# (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 slashes or spaces
import os
datadir = "/content/drive/My Drive/CS444/assignment3_starter/assignment3_part1/"
if not os.path.exists(datadir):
  !ln -s "/content/drive/My Drive/CS444/assignment3_starter/assignment3_part1/" $datadir # TODO: Fill your A3 path
os.chdir(datadir)
!pwd
```

```
/content/drive/My Drive/CS444/assignment3_starter/assignment3_part1
```

# 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
    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=transform)

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

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

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

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


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

transform_test = transforms.Compose([
```

```
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

batch_size = 128

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

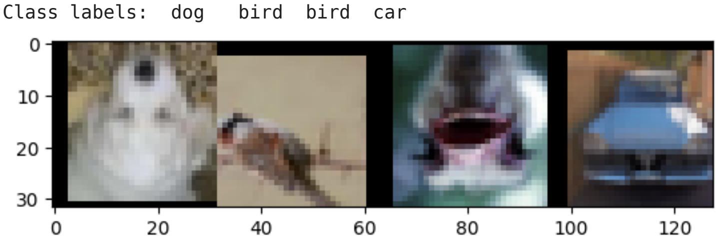
```
Files already downloaded and verified
Files already downloaded and verified
```

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 range(4)))
```

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


```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data
Class labels:  dog   bird  bird  car
```




```
Rotation labels:  180   90    180   0
```

# Evaluation code

```
import time

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

```
    with torch.no_grad():
        for images, images_rotated, labels, cls_labels in testloader:
            if task == 'rotation':
              images, labels = images_rotated.to(device), labels.to(device)
            elif task == 'classification':
              images, labels = images.to(device), cls_labels.to(device)
            # TODO: Calculate outputs by running images through the network
            # The class with the highest energy is what we choose as prediction
            outputs = net(images)
            predicted = torch.argmax(outputs, axis =1 )
            total += labels.size(0)

            avg_test_loss += criterion(outputs, labels)  / len(testloader)
            correct += (predicted == labels).sum().item()
    print('TESTING:')

    print(f'Accuracy of the network on the 10000 test images: {100 * correct / total:.2f} %')
    print(f'Average loss on the 10000 test images: {avg_test_loss:.3f}')


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

## Train a ResNet18 on the rotation task

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
criterion = None
optimizer = None

# TODO: Define criterion and optimizer
#
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lr = 1e-3, params= net.parameters())
```

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

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

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

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

        net.train()

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

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

            if task == 'rotation':
              images, labels = imgs_rotated.to(device), rotation_label.to(device)
            elif task == 'classification':
              images, labels = imgs.to(device), cls_label.to(device)
```

```
            # TODO: Zero the parameter gradients
            #
            optimizer.zero_grad()

            # TODO: forward + backward + optimize

            outputs = net(images)
            loss = criterion(outputs, labels)
            loss.backward()

            optimizer.step()
            #

            # TODO: Get predicted results
            predicted = torch.argmax(outputs, axis =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-batches
                print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / print_freq:.3f} acc: {100*running_correct / running_total:.2f} time: {
                running_loss, running_correct, running_total = 0.0, 0.0, 0.0
                start_time = time.time()

        # TODO: Run the run_test() function after each epoch; Set the model to the evaluation mode.
        net.eval()
        run_test(net, testloader, criterion, task)

    print('Finished Training')


train(net, criterion, optimizer, num_epochs=30, decay_epochs=15, init_lr=0.001, task='rotation')

# TODO: Save the model
torch.save(net.state_dict(), 'Resnet_03_26_15_41.pt')
```

```
[29,   200] loss: 0.548 acc: 78.52 time: 11.27
[29,   300] loss: 0.562 acc: 78.18 time: 7.60
TESTING:
Accuracy of the network on the 10000 test images: 78.72 %
Average loss on the 10000 test images: 0.548
[30,   100] loss: 0.541 acc: 78.73 time: 9.48
[30,   200] loss: 0.558 acc: 78.34 time: 7.68
[30,   300] loss: 0.538 acc: 78.95 time: 8.95
TESTING:
Accuracy of the network on the 10000 test images: 79.22 %
Average loss on the 10000 test images: 0.536
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(num_classes=4)
net.load_state_dict(torch.load('Resnet_03_26_15_41.pt'))
```

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

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

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



num_ftrs  = net.fc.in_features
net.fc    = nn.Linear(num_ftrs, 10)

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

net = net.to(device)
```

```
# 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 = torch.optim.Adam(net.parameters(), lr = 1e-3)


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

```
TESTING:
Accuracy of the network on the 10000 test images: 65.41 %
Average loss on the 10000 test images: 0.967
[12,   100] loss: 0.944 acc: 66.24 time: 8.03
[12,   200] loss: 0.966 acc: 64.73 time: 8.60
[12,   300] loss: 0.963 acc: 65.67 time: 8.02
TESTING:
Accuracy of the network on the 10000 test images: 65.97 %
Average loss on the 10000 test images: 0.962
[13,   100] loss: 0.944 acc: 66.59 time: 8.35
[13,   200] loss: 0.949 acc: 66.43 time: 6.74
[13,   300] loss: 0.953 acc: 65.43 time: 8.20
TESTING:
Accuracy of the network on the 10000 test images: 65.67 %
Average loss on the 10000 test images: 0.963
[14,   100] loss: 0.941 acc: 66.38 time: 6.95
[14,   200] loss: 0.953 acc: 65.66 time: 8.32
[14,   300] loss: 0.945 acc: 66.00 time: 7.61
TESTING:
Accuracy of the network on the 10000 test images: 65.30 %
Average loss on the 10000 test images: 0.963
[15,   100] loss: 0.938 acc: 66.71 time: 8.51
[15,   200] loss: 0.927 acc: 66.70 time: 7.57
[15,   300] loss: 0.943 acc: 66.05 time: 7.63
TESTING:
Accuracy of the network on the 10000 test images: 65.69 %
Average loss on the 10000 test images: 0.958
[16,   100] loss: 0.929 acc: 66.73 time: 7.78
[16,   200] loss: 0.941 acc: 66.09 time: 7.39
[16,   300] loss: 0.930 acc: 66.55 time: 8.14
TESTING:
Accuracy of the network on the 10000 test images: 66.35 %
Average loss on the 10000 test images: 0.957
[17,   100] loss: 0.919 acc: 67.04 time: 7.59
[17,   200] loss: 0.946 acc: 66.00 time: 8.02
[17,   300] loss: 0.930 acc: 66.25 time: 6.83
TESTING:
Accuracy of the network on the 10000 test images: 65.94 %
Average loss on the 10000 test images: 0.956
[18,   100] loss: 0.927 acc: 66.74 time: 8.46
[18,   200] loss: 0.930 acc: 66.89 time: 6.86
[18,   300] loss: 0.934 acc: 66.23 time: 8.13
TESTING:
Accuracy of the network on the 10000 test images: 66.00 %
Average loss on the 10000 test images: 0.957
[19,   100] loss: 0.936 acc: 66.16 time: 7.00
[19,   200] loss: 0.931 acc: 66.19 time: 8.40
[19,   300] loss: 0.930 acc: 66.63 time: 6.75
TESTING:
Accuracy of the network on the 10000 test images: 66.41 %
Average loss on the 10000 test images: 0.950
[20,   100] loss: 0.915 acc: 67.30 time: 8.38
[20,   200] loss: 0.928 acc: 66.71 time: 6.89
[20,   300] loss: 0.928 acc: 67.21 time: 8.16
TESTING:
Accuracy of the network on the 10000 test images: 66.29 %
Average loss on the 10000 test images: 0.950
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
#
net = resnet18(num_classes  = 10)
for param in net.parameters():
  param= torch.rand(size = param.size())


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

```
for 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

net = net.to(device)


# 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 = torch.optim.Adam(lr = 1e-3, params= net.parameters())


train(net, criterion, optimizer, num_epochs=30, decay_epochs=5, init_lr=0.001, task='classification')
```

```
[28,   200] loss: 1.569 acc: 43.52 time: 6.99
[28,   300] loss: 1.562 acc: 44.90 time: 8.59
TESTING:
Accuracy of the network on the 10000 test images: 45.53 %
Average loss on the 10000 test images: 1.527
[29,   100] loss: 1.575 acc: 43.59 time: 7.15
[29,   200] loss: 1.567 acc: 44.27 time: 8.67
[29,   300] loss: 1.561 acc: 44.26 time: 6.95
TESTING:
Accuracy of the network on the 10000 test images: 45.61 %
Average loss on the 10000 test images: 1.525
[30,   100] loss: 1.574 acc: 43.45 time: 8.80
[30,   200] loss: 1.565 acc: 43.60 time: 7.04
[30,   300] loss: 1.568 acc: 43.89 time: 8.71
TESTING:
Accuracy of the network on the 10000 test images: 45.32 %
Average loss on the 10000 test images: 1.527
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.

```
from prompt_toolkit.filters import in_editing_mode
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(num_classes = 4)
net.load_state_dict(torch.load('Resnet_03_26_15_41.pt'))

in_features = net.fc.in_features
net.fc = nn.Linear(in_features, 10)
net = net.to(device)


# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lr = 1e-3, params= net.parameters())


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

```
[18,   300] loss: 0.546 acc: 80.61 time: 9.69
TESTING:
Accuracy of the network on the 10000 test images: 78.51 %
Average loss on the 10000 test images: 0.614
[19,   100] loss: 0.544 acc: 80.63 time: 9.70
[19,   200] loss: 0.534 acc: 81.20 time: 9.79
[19,   300] loss: 0.556 acc: 80.47 time: 7.78
TESTING:
Accuracy of the network on the 10000 test images: 78.68 %
Average loss on the 10000 test images: 0.612
[20,   100] loss: 0.544 acc: 80.74 time: 9.47
[20,   200] loss: 0.550 acc: 80.73 time: 9.46
[20,   300] loss: 0.532 acc: 81.28 time: 8.28
TESTING:
Accuracy of the network on the 10000 test images: 78.75 %
Average loss on the 10000 test images: 0.611
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
#
net = resnet18(num_classes = 10)
for param in net.parameters():
  param= torch.rand(size = param.size())

net = net.to(device)


# TODO: Define criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lr = 1e-3, params= net.parameters())


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

```
Accuracy of the network on the 10000 test images: 82.56 %
Average loss on the 10000 test images: 0.516
[19,   100] loss: 0.409 acc: 85.80 time: 10.03
[19,   200] loss: 0.394 acc: 86.06 time: 7.73
[19,   300] loss: 0.406 acc: 85.89 time: 9.20
TESTING:
Accuracy of the network on the 10000 test images: 82.69 %
Average loss on the 10000 test images: 0.524
[20,   100] loss: 0.394 acc: 86.34 time: 8.50
[20,   200] loss: 0.386 acc: 86.69 time: 10.90
[20,   300] loss: 0.415 acc: 85.73 time: 9.46
TESTING:
Accuracy of the network on the 10000 test images: 82.70 %
Average loss on the 10000 test images: 0.515
Finished Training
```