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 陳文薇, B094020007.

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.

```
from google.colab import drive
drive.mount('/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.

```
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
    ########################################################################
```

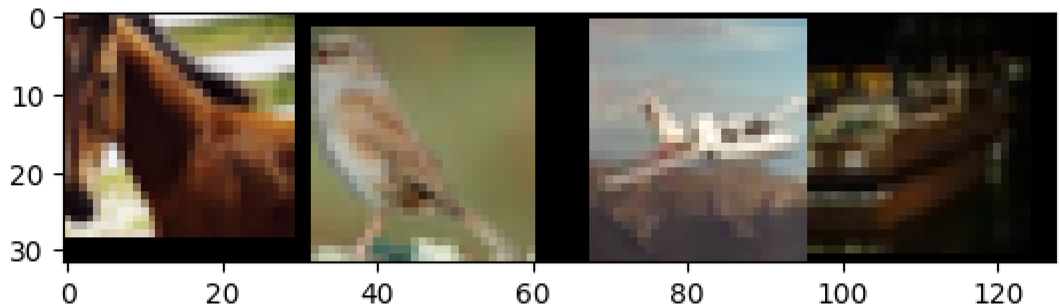
```
    #       TODO: Implement rotate_img() - return the rotated img        #
    ######################################################################
    else:
        rotated_img = transforms.functional.rotate(img, rot * 90)
        return rotated_img
    ######################################################################
    #                           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=tran
        # super().__init__()

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

    def __getitem__(self, index: int):
        image, cls_label = super(CIFAR10Rotation, self).__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=2)

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=2)
```

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

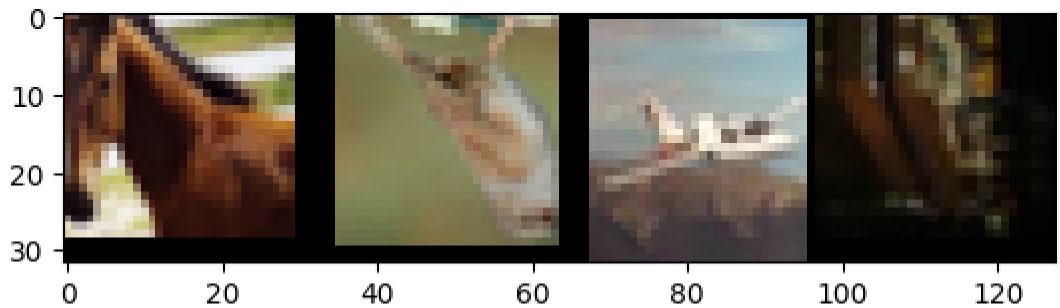
Show some example images and rotated images with labels:

```
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 ra
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Class labels:  horse bird  plane ship
```



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 o
    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)
            # loss = criterion(outputs.data, labels)
            _, predicted = torch.max(outputs, 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 / tot
    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
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runn
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ce
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=Fals
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), paddin
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
      (relu): ReLU(inplace=True)
```

```
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=Fal
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
```

```
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=1e-4, betas=(0.9, 0.999), eps=1e-08, w
################################################################################
#                               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(trainl
            adjust_learning_rate(optimizer, epoch, init_lr, decay_epochs)
            ############################################################################
            # TODO: Set the data to the correct device; Different task will use di
            # TODO: Zero the parameter gradients
            # TODO: forward + backward + optimize
            # TODO: Get predicted results
            ############################################################################
            device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
            imgs = imgs.to(device)
            imgs_rotated = imgs_rotated.to(device)
            rotation_label = rotation_label.to(device)
            cls_label = cls_label.to(device)
```

```
            if(task=='rotation'):
                labels = rotation_label
                labels = labels.to(device)
                outputs = net(imgs_rotated)#process input through the network
                loss = criterion(outputs, labels)#compute the loss
                optimizer.zero_grad()#zero the parameter gradients
                loss.backward()#propagate gradients back into the network’s parame
                optimizer.step()#Update the weights of the network

                _, predicted = torch.max(outputs.data, 1)
            else:
                labels = cls_label
                labels = labels.to(device)
                outputs = net(imgs)#process input through the network
                loss = criterion(outputs, labels)#compute the loss
                optimizer.zero_grad()#zero the parameter gradients
                loss.backward()#propagate gradients back into the network’s parame
                optimizer.step()#Update the weights of the network

                _, 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-batc
                print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / print_freq
                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
        ##########################################################################
        # def run_test():
        net.eval()
        if task == 'rotation':
            run_test(net, testloader, criterion, 'rotation')
        else:
            run_test(net, testloader, criterion, 'classification')
        ##########################################################################
        #                               End of your code
        ##########################################################################

    print('Finished Training')


train(net, criterion, optimizer, num_epochs=100, decay_epochs=15, init_lr=0.01, ta
################################
```

```
#     TODO: Save the model     #
################################
torch.save(net,'model.pt')
################################
#      End of your code        #
################################
```

```
[100,   200] loss: 0.551 acc: 78.51 time: 2.87
[100,   300] loss: 0.559 acc: 77.95 time: 2.88
TESTING:
Accuracy of the network on the 10000 test images: 78.52 %
Average loss on the 10000 test images: 0.549
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.

```
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('model.pt')
net.fc = nn.Linear(in_features=512, out_features=10, bias=True)
net = net.cuda()
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
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runn
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ce
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
```

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

```
##################################################################################
#   TODO: Freeze all previous layers; only keep the 'layer4' block and 'fc' layer
##################################################################################
for param in net.parameters():
    param.requires_grad = False

for param in net.layer4.parameters():
    param.requires_grad = True

for param in net.fc.parameters():
    param.requires_grad = True
##################################################################################
#                                            End of your code
##################################################################################
```

```
# 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_to_update, lr=1e-4)


train(net, criterion, optimizer, num_epochs=50, decay_epochs=10, init_lr=0.01, tas
```

```
[46,   100] loss: 1.093 acc: 60.88 time: 3.06
[46,   200] loss: 1.072 acc: 61.00 time: 2.88
[46,   300] loss: 1.077 acc: 61.35 time: 2.91
TESTING:
Accuracy of the network on the 10000 test images: 61.50 %
Average loss on the 10000 test images: 1.062
[47,   100] loss: 1.074 acc: 61.12 time: 3.04
[47,   200] loss: 1.077 acc: 61.38 time: 2.91
[47,   300] loss: 1.087 acc: 60.69 time: 2.95
TESTING:
Accuracy of the network on the 10000 test images: 61.61 %
Average loss on the 10000 test images: 1.066
[48,   100] loss: 1.082 acc: 60.81 time: 3.04
[48,   200] loss: 1.081 acc: 60.79 time: 2.87
[48,   300] loss: 1.087 acc: 60.75 time: 2.91
TESTING:
Accuracy of the network on the 10000 test images: 61.67 %
Average loss on the 10000 test images: 1.060
[49,   100] loss: 1.102 acc: 60.09 time: 3.05
[49,   200] loss: 1.075 acc: 60.87 time: 2.88
[49,   300] loss: 1.074 acc: 61.52 time: 2.91
TESTING:
Accuracy of the network on the 10000 test images: 61.64 %
Average loss on the 10000 test images: 1.062
[50,   100] loss: 1.092 acc: 60.15 time: 3.02
[50,   200] loss: 1.074 acc: 61.16 time: 2.93
[50,   300] loss: 1.075 acc: 61.30 time: 2.93
TESTING:
Accuracy of the network on the 10000 test images: 61.80 %
Average loss on the 10000 test images: 1.065
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.

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

from torchvision.models import resnet18
#################################################
# TODO: Randomly initialize a ResNet18 model    #
#################################################
#net = resnet18(num_classes=10, pretrained=False)
def initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                # Initialize weights using Xavier uniform distribution
                nn.init.xavier_uniform_(m.weight)
                if m.bias is not None:
                    # Initialize bias to zero
                    nn.init.constant_(m.bias, 0)

initialize_weights(net)
net = net.cuda()
```

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

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

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

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

for param in net.layer4.parameters():
    param.requires_grad = True

for param in net.fc.parameters():
    param.requires_grad = True
####################################################################################
#                                           End of your code
####################################################################################
```

```
# 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_to_update, lr=1e-4)
```

```
train(net, criterion, optimizer, num_epochs=50, decay_epochs=10, init_lr=0.01, tas
```

```
TESTING:
Accuracy of the network on the 10000 test images: 43.62 %
Average loss on the 10000 test images: 1.568
[42,   100] loss: 1.618 acc: 42.10 time: 3.01
[42,   200] loss: 1.607 acc: 41.99 time: 2.85
[42,   300] loss: 1.612 acc: 42.25 time: 2.87
TESTING:
Accuracy of the network on the 10000 test images: 43.68 %
Average loss on the 10000 test images: 1.570
[43,   100] loss: 1.619 acc: 41.61 time: 2.99
[43,   200] loss: 1.612 acc: 41.85 time: 2.89
[43,   300] loss: 1.597 acc: 42.05 time: 2.88
TESTING:
Accuracy of the network on the 10000 test images: 43.67 %
Average loss on the 10000 test images: 1.568
[44,   100] loss: 1.607 acc: 42.63 time: 3.08
[44,   200] loss: 1.618 acc: 41.83 time: 2.82
[44,   300] loss: 1.606 acc: 41.92 time: 2.90
TESTING:
Accuracy of the network on the 10000 test images: 43.85 %
Average loss on the 10000 test images: 1.570
[45,   100] loss: 1.588 acc: 43.04 time: 3.08
[45,   200] loss: 1.624 acc: 41.26 time: 2.91
[45,   300] loss: 1.607 acc: 42.07 time: 2.83
TESTING:
Accuracy of the network on the 10000 test images: 43.70 %
Average loss on the 10000 test images: 1.569
[46,   100] loss: 1.615 acc: 41.59 time: 3.07
[46,   200] loss: 1.622 acc: 42.01 time: 2.89
[46,   300] loss: 1.597 acc: 42.92 time: 2.98
TESTING:
Accuracy of the network on the 10000 test images: 43.44 %
Average loss on the 10000 test images: 1.568
[47,   100] loss: 1.615 acc: 41.64 time: 3.02
[47,   200] loss: 1.600 acc: 42.34 time: 2.87
[47,   300] loss: 1.600 acc: 42.34 time: 2.92
TESTING:
Accuracy of the network on the 10000 test images: 43.69 %
Average loss on the 10000 test images: 1.568
[48,   100] loss: 1.619 acc: 42.27 time: 3.05
[48,   200] loss: 1.611 acc: 42.31 time: 2.88
[48,   300] loss: 1.612 acc: 42.58 time: 2.90
TESTING:
Accuracy of the network on the 10000 test images: 43.57 %
Average loss on the 10000 test images: 1.570
[49,   100] loss: 1.606 acc: 42.64 time: 3.08
[49,   200] loss: 1.612 acc: 42.15 time: 2.91
[49,   300] loss: 1.620 acc: 41.96 time: 2.85
TESTING:
Accuracy of the network on the 10000 test images: 43.50 %
Average loss on the 10000 test images: 1.568
[50,   100] loss: 1.603 acc: 42.96 time: 3.01
[50,   200] loss: 1.612 acc: 41.98 time: 2.87
[50,   300] loss: 1.617 acc: 42.34 time: 2.86
TESTING:
Accuracy of the network on the 10000 test images: 43.26 %
Average loss on the 10000 test images: 1.570
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.

```
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('model.pt')
net.fc = nn.Linear(in_features=512, out_features=10, bias=True)
net = net.cuda()
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
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runn
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ce
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=Fals
```

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

```
# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=1e-4, betas=(0.9, 0.999), eps=1e-08, w
```

```
train(net, criterion, optimizer, num_epochs=50, decay_epochs=10, init_lr=0.01, tas
```

```
TESTING:
Accuracy of the network on the 10000 test images: 84.09 %
Average loss on the 10000 test images: 0.476
[46,   100] loss: 0.325 acc: 88.80 time: 3.06
[46,   200] loss: 0.322 acc: 88.74 time: 3.05
[46,   300] loss: 0.326 acc: 88.67 time: 2.99
TESTING:
Accuracy of the network on the 10000 test images: 84.29 %
Average loss on the 10000 test images: 0.480
[47,   100] loss: 0.329 acc: 88.45 time: 3.10
[47,   200] loss: 0.317 acc: 88.95 time: 2.89
[47,   300] loss: 0.332 acc: 88.33 time: 2.94
TESTING:
Accuracy of the network on the 10000 test images: 84.28 %
Average loss on the 10000 test images: 0.476
[48,   100] loss: 0.322 acc: 88.76 time: 3.16
[48,   200] loss: 0.343 acc: 88.05 time: 2.89
[48,   300] loss: 0.322 acc: 88.62 time: 2.91
TESTING:
Accuracy of the network on the 10000 test images: 84.04 %
Average loss on the 10000 test images: 0.476
[49,   100] loss: 0.314 acc: 88.98 time: 3.08
[49,   200] loss: 0.321 acc: 88.62 time: 2.90
[49,   300] loss: 0.332 acc: 88.27 time: 2.92
TESTING:
Accuracy of the network on the 10000 test images: 84.12 %
Average loss on the 10000 test images: 0.475
[50,   100] loss: 0.313 acc: 89.05 time: 3.04
[50,   200] loss: 0.329 acc: 88.19 time: 2.89
[50,   300] loss: 0.333 acc: 88.09 time: 2.91
TESTING:
Accuracy of the network on the 10000 test images: 84.04 %
Average loss on the 10000 test images: 0.476
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.

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

from torchvision.models import resnet18

##################################################
# TODO: Randomly initialize a ResNet18 model    #
##################################################
net = resnet18()
net.fc = nn.Linear(in_features=512, out_features=10, bias=True)
net = net.cuda()
for param in net.parameters():
    nn.init.normal_(param, mean=0, std=0.01)
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
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runn
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ce
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=Fals
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), paddin
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=Fal
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddin
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
      (relu): ReLU(inplace=True)
```

```
# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=1e-4, betas=(0.9, 0.999), eps=1e-08, w
```

```
train(net, criterion, optimizer, num_epochs=50, decay_epochs=10, init_lr=0.01, tas
```

```
Average loss on the 10000 test images: 0.725
[50,   100] loss: 0.730 acc: 73.90 time: 3.05
[50,   200] loss: 0.749 acc: 73.40 time: 3.01
[50,   300] loss: 0.726 acc: 74.38 time: 2.88
TESTING:
Accuracy of the network on the 10000 test images: 75.37 %
Average loss on the 10000 test images: 0.726
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).

按兩下 (或按 Enter 鍵) 即可編輯