(Optional) Colab Setup

If you aren't using Colab, you can delete the following code cell. This is just to help students with mounting to Google Drive to access the other .py files and downloading the data, which is a little trickier on Colab than on your local machine using Jupyter.

```
# you will be prompted with a window asking to grant permissions
from google.colab import drive
drive.mount("/content/drive")

    Mounted at /content/drive
```

```
# fill in the path in your Google Drive in the string below. Note: do not escape sl
import os
datadir = "/content/assignment3"
if not os.path.exists(datadir):
  !ln -s "/content/drive/My Drive/CS444/assignment3/" $datadir # TODO: Fill your A3
os.chdir(datadir)
!pwd

    /content/drive/My Drive/CS444/assignment3
```

Data Setup

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

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

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


def rotate_img(img, rot):
    if rot == 0: # 0 degrees rotation
        return img
    # TODO: Implement rotate_img() - return the rotated img
    elif rot == 1:
      return transforms.functional.rotate(img, -90)
    elif rot == 2:
      return transforms.functional.rotate(img, -180)
    elif rot == 3:
      return transforms.functional.rotate(img, -270)
    else:
        raise ValueError('rotation should be 0, 90, 180, or 270 degrees')


class CIFAR10Rotation(torchvision.datasets.CIFAR10):

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

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

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

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

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

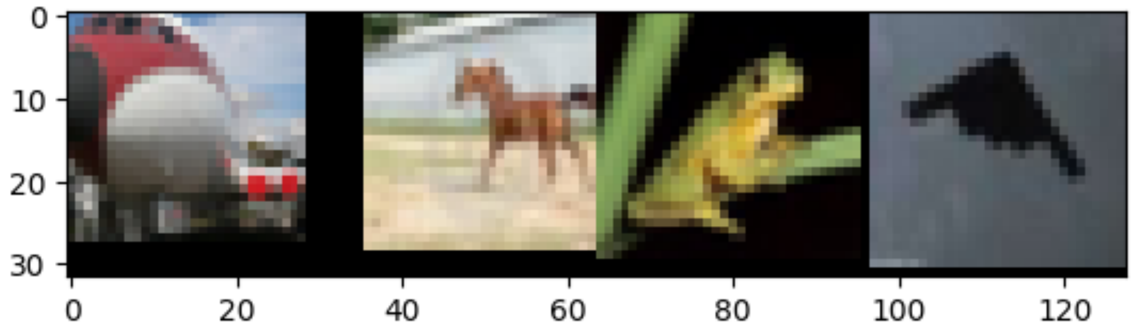
```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

batch_size = 128

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

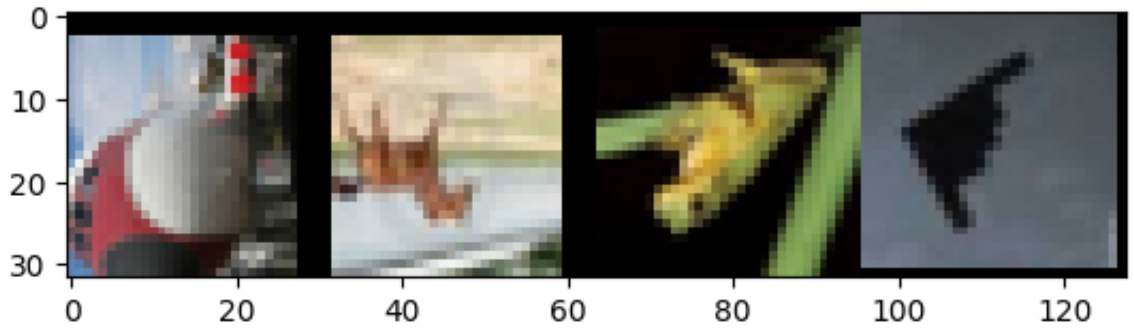
Show some example images and rotated images with labels:

```
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 rar
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
```



```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Class labels:  plane horse frog  plane
```



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

Evaluation code

```
import time

def run_test(net, testloader, criterion, task):
    correct = 0
    total = 0
    avg_test_loss = 0.0
    # since we're not training, we don't need to calculate the gradients for our ou
    with torch.no_grad():
        for images, images_rotated, labels, cls_labels in testloader:
            if task == 'rotation':
              images, labels = images_rotated.to(device), labels.to(device)
            elif task == 'classification':
              images, labels = images.to(device), cls_labels.to(device)
            # TODO: Calculate outputs by running images through the network
            # The class with the highest energy is what we choose as prediction
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            # loss
            avg_test_loss += criterion(outputs, labels)  / len(testloader)
    print('TESTING:')
    print(f'Accuracy of the network on the 10000 test images: {100 * correct / tota
    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

In this section, we will train a ResNet18 model 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'
```

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

from torchvision.models import resnet18

net = resnet18(num_classes=4)
net = net.to(device)
```

```
import torch.optim as optim

# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr = 1e-2, momentum=0.9)
```

```
# 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(trainlo
            adjust_learning_rate(optimizer, epoch, init_lr, decay_epochs)

            # TODO: Set the data to the correct device; Different task will use dif
            #
            if task == 'rotation':
              inputs = imgs_rotated.to(device)
              labels = rotation_label.to(device)
            elif task == 'classification':
              inputs = imgs.to(device)
              labels = cls_label.to(device)

            # Zero the parameter gradients
            #
            optimizer.zero_grad()

            # TODO: forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()


            # TODO: Get predicted results
            _, predicted = torch.max(outputs.data, 1)

            # 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-batch
                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
        #
        #
        net.eval()
        run_test(net, testloader, criterion, task)

    print('Finished Training')


train(net, criterion, optimizer, num_epochs=50, decay_epochs=12, init_lr=0.01, task

# TODO: Save the model
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

```
     [illegible]
     [41,   300] loss: 0.550 acc: 78.76 time: 4.51
```

```
TESTING:
Accuracy of the network on the 10000 test images: 78.34 %
Average loss on the 10000 test images: 0.549
[42,   100] loss: 0.553 acc: 78.13 time: 4.61
[42,   200] loss: 0.565 acc: 77.80 time: 4.97
[42,   300] loss: 0.539 acc: 78.77 time: 4.72
TESTING:
Accuracy of the network on the 10000 test images: 78.16 %
Average loss on the 10000 test images: 0.551
[43,   100] loss: 0.550 acc: 78.84 time: 5.11
[43,   200] loss: 0.562 acc: 77.80 time: 4.50
[43,   300] loss: 0.548 acc: 78.42 time: 4.72
TESTING:
Accuracy of the network on the 10000 test images: 77.88 %
Average loss on the 10000 test images: 0.555
[44,   100] loss: 0.553 acc: 78.34 time: 4.74
[44,   200] loss: 0.555 acc: 78.29 time: 4.96
[44,   300] loss: 0.562 acc: 77.97 time: 4.42
TESTING:
Accuracy of the network on the 10000 test images: 77.81 %
Average loss on the 10000 test images: 0.555
[45,   100] loss: 0.559 acc: 78.38 time: 4.66
[45,   200] loss: 0.548 acc: 78.45 time: 4.50
[45,   300] loss: 0.557 acc: 78.07 time: 5.10
TESTING:
Accuracy of the network on the 10000 test images: 78.06 %
Average loss on the 10000 test images: 0.552
[46,   100] loss: 0.551 acc: 78.49 time: 5.17
[46,   200] loss: 0.538 acc: 78.92 time: 4.42
[46,   300] loss: 0.545 acc: 78.43 time: 4.46
TESTING:
Accuracy of the network on the 10000 test images: 77.93 %
Average loss on the 10000 test images: 0.556
[47,   100] loss: 0.555 acc: 78.62 time: 4.70
[47,   200] loss: 0.561 acc: 77.80 time: 4.91
[47,   300] loss: 0.555 acc: 78.31 time: 4.56
TESTING:
Accuracy of the network on the 10000 test images: 78.33 %
Average loss on the 10000 test images: 0.551
[48,   100] loss: 0.555 acc: 78.25 time: 4.98
[48,   200] loss: 0.543 acc: 78.68 time: 4.48
[48,   300] loss: 0.553 acc: 78.84 time: 4.94
TESTING:
Accuracy of the network on the 10000 test images: 77.91 %
Average loss on the 10000 test images: 0.558
[49,   100] loss: 0.558 acc: 78.19 time: 4.72
[49,   200] loss: 0.550 acc: 78.51 time: 4.80
[49,   300] loss: 0.551 acc: 78.42 time: 4.52
TESTING:
Accuracy of the network on the 10000 test images: 78.19 %
```

```
Average loss on the 10000 test images: 0.555
[50,   100] loss: 0.550 acc: 78.40 time: 4.63
[50,   200] loss: 0.546 acc: 78.59 time: 4.44
[50,   300] loss: 0.558 acc: 78.14 time: 5.12
TESTING:
Accuracy of the network on the 10000 test images: 78.26 %
Average loss on the 10000 test images: 0.549
Finished Training
```

Fine-tuning on the pre-trained model

In this section, we will load the pre-trained 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: Load the pre-trained ResNet18 model
net = resnet18(pretrained=True)
net = net.to(device)
```

```
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:208: UserW
  warnings.warn(
/usr/local/lib/python3.9/dist-packages/torchvision/models/_utils.py:223: UserW
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /r
100%                                          44.7M/44.7M [00:00<00:00, 177MB/s]
```

```
# TODO: Freeze all previous layers; only keep the 'layer4' block and 'fc' layer tra
for name, param in net.named_parameters():
    if 'layer4' in name or 'fc' in name:
        param.requires_grad = True
    else:
        param.requires_grad = False
```

```
# 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.SGD(filter(lambda p: p.requires_grad, net.parameters()), lr=0.001
```

```
train(net, criterion, optimizer, num_epochs=40, decay_epochs=10, init_lr=0.01, task
```

```
[31,   300] loss: 0.721 acc: 74.49 time: 4.82
TESTING:
Accuracy of the network on the 10000 test images: 70.88 %
Average loss on the 10000 test images: 0.877
[32,   100] loss: 0.716 acc: 74.55 time: 5.20
[32,   200] loss: 0.714 acc: 74.56 time: 4.39
[32,   300] loss: 0.718 acc: 74.77 time: 4.35
TESTING:
Accuracy of the network on the 10000 test images: 69.99 %
Average loss on the 10000 test images: 0.959
[33,   100] loss: 0.731 acc: 74.23 time: 4.43
```

```
[33,   100] loss: 0.731 acc: 74.23 time: 4.43
[33,   200] loss: 0.703 acc: 75.30 time: 4.90
[33,   300] loss: 0.730 acc: 74.20 time: 4.36
TESTING:
Accuracy of the network on the 10000 test images: 69.98 %
Average loss on the 10000 test images: 0.969
[34,   100] loss: 0.716 acc: 74.62 time: 4.64
[34,   200] loss: 0.709 acc: 74.72 time: 4.31
[34,   300] loss: 0.711 acc: 74.75 time: 4.60
TESTING:
Accuracy of the network on the 10000 test images: 70.64 %
Average loss on the 10000 test images: 0.891
[35,   100] loss: 0.714 acc: 74.27 time: 4.49
[35,   200] loss: 0.712 acc: 74.68 time: 4.87
[35,   300] loss: 0.706 acc: 74.95 time: 4.26
TESTING:
Accuracy of the network on the 10000 test images: 70.25 %
Average loss on the 10000 test images: 0.928
[36,   100] loss: 0.709 acc: 74.45 time: 4.66
[36,   200] loss: 0.722 acc: 74.28 time: 4.25
[36,   300] loss: 0.716 acc: 74.62 time: 4.75
TESTING:
Accuracy of the network on the 10000 test images: 70.25 %
Average loss on the 10000 test images: 0.901
[37,   100] loss: 0.711 acc: 75.28 time: 4.58
[37,   200] loss: 0.718 acc: 74.66 time: 4.82
[37,   300] loss: 0.719 acc: 74.73 time: 4.35
TESTING:
Accuracy of the network on the 10000 test images: 70.30 %
Average loss on the 10000 test images: 0.913
[38,   100] loss: 0.725 acc: 74.33 time: 4.53
[38,   200] loss: 0.709 acc: 74.63 time: 4.44
[38,   300] loss: 0.716 acc: 74.78 time: 5.00
TESTING:
Accuracy of the network on the 10000 test images: 69.92 %
Average loss on the 10000 test images: 0.974
[39,   100] loss: 0.733 acc: 74.20 time: 5.08
[39,   200] loss: 0.712 acc: 74.90 time: 4.45
[39,   300] loss: 0.721 acc: 74.86 time: 4.84
TESTING:
Accuracy of the network on the 10000 test images: 70.17 %
Average loss on the 10000 test images: 0.903
[40,   100] loss: 0.704 acc: 75.03 time: 4.56
[40,   200] loss: 0.702 acc: 75.05 time: 5.00
[40,   300] loss: 0.708 acc: 74.70 time: 4.45
TESTING:
Accuracy of the network on the 10000 test images: 70.54 %
Average loss on the 10000 test images: 0.906
Finished Training
```

Fine-tuning on the randomly initialized model

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
# Define ResNet18 model
model = resnet18()
for param in model.parameters():
    nn.init.normal_(param, mean=0, std=0.01)


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

```
# 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.SGD(filter(lambda p: p.requires_grad, net.parameters()), lr=0.001
```

```
train(net, criterion, optimizer, num_epochs=30, decay_epochs=10, init_lr=0.01, task
```

```
[21,   300] loss: 0.737 acc: 74.20 time: 4.21
TESTING:
Accuracy of the network on the 10000 test images: 68.82 %
Average loss on the 10000 test images: 0.992
[22,   100] loss: 0.738 acc: 73.54 time: 4.46
[22,   200] loss: 0.712 acc: 74.85 time: 4.47
[22,   300] loss: 0.728 acc: 74.09 time: 4.66
TESTING:
Accuracy of the network on the 10000 test images: 70.26 %
Average loss on the 10000 test images: 0.867
[23,   100] loss: 0.723 acc: 74.68 time: 5.04
[23,   200] loss: 0.721 acc: 74.15 time: 4.19
```

```
[23,   200] loss: 0.721 acc: 74.15 time: 4.19
[23,   300] loss: 0.731 acc: 74.15 time: 4.27
TESTING:
Accuracy of the network on the 10000 test images: 70.13 %
Average loss on the 10000 test images: 0.896
[24,   100] loss: 0.723 acc: 74.27 time: 4.50
[24,   200] loss: 0.729 acc: 73.80 time: 4.69
[24,   300] loss: 0.719 acc: 74.06 time: 4.41
TESTING:
Accuracy of the network on the 10000 test images: 69.35 %
Average loss on the 10000 test images: 0.917
[25,   100] loss: 0.729 acc: 74.59 time: 4.96
[25,   200] loss: 0.736 acc: 73.77 time: 4.34
[25,   300] loss: 0.725 acc: 74.41 time: 4.62
TESTING:
Accuracy of the network on the 10000 test images: 69.55 %
Average loss on the 10000 test images: 0.927
[26,   100] loss: 0.734 acc: 74.00 time: 4.46
[26,   200] loss: 0.726 acc: 74.21 time: 4.97
[26,   300] loss: 0.718 acc: 74.66 time: 4.51
TESTING:
Accuracy of the network on the 10000 test images: 69.44 %
Average loss on the 10000 test images: 0.923
[27,   100] loss: 0.725 acc: 74.37 time: 4.54
[27,   200] loss: 0.715 acc: 74.66 time: 4.29
[27,   300] loss: 0.724 acc: 74.23 time: 4.79
TESTING:
Accuracy of the network on the 10000 test images: 69.98 %
Average loss on the 10000 test images: 0.908
[28,   100] loss: 0.721 acc: 74.12 time: 4.72
[28,   200] loss: 0.724 acc: 74.27 time: 4.68
[28,   300] loss: 0.729 acc: 74.33 time: 4.38
TESTING:
Accuracy of the network on the 10000 test images: 70.17 %
Average loss on the 10000 test images: 0.870
[29,   100] loss: 0.728 acc: 74.45 time: 4.44
[29,   200] loss: 0.730 acc: 73.88 time: 4.30
[29,   300] loss: 0.736 acc: 73.84 time: 5.04
TESTING:
Accuracy of the network on the 10000 test images: 70.40 %
Average loss on the 10000 test images: 0.878
[30,   100] loss: 0.723 acc: 74.27 time: 5.14
[30,   200] loss: 0.736 acc: 73.76 time: 4.35
[30,   300] loss: 0.723 acc: 74.46 time: 4.25
TESTING:
Accuracy of the network on the 10000 test images: 69.72 %
Average loss on the 10000 test images: 0.927
Finished Training
```

Supervised training on the pre-trained model

In this section, we will load the pre-trained 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: Load the pre-trained ResNet18 model
net = resnet18(pretrained=True)
net = net.to(device)


# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(filter(lambda p: p.requires_grad, net.parameters()), lr=0.01,


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

```
    [11,   300] loss: 0.340 acc: 88.29 time: 5.03
    TESTING:
    Accuracy of the network on the 10000 test images: 85.81 %
    Average loss on the 10000 test images: 0.432
    [12,   100] loss: 0.321 acc: 88.79 time: 4.77
    [12,   200] loss: 0.329 acc: 88.69 time: 4.72
    [12,   300] loss: 0.329 acc: 88.87 time: 4.40
    TESTING:
    Accuracy of the network on the 10000 test images: 85.91 %
    Average loss on the 10000 test images: 0.431
    [13,   100] loss: 0.320 acc: 88.80 time: 4.59
    [13,   200] loss: 0.318 acc: 88.95 time: 4.56
    [13,   300] loss: 0.306 acc: 89.32 time: 4.76
    TESTING:
    Accuracy of the network on the 10000 test images: 86.04 %
    Average loss on the 10000 test images: 0.429
    [14,   100] loss: 0.294 acc: 89.63 time: 5.18
    [14,   200] loss: 0.312 acc: 89.26 time: 4.52
    [14,   300] loss: 0.302 acc: 89.44 time: 4.53
    TESTING:
    Accuracy of the network on the 10000 test images: 86.22 %
    Average loss on the 10000 test images: 0.427
    [15,   100] loss: 0.288 acc: 90.12 time: 4.66
    [15,   200] loss: 0.282 acc: 90.27 time: 4.95
```

```
[15,   300] loss: 0.299 acc: 89.91 time: 4.42
TESTING:
Accuracy of the network on the 10000 test images: 86.27 %
Average loss on the 10000 test images: 0.427
[16,   100] loss: 0.290 acc: 89.75 time: 4.92
[16,   200] loss: 0.290 acc: 90.05 time: 4.39
[16,   300] loss: 0.294 acc: 90.10 time: 4.79
TESTING:
Accuracy of the network on the 10000 test images: 86.18 %
Average loss on the 10000 test images: 0.426
[17,   100] loss: 0.281 acc: 90.09 time: 4.65
[17,   200] loss: 0.284 acc: 90.21 time: 5.04
[17,   300] loss: 0.278 acc: 90.40 time: 4.47
TESTING:
Accuracy of the network on the 10000 test images: 86.30 %
Average loss on the 10000 test images: 0.424
[18,   100] loss: 0.270 acc: 90.61 time: 4.56
[18,   200] loss: 0.286 acc: 89.89 time: 4.39
[18,   300] loss: 0.278 acc: 90.39 time: 5.00
TESTING:
Accuracy of the network on the 10000 test images: 86.44 %
Average loss on the 10000 test images: 0.423
[19,   100] loss: 0.281 acc: 90.30 time: 5.11
[19,   200] loss: 0.288 acc: 90.20 time: 4.44
[19,   300] loss: 0.281 acc: 90.16 time: 4.46
TESTING:
Accuracy of the network on the 10000 test images: 86.44 %
Average loss on the 10000 test images: 0.426
[20,   100] loss: 0.258 acc: 90.65 time: 4.67
[20,   200] loss: 0.282 acc: 90.24 time: 4.83
[20,   300] loss: 0.264 acc: 90.93 time: 4.52
TESTING:
Accuracy of the network on the 10000 test images: 86.16 %
Average loss on the 10000 test images: 0.431
Finished Training
```

Supervised training on the randomly initialized model

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
model = resnet18()
for param in model.parameters():
    nn.init.normal_(param, mean=0, std=0.01)


# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(filter(lambda p: p.requires_grad, net.parameters()), lr=0.01,


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

```
    [11,   300] loss: 0.210 acc: 92.70 time: 4.36
    TESTING:
    Accuracy of the network on the 10000 test images: 86.55 %
    Average loss on the 10000 test images: 0.420
    [12,   100] loss: 0.207 acc: 92.83 time: 4.76
    [12,   200] loss: 0.207 acc: 92.55 time: 4.63
    [12,   300] loss: 0.195 acc: 93.30 time: 4.80
    TESTING:
    Accuracy of the network on the 10000 test images: 86.65 %
    Average loss on the 10000 test images: 0.426
    [13,   100] loss: 0.190 acc: 93.66 time: 5.14
    [13,   200] loss: 0.186 acc: 93.40 time: 4.54
    [13,   300] loss: 0.190 acc: 93.26 time: 4.57
    TESTING:
    Accuracy of the network on the 10000 test images: 86.78 %
    Average loss on the 10000 test images: 0.431
    [14,   100] loss: 0.185 acc: 93.45 time: 4.58
    [14,   200] loss: 0.175 acc: 93.96 time: 4.99
    [14,   300] loss: 0.183 acc: 93.53 time: 4.35
    TESTING:
    Accuracy of the network on the 10000 test images: 86.61 %
    Average loss on the 10000 test images: 0.429
    [15,   100] loss: 0.176 acc: 93.80 time: 4.74
    [15,   200] loss: 0.176 acc: 93.84 time: 4.58
    [15,   300] loss: 0.178 acc: 93.87 time: 5.08
    TESTING:
    Accuracy of the network on the 10000 test images: 86.79 %
    Average loss on the 10000 test images: 0.429
    [16,   100] loss: 0.165 acc: 94.57 time: 4.91
    [16,   200] loss: 0.163 acc: 94.20 time: 4.71
    [16,   300] loss: 0.180 acc: 93.50 time: 4.57
```

```
TESTING:
Accuracy of the network on the 10000 test images: 86.70 %
Average loss on the 10000 test images: 0.433
[17,   100] loss: 0.165 acc: 94.28 time: 4.55
[17,   200] loss: 0.173 acc: 93.95 time: 4.60
[17,   300] loss: 0.171 acc: 93.94 time: 4.75
TESTING:
Accuracy of the network on the 10000 test images: 86.86 %
Average loss on the 10000 test images: 0.441
[18,   100] loss: 0.161 acc: 94.38 time: 5.14
[18,   200] loss: 0.158 acc: 94.60 time: 4.44
[18,   300] loss: 0.156 acc: 94.31 time: 4.47
TESTING:
Accuracy of the network on the 10000 test images: 86.85 %
Average loss on the 10000 test images: 0.446
[19,   100] loss: 0.161 acc: 94.28 time: 4.55
[19,   200] loss: 0.153 acc: 94.64 time: 4.93
[19,   300] loss: 0.159 acc: 94.55 time: 4.66
TESTING:
Accuracy of the network on the 10000 test images: 86.90 %
Average loss on the 10000 test images: 0.444
[20,   100] loss: 0.147 acc: 94.65 time: 4.93
[20,   200] loss: 0.153 acc: 94.47 time: 4.36
[20,   300] loss: 0.154 acc: 94.45 time: 4.65
TESTING:
Accuracy of the network on the 10000 test images: 86.95 %
Average loss on the 10000 test images: 0.444
Finished Training
```