 %
Average loss on the 10000 test images: 0.451
[37,   100] loss: 0.435 acc: 83.66 time: 8.39
[37,   200] loss: 0.427 acc: 83.72 time: 9.21
[37,   300] loss: 0.438 acc: 83.47 time: 9.08
TESTING:
Accuracy of the network on the 10000 test images: 82.79 %
Average loss on the 10000 test images: 0.445
[38,   100] loss: 0.442 acc: 83.13 time: 9.42
[38,   200] loss: 0.437 acc: 83.23 time: 9.34
[38,   300] loss: 0.428 acc: 83.47 time: 8.18
TESTING:
Accuracy of the network on the 10000 test images: 82.42 %
Average loss on the 10000 test images: 0.451
[39,   100] loss: 0.440 acc: 83.09 time: 9.39
[39,   200] loss: 0.442 acc: 83.17 time: 8.05
[39,   300] loss: 0.443 acc: 83.08 time: 9.67
TESTING:
Accuracy of the network on the 10000 test images: 82.51 %
Average loss on the 10000 test images: 0.452
[40,   100] loss: 0.427 acc: 83.54 time: 8.62
[40,   200] loss: 0.443 acc: 82.91 time: 9.88
[40,   300] loss: 0.445 acc: 82.78 time: 7.52
TESTING:
Accuracy of the network on the 10000 test images: 82.59 %
Average loss on the 10000 test images: 0.451
[41,   100] loss: 0.442 acc: 83.16 time: 9.89
[41,   200] loss: 0.440 acc: 83.20 time: 7.42
[41,   300] loss: 0.428 acc: 83.35 time: 10.69
TESTING:
Accuracy of the network on the 10000 test images: 82.60 %
Average loss on the 10000 test images: 0.445
[42,   100] loss: 0.431 acc: 83.48 time: 7.74
[42,   200] loss: 0.440 acc: 83.20 time: 9.92
[42,   300] loss: 0.417 acc: 84.21 time: 7.83
TESTING:
Accuracy of the network on the 10000 test images: 83.04 %
Average loss on the 10000 test images: 0.446
[43,   100] loss: 0.425 acc: 83.98 time: 10.03
[43,   200] loss: 0.430 acc: 83.62 time: 7.95
[43,   300] loss: 0.435 acc: 83.30 time: 9.41
TESTING:
```

```
Accuracy of the network on the 10000 test images: 82.41 %
Average loss on the 10000 test images: 0.449
[44,   100] loss: 0.438 acc: 83.30 time: 7.74
[44,   200] loss: 0.432 acc: 83.34 time: 9.60
[44,   300] loss: 0.428 acc: 83.48 time: 8.69
TESTING:
Accuracy of the network on the 10000 test images: 82.77 %
Average loss on the 10000 test images: 0.444
[45,   100] loss: 0.429 acc: 83.37 time: 9.92
[45,   200] loss: 0.440 acc: 82.95 time: 8.90
[45,   300] loss: 0.435 acc: 83.33 time: 8.64
TESTING:
Accuracy of the network on the 10000 test images: 82.82 %
Average loss on the 10000 test images: 0.451
Finished Training
```

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.

In [ ]:

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

from torchvision.models import resnet18

#####################################################
#     TODO: Load the pre-trained ResNet18 model     #
#####################################################
checkpoint = torch.load('/content/drive/MyDrive/Colab Notebooks/best_model.p
net.load_state_dict(checkpoint)
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_runnin
g_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_ru
nning_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_ru
nning_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_ru
nning_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_ru
nning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
unning_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_r
unning_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_r
unning_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=4, bias=True)
)
```

In [ ]:

```
###########################################################################
#   TODO: Freeze all previous layers; only keep the 'layer4' block and 'fc'
###########################################################################
```

```
for name, param in net.named_parameters():
    if 'layer4' in name or 'fc' in name:
        param.requires_grad = True
    else:
        param.requires_grad = False
num_classes = 10
net.fc = nn.Linear(net.fc.in_features, num_classes).to(device)
############################################################################
#                                           End of your code
############################################################################
```

In [ ]:
```
# 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
```

In [ ]:
```
# TODO: Define criterion and optimizer
# Note that your optimizer only needs to update the parameters that are tra
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(params_to_update, lr=0.001)
```

In [ ]:
```
train(net, criterion, optimizer, num_epochs=20, decay_epochs=10, init_lr=0.(
```

```
[1,   100] loss: 1.388 acc: 49.67 time: 7.99
[1,   200] loss: 1.037 acc: 62.31 time: 8.67
[1,   300] loss: 0.982 acc: 65.05 time: 8.19
TESTING:
Accuracy of the network on the 10000 test images: 67.27 %
Average loss on the 10000 test images: 0.922
[2,   100] loss: 0.930 acc: 66.17 time: 9.46
[2,   200] loss: 0.944 acc: 66.49 time: 7.49
[2,   300] loss: 0.905 acc: 67.53 time: 9.12
TESTING:
Accuracy of the network on the 10000 test images: 69.21 %
Average loss on the 10000 test images: 0.863
[3,   100] loss: 0.906 acc: 67.53 time: 7.33
[3,   200] loss: 0.891 acc: 68.15 time: 10.58
[3,   300] loss: 0.892 acc: 67.94 time: 7.92
TESTING:
Accuracy of the network on the 10000 test images: 69.63 %
Average loss on the 10000 test images: 0.855
[4,   100] loss: 0.888 acc: 68.34 time: 9.34
[4,   200] loss: 0.879 acc: 68.28 time: 7.11
[4,   300] loss: 0.877 acc: 68.09 time: 9.27
TESTING:
Accuracy of the network on the 10000 test images: 67.07 %
Average loss on the 10000 test images: 1.001
[5,   100] loss: 0.860 acc: 68.88 time: 8.23
[5,   200] loss: 0.860 acc: 69.48 time: 8.79
[5,   300] loss: 0.869 acc: 68.80 time: 7.51
TESTING:
Accuracy of the network on the 10000 test images: 70.67 %
Average loss on the 10000 test images: 0.862
[6,   100] loss: 0.838 acc: 69.96 time: 7.76
[6,   200] loss: 0.859 acc: 69.39 time: 9.01
[6,   300] loss: 0.849 acc: 69.10 time: 7.75
TESTING:
Accuracy of the network on the 10000 test images: 71.60 %
Average loss on the 10000 test images: 0.804
[7,   100] loss: 0.843 acc: 70.00 time: 9.53
[7,   200] loss: 0.847 acc: 69.41 time: 7.23
[7,   300] loss: 0.840 acc: 69.82 time: 9.27
TESTING:
Accuracy of the network on the 10000 test images: 71.21 %
Average loss on the 10000 test images: 0.812
[8,   100] loss: 0.838 acc: 69.80 time: 7.37
[8,   200] loss: 0.831 acc: 70.20 time: 9.22
[8,   300] loss: 0.839 acc: 70.16 time: 7.12
TESTING:
Accuracy of the network on the 10000 test images: 71.60 %
Average loss on the 10000 test images: 0.808
[9,   100] loss: 0.838 acc: 69.94 time: 9.18
[9,   200] loss: 0.822 acc: 70.40 time: 7.34
[9,   300] loss: 0.833 acc: 70.04 time: 9.32
TESTING:
Accuracy of the network on the 10000 test images: 71.69 %
Average loss on the 10000 test images: 0.798
[10,   100] loss: 0.833 acc: 69.95 time: 8.69
[10,   200] loss: 0.831 acc: 70.54 time: 8.64
[10,   300] loss: 0.821 acc: 70.74 time: 10.05
TESTING:
Accuracy of the network on the 10000 test images: 71.04 %
Average loss on the 10000 test images: 0.821
[11,   100] loss: 0.790 acc: 71.99 time: 7.30
[11,   200] loss: 0.798 acc: 71.23 time: 9.12
[11,   300] loss: 0.789 acc: 71.77 time: 7.35
TESTING:
```

```
Accuracy of the network on the 10000 test images: 72.98 %
Average loss on the 10000 test images: 0.765
[12,   100] loss: 0.798 acc: 71.22 time: 9.37
[12,   200] loss: 0.775 acc: 72.30 time: 7.26
[12,   300] loss: 0.778 acc: 71.69 time: 9.25
TESTING:
Accuracy of the network on the 10000 test images: 72.83 %
Average loss on the 10000 test images: 0.767
[13,   100] loss: 0.771 acc: 72.45 time: 7.84
[13,   200] loss: 0.768 acc: 72.08 time: 9.11
[13,   300] loss: 0.793 acc: 71.40 time: 7.02
TESTING:
Accuracy of the network on the 10000 test images: 73.04 %
Average loss on the 10000 test images: 0.760
[14,   100] loss: 0.779 acc: 72.06 time: 8.17
[14,   200] loss: 0.768 acc: 72.45 time: 8.41
[14,   300] loss: 0.783 acc: 71.77 time: 8.48
TESTING:
Accuracy of the network on the 10000 test images: 73.17 %
Average loss on the 10000 test images: 0.764
[15,   100] loss: 0.784 acc: 71.93 time: 9.34
[15,   200] loss: 0.776 acc: 72.40 time: 7.13
[15,   300] loss: 0.765 acc: 72.10 time: 9.24
TESTING:
Accuracy of the network on the 10000 test images: 73.10 %
Average loss on the 10000 test images: 0.760
[16,   100] loss: 0.759 acc: 72.68 time: 7.33
[16,   200] loss: 0.785 acc: 71.78 time: 9.25
[16,   300] loss: 0.778 acc: 72.00 time: 7.00
TESTING:
Accuracy of the network on the 10000 test images: 73.07 %
Average loss on the 10000 test images: 0.759
[17,   100] loss: 0.755 acc: 73.51 time: 8.62
[17,   200] loss: 0.773 acc: 71.91 time: 9.53
[17,   300] loss: 0.772 acc: 72.45 time: 9.59
TESTING:
Accuracy of the network on the 10000 test images: 73.25 %
Average loss on the 10000 test images: 0.759
[18,   100] loss: 0.761 acc: 72.53 time: 7.94
[18,   200] loss: 0.772 acc: 72.33 time: 8.78
[18,   300] loss: 0.765 acc: 72.34 time: 7.43
TESTING:
Accuracy of the network on the 10000 test images: 73.01 %
Average loss on the 10000 test images: 0.759
[19,   100] loss: 0.768 acc: 72.75 time: 8.20
[19,   200] loss: 0.774 acc: 72.02 time: 8.38
[19,   300] loss: 0.757 acc: 73.15 time: 8.27
TESTING:
Accuracy of the network on the 10000 test images: 73.00 %
Average loss on the 10000 test images: 0.763
[20,   100] loss: 0.756 acc: 73.01 time: 9.63
[20,   200] loss: 0.760 acc: 72.48 time: 7.16
[20,   300] loss: 0.783 acc: 71.65 time: 9.21
TESTING:
Accuracy of the network on the 10000 test images: 73.53 %
Average loss on the 10000 test images: 0.752
Finished Training
```

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.

In [ ]:
```
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, num_classes=10)
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_runnin
g_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_ru
nning_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_ru
nning_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_ru
nning_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_ru
nning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
unning_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_r
unning_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_r
unning_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

In [ ]:

```
#############################################################################
# TODO: Freeze all previous layers; only keep the 'layer4' block and 'fc' la
# To do this, you should set requires_grad=False for the frozen layers.
```

```
################################################################################
for name, param in net.named_parameters():
    if 'layer4' in name or 'fc' in name:
        param.requires_grad = True
    else:
        param.requires_grad = False
################################################################################
#                                          End of your code
################################################################################
```

In [ ]:
```
# 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
```

In [ ]:
```
# TODO: Define criterion and optimizer
# Note that your optimizer only needs to update the parameters that are tra
criterion = nn.CrossEntropyLoss()
criterion = criterion
optimizer = optim.Adam(params_to_update, lr=0.0001, weight_decay=1e-4)
```

In [ ]:
```
train(net, criterion, optimizer, num_epochs=20, decay_epochs=10, init_lr=0.(
```

```
[1,   100] loss: 1.781 acc: 35.27 time: 9.35
[1,   200] loss: 1.753 acc: 36.62 time: 9.71
[1,   300] loss: 1.775 acc: 36.11 time: 7.91
TESTING:
Accuracy of the network on the 10000 test images: 37.89 %
Average loss on the 10000 test images: 1.733
[2,   100] loss: 1.763 acc: 35.74 time: 12.20
[2,   200] loss: 1.770 acc: 35.77 time: 9.85
[2,   300] loss: 1.772 acc: 36.03 time: 7.88
TESTING:
Accuracy of the network on the 10000 test images: 39.10 %
Average loss on the 10000 test images: 1.703
[3,   100] loss: 1.761 acc: 35.84 time: 10.29
[3,   200] loss: 1.745 acc: 37.23 time: 7.93
[3,   300] loss: 1.757 acc: 36.44 time: 9.68
TESTING:
Accuracy of the network on the 10000 test images: 39.46 %
Average loss on the 10000 test images: 1.699
[4,   100] loss: 1.757 acc: 36.53 time: 8.83
[4,   200] loss: 1.754 acc: 36.52 time: 8.86
[4,   300] loss: 1.754 acc: 36.78 time: 9.84
TESTING:
Accuracy of the network on the 10000 test images: 26.52 %
Average loss on the 10000 test images: 3.188
[5,   100] loss: 1.741 acc: 37.79 time: 7.86
[5,   200] loss: 1.744 acc: 36.77 time: 9.60
[5,   300] loss: 1.750 acc: 36.38 time: 8.15
TESTING:
Accuracy of the network on the 10000 test images: 37.65 %
Average loss on the 10000 test images: 1.707
[6,   100] loss: 1.738 acc: 37.30 time: 9.96
[6,   200] loss: 1.758 acc: 36.55 time: 8.93
[6,   300] loss: 1.758 acc: 36.52 time: 8.38
TESTING:
Accuracy of the network on the 10000 test images: 40.35 %
Average loss on the 10000 test images: 1.664
[7,   100] loss: 1.750 acc: 37.09 time: 10.05
[7,   200] loss: 1.743 acc: 36.70 time: 8.09
[7,   300] loss: 1.757 acc: 36.13 time: 10.36
TESTING:
Accuracy of the network on the 10000 test images: 39.34 %
Average loss on the 10000 test images: 1.686
[8,   100] loss: 1.749 acc: 36.44 time: 8.66
[8,   200] loss: 1.752 acc: 36.51 time: 9.15
[8,   300] loss: 1.742 acc: 36.84 time: 9.76
TESTING:
Accuracy of the network on the 10000 test images: 39.50 %
Average loss on the 10000 test images: 1.686
[9,   100] loss: 1.750 acc: 36.56 time: 8.48
[9,   200] loss: 1.745 acc: 36.79 time: 9.90
[9,   300] loss: 1.748 acc: 37.14 time: 8.30
TESTING:
Accuracy of the network on the 10000 test images: 38.81 %
Average loss on the 10000 test images: 1.690
[10,   100] loss: 1.753 acc: 36.93 time: 10.10
[10,   200] loss: 1.744 acc: 36.82 time: 9.19
[10,   300] loss: 1.745 acc: 36.95 time: 8.64
TESTING:
Accuracy of the network on the 10000 test images: 39.34 %
Average loss on the 10000 test images: 1.692
[11,   100] loss: 1.732 acc: 37.05 time: 9.88
[11,   200] loss: 1.705 acc: 38.59 time: 7.96
[11,   300] loss: 1.698 acc: 38.81 time: 9.88
TESTING:
```

```
Accuracy of the network on the 10000 test images: 42.45 %
Average loss on the 10000 test images: 1.624
[12,   100] loss: 1.701 acc: 38.62 time: 8.42
[12,   200] loss: 1.683 acc: 39.20 time: 9.55
[12,   300] loss: 1.675 acc: 39.98 time: 9.65
TESTING:
Accuracy of the network on the 10000 test images: 42.43 %
Average loss on the 10000 test images: 1.613
[13,   100] loss: 1.685 acc: 38.91 time: 8.86
[13,   200] loss: 1.688 acc: 38.85 time: 9.84
[13,   300] loss: 1.684 acc: 39.05 time: 8.02
TESTING:
Accuracy of the network on the 10000 test images: 42.81 %
Average loss on the 10000 test images: 1.606
[14,   100] loss: 1.660 acc: 39.89 time: 9.45
[14,   200] loss: 1.679 acc: 39.48 time: 9.09
[14,   300] loss: 1.660 acc: 40.56 time: 8.03
TESTING:
Accuracy of the network on the 10000 test images: 43.28 %
Average loss on the 10000 test images: 1.599
[15,   100] loss: 1.666 acc: 40.38 time: 9.43
[15,   200] loss: 1.674 acc: 39.94 time: 8.86
[15,   300] loss: 1.656 acc: 40.68 time: 8.10
TESTING:
Accuracy of the network on the 10000 test images: 43.21 %
Average loss on the 10000 test images: 1.595
[16,   100] loss: 1.658 acc: 40.31 time: 9.28
[16,   200] loss: 1.646 acc: 41.02 time: 8.69
[16,   300] loss: 1.653 acc: 41.21 time: 8.45
TESTING:
Accuracy of the network on the 10000 test images: 43.69 %
Average loss on the 10000 test images: 1.593
[17,   100] loss: 1.643 acc: 40.76 time: 9.24
[17,   200] loss: 1.660 acc: 40.37 time: 8.16
[17,   300] loss: 1.652 acc: 40.49 time: 8.80
TESTING:
Accuracy of the network on the 10000 test images: 43.19 %
Average loss on the 10000 test images: 1.589
[18,   100] loss: 1.642 acc: 40.81 time: 9.25
[18,   200] loss: 1.653 acc: 40.71 time: 7.83
[18,   300] loss: 1.637 acc: 41.11 time: 9.04
TESTING:
Accuracy of the network on the 10000 test images: 43.74 %
Average loss on the 10000 test images: 1.583
[19,   100] loss: 1.648 acc: 40.49 time: 9.24
[19,   200] loss: 1.665 acc: 40.53 time: 7.72
[19,   300] loss: 1.635 acc: 41.73 time: 9.10
TESTING:
Accuracy of the network on the 10000 test images: 43.78 %
Average loss on the 10000 test images: 1.572
[20,   100] loss: 1.627 acc: 41.20 time: 9.15
[20,   200] loss: 1.632 acc: 40.95 time: 8.08
[20,   300] loss: 1.640 acc: 40.91 time: 9.02
TESTING:
Accuracy of the network on the 10000 test images: 44.07 %
Average loss on the 10000 test images: 1.581
Finished Training
```

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.

In [ ]:
```
import torch.nn as nn
import torch.nn.functional as F

from torchvision.models import resnet18

#####################################################
#     TODO: Load the pre-trained ResNet18 model     #
#####################################################
checkpoint = torch.load('/content/drive/MyDrive/Colab Notebooks/best_model.
net.load_state_dict(checkpoint)
net.fc = nn.Linear(net.fc.in_features, len(classes))
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_runnin
g_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_ru
nning_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_ru
nning_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_ru
nning_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_ru
nning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
unning_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_r
unning_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_r
unning_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

In [ ]:
```
# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

In [ ]:
```
device = 'cuda'
train(net.to(device), criterion, optimizer, num_epochs=20, decay_epochs=10,
```

```
[1,   100] loss: 1.515 acc: 43.87 time: 12.92
[1,   200] loss: 1.202 acc: 56.76 time: 12.04
[1,   300] loss: 1.088 acc: 60.96 time: 7.93
TESTING:
Accuracy of the network on the 10000 test images: 66.45 %
Average loss on the 10000 test images: 0.957
[2,   100] loss: 0.958 acc: 66.09 time: 11.62
[2,   200] loss: 0.925 acc: 67.37 time: 8.14
[2,   300] loss: 0.896 acc: 68.70 time: 11.20
TESTING:
Accuracy of the network on the 10000 test images: 69.71 %
Average loss on the 10000 test images: 0.868
[3,   100] loss: 0.831 acc: 71.18 time: 9.29
[3,   200] loss: 0.833 acc: 70.83 time: 11.32
[3,   300] loss: 0.804 acc: 72.15 time: 8.40
TESTING:
Accuracy of the network on the 10000 test images: 72.39 %
Average loss on the 10000 test images: 0.820
[4,   100] loss: 0.748 acc: 74.42 time: 11.62
[4,   200] loss: 0.734 acc: 74.37 time: 8.86
[4,   300] loss: 0.771 acc: 73.65 time: 10.83
TESTING:
Accuracy of the network on the 10000 test images: 74.28 %
Average loss on the 10000 test images: 0.754
[5,   100] loss: 0.694 acc: 75.80 time: 8.84
[5,   200] loss: 0.707 acc: 75.27 time: 11.24
[5,   300] loss: 0.686 acc: 76.16 time: 10.17
TESTING:
Accuracy of the network on the 10000 test images: 76.92 %
Average loss on the 10000 test images: 0.694
[6,   100] loss: 0.645 acc: 77.70 time: 11.18
[6,   200] loss: 0.673 acc: 76.96 time: 10.00
[6,   300] loss: 0.675 acc: 77.03 time: 9.46
TESTING:
Accuracy of the network on the 10000 test images: 76.07 %
Average loss on the 10000 test images: 0.699
[7,   100] loss: 0.619 acc: 78.73 time: 10.32
[7,   200] loss: 0.638 acc: 77.83 time: 9.76
[7,   300] loss: 0.627 acc: 78.20 time: 10.92
TESTING:
Accuracy of the network on the 10000 test images: 76.55 %
Average loss on the 10000 test images: 0.699
[8,   100] loss: 0.586 acc: 80.03 time: 10.35
[8,   200] loss: 0.590 acc: 80.06 time: 11.01
[8,   300] loss: 0.596 acc: 79.09 time: 8.75
TESTING:
Accuracy of the network on the 10000 test images: 78.42 %
Average loss on the 10000 test images: 0.619
[9,   100] loss: 0.555 acc: 80.62 time: 10.89
[9,   200] loss: 0.574 acc: 80.40 time: 8.94
[9,   300] loss: 0.563 acc: 80.40 time: 11.51
TESTING:
Accuracy of the network on the 10000 test images: 79.46 %
Average loss on the 10000 test images: 0.607
[10,   100] loss: 0.545 acc: 81.24 time: 9.37
[10,   200] loss: 0.536 acc: 81.28 time: 11.29
[10,   300] loss: 0.551 acc: 81.31 time: 8.64
TESTING:
Accuracy of the network on the 10000 test images: 79.31 %
Average loss on the 10000 test images: 0.638
[11,   100] loss: 0.473 acc: 83.52 time: 11.53
[11,   200] loss: 0.428 acc: 85.12 time: 8.36
[11,   300] loss: 0.437 acc: 85.20 time: 11.15
TESTING:
```

```
Accuracy of the network on the 10000 test images: 83.25 %
Average loss on the 10000 test images: 0.500
[12,   100] loss: 0.429 acc: 85.48 time: 8.91
[12,   200] loss: 0.425 acc: 85.43 time: 11.27
[12,   300] loss: 0.409 acc: 86.12 time: 8.43
TESTING:
Accuracy of the network on the 10000 test images: 83.30 %
Average loss on the 10000 test images: 0.503
[13,   100] loss: 0.411 acc: 85.59 time: 11.32
[13,   200] loss: 0.400 acc: 85.98 time: 8.60
[13,   300] loss: 0.404 acc: 85.95 time: 11.22
TESTING:
Accuracy of the network on the 10000 test images: 83.56 %
Average loss on the 10000 test images: 0.492
[14,   100] loss: 0.391 acc: 86.47 time: 9.22
[14,   200] loss: 0.385 acc: 86.77 time: 10.83
[14,   300] loss: 0.402 acc: 86.20 time: 9.61
TESTING:
Accuracy of the network on the 10000 test images: 83.68 %
Average loss on the 10000 test images: 0.492
[15,   100] loss: 0.390 acc: 86.37 time: 11.45
[15,   200] loss: 0.390 acc: 86.42 time: 9.21
[15,   300] loss: 0.389 acc: 86.46 time: 10.31
TESTING:
Accuracy of the network on the 10000 test images: 83.85 %
Average loss on the 10000 test images: 0.488
[16,   100] loss: 0.387 acc: 86.57 time: 9.51
[16,   200] loss: 0.381 acc: 86.53 time: 10.41
[16,   300] loss: 0.385 acc: 86.66 time: 10.44
TESTING:
Accuracy of the network on the 10000 test images: 83.68 %
Average loss on the 10000 test images: 0.491
[17,   100] loss: 0.372 acc: 87.12 time: 10.38
[17,   200] loss: 0.377 acc: 86.91 time: 10.28
[17,   300] loss: 0.379 acc: 87.11 time: 9.57
TESTING:
Accuracy of the network on the 10000 test images: 83.98 %
Average loss on the 10000 test images: 0.485
[18,   100] loss: 0.366 acc: 87.05 time: 10.67
[18,   200] loss: 0.357 acc: 87.51 time: 9.34
[18,   300] loss: 0.367 acc: 87.34 time: 11.31
TESTING:
Accuracy of the network on the 10000 test images: 83.78 %
Average loss on the 10000 test images: 0.483
[19,   100] loss: 0.356 acc: 87.31 time: 9.60
[19,   200] loss: 0.360 acc: 87.50 time: 11.34
[19,   300] loss: 0.360 acc: 87.61 time: 8.45
TESTING:
Accuracy of the network on the 10000 test images: 84.12 %
Average loss on the 10000 test images: 0.481
[20,   100] loss: 0.361 acc: 87.34 time: 11.52
[20,   200] loss: 0.343 acc: 88.01 time: 8.71
[20,   300] loss: 0.351 acc: 87.82 time: 11.24
TESTING:
Accuracy of the network on the 10000 test images: 84.05 %
Average loss on the 10000 test images: 0.483
Finished Training
```

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.

In [ ]:
```
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, num_classes=10)
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_runnin
g_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_ru
nning_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_ru
nning_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_ru
nning_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_ru
nning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_r
unning_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_r
unning_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_r
unning_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_r
unning_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_r
unning_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=Fals
e)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
unning_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_r
unning_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_r
unning_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

In [ ]:
```
# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

In [ ]:
```
train(net.to(device), criterion, optimizer, num_epochs=20, decay_epochs=10,
```

```
[1,   100] loss: 2.258 acc: 23.20 time: 8.54
[1,   200] loss: 1.853 acc: 31.73 time: 11.02
[1,   300] loss: 1.751 acc: 35.61 time: 8.62
TESTING:
Accuracy of the network on the 10000 test images: 43.55 %
Average loss on the 10000 test images: 1.553
[2,   100] loss: 1.555 acc: 43.12 time: 11.23
[2,   200] loss: 1.492 acc: 44.55 time: 8.12
[2,   300] loss: 1.425 acc: 47.93 time: 11.00
TESTING:
Accuracy of the network on the 10000 test images: 50.93 %
Average loss on the 10000 test images: 1.367
[3,   100] loss: 1.305 acc: 52.10 time: 8.70
[3,   200] loss: 1.238 acc: 55.37 time: 11.03
[3,   300] loss: 1.248 acc: 55.66 time: 11.49
TESTING:
Accuracy of the network on the 10000 test images: 57.23 %
Average loss on the 10000 test images: 1.213
[4,   100] loss: 1.144 acc: 59.27 time: 10.77
[4,   200] loss: 1.094 acc: 61.18 time: 9.65
[4,   300] loss: 1.060 acc: 62.58 time: 9.55
TESTING:
Accuracy of the network on the 10000 test images: 63.76 %
Average loss on the 10000 test images: 1.042
[5,   100] loss: 1.030 acc: 63.30 time: 10.73
[5,   200] loss: 0.995 acc: 64.96 time: 9.19
[5,   300] loss: 0.972 acc: 66.05 time: 11.10
TESTING:
Accuracy of the network on the 10000 test images: 66.68 %
Average loss on the 10000 test images: 0.973
[6,   100] loss: 0.913 acc: 67.93 time: 9.15
[6,   200] loss: 0.913 acc: 67.70 time: 11.60
[6,   300] loss: 0.896 acc: 68.66 time: 8.27
TESTING:
Accuracy of the network on the 10000 test images: 70.60 %
Average loss on the 10000 test images: 0.830
[7,   100] loss: 0.859 acc: 69.77 time: 11.21
[7,   200] loss: 0.842 acc: 70.23 time: 8.38
[7,   300] loss: 0.827 acc: 70.99 time: 11.20
TESTING:
Accuracy of the network on the 10000 test images: 71.00 %
Average loss on the 10000 test images: 0.854
[8,   100] loss: 0.800 acc: 72.46 time: 8.90
[8,   200] loss: 0.790 acc: 72.70 time: 11.20
[8,   300] loss: 0.776 acc: 72.97 time: 8.52
TESTING:
Accuracy of the network on the 10000 test images: 74.70 %
Average loss on the 10000 test images: 0.727
[9,   100] loss: 0.744 acc: 74.09 time: 11.96
[9,   200] loss: 0.746 acc: 73.88 time: 8.47
[9,   300] loss: 0.736 acc: 74.31 time: 11.34
TESTING:
Accuracy of the network on the 10000 test images: 74.50 %
Average loss on the 10000 test images: 0.746
[10,   100] loss: 0.697 acc: 76.00 time: 8.60
[10,   200] loss: 0.728 acc: 74.77 time: 11.54
[10,   300] loss: 0.701 acc: 75.77 time: 9.36
TESTING:
Accuracy of the network on the 10000 test images: 74.02 %
Average loss on the 10000 test images: 0.785
[11,   100] loss: 0.599 acc: 79.10 time: 11.66
[11,   200] loss: 0.570 acc: 80.46 time: 9.24
[11,   300] loss: 0.554 acc: 80.74 time: 10.56
TESTING:
```

```
Accuracy of the network on the 10000 test images: 79.88 %
Average loss on the 10000 test images: 0.583
[12,   100] loss: 0.544 acc: 80.97 time: 9.39
[12,   200] loss: 0.536 acc: 81.50 time: 10.63
[12,   300] loss: 0.533 acc: 81.73 time: 10.40
TESTING:
Accuracy of the network on the 10000 test images: 80.53 %
Average loss on the 10000 test images: 0.572
[13,   100] loss: 0.514 acc: 81.70 time: 10.98
[13,   200] loss: 0.522 acc: 81.76 time: 10.21
[13,   300] loss: 0.517 acc: 82.09 time: 9.29
TESTING:
Accuracy of the network on the 10000 test images: 80.71 %
Average loss on the 10000 test images: 0.562
[14,   100] loss: 0.508 acc: 82.42 time: 10.35
[14,   200] loss: 0.498 acc: 82.34 time: 9.74
[14,   300] loss: 0.502 acc: 82.68 time: 11.34
TESTING:
Accuracy of the network on the 10000 test images: 80.88 %
Average loss on the 10000 test images: 0.562
[15,   100] loss: 0.476 acc: 83.36 time: 9.97
[15,   200] loss: 0.489 acc: 83.02 time: 11.42
[15,   300] loss: 0.490 acc: 82.99 time: 9.12
TESTING:
Accuracy of the network on the 10000 test images: 81.14 %
Average loss on the 10000 test images: 0.552
[16,   100] loss: 0.483 acc: 83.16 time: 11.47
[16,   200] loss: 0.491 acc: 82.88 time: 8.13
[16,   300] loss: 0.470 acc: 83.73 time: 11.15
TESTING:
Accuracy of the network on the 10000 test images: 81.24 %
Average loss on the 10000 test images: 0.547
[17,   100] loss: 0.466 acc: 83.83 time: 8.69
[17,   200] loss: 0.478 acc: 83.19 time: 11.11
[17,   300] loss: 0.467 acc: 83.64 time: 8.63
TESTING:
Accuracy of the network on the 10000 test images: 81.59 %
Average loss on the 10000 test images: 0.539
[18,   100] loss: 0.459 acc: 84.00 time: 11.49
[18,   200] loss: 0.464 acc: 84.02 time: 8.45
[18,   300] loss: 0.452 acc: 84.11 time: 11.16
TESTING:
Accuracy of the network on the 10000 test images: 81.68 %
Average loss on the 10000 test images: 0.539
[19,   100] loss: 0.449 acc: 84.30 time: 8.64
[19,   200] loss: 0.446 acc: 84.30 time: 11.25
[19,   300] loss: 0.448 acc: 84.55 time: 9.49
TESTING:
Accuracy of the network on the 10000 test images: 81.48 %
Average loss on the 10000 test images: 0.537
[20,   100] loss: 0.430 acc: 84.91 time: 11.40
[20,   200] loss: 0.440 acc: 84.60 time: 9.96
[20,   300] loss: 0.435 acc: 84.85 time: 9.72
TESTING:
Accuracy of the network on the 10000 test images: 82.02 %
Average loss on the 10000 test images: 0.534
Finished Training
```

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).