Sai Jayanth Kalisi

Problem 1: Long-Tailed Recognition on Imbalanced Dataset

In the existing visual recognition setting, the training data and testing data are both balanced under a closed-world setting, e.g., the ImageNet dataset. However, this setting is not a good proxy for the real-world scenario. This imbalanced data distribution in the training set may largely degrade the performance of the machine learning or deep learning-based method.

Our goal is to build a CNN model that can accurately classify the images into their respective categories under imbalanced settings.

Readings before you start

1. Bag of tricks for long-tailed visual recognition with deep convolutional neural networks [Paper] [Github]

```
%matplotlib inline

import csv
import math
import os

import numpy as np
import pandas as pd

from tqdm import tqdm

# Pytorch
import torch
import torch.nn as nn

from torch.utils.tensorboard import SummaryWriter
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, random_split

import torchvision
import torchvision.datasets as datasets
import torchvision.transforms as transforms


import matplotlib.pyplot as plt
import seaborn as sns
sns.set(palette='pastel')
```

```

np.random.seed(568)
```

Prepare Imbalanced CIFAR-30 Dataset from CIFAR-*100*

You will be building the imbalanced version of CIFAR-30 from the CIFAR-100:

$$\beta = \frac{max(\{n_1, n_2, \cdots, n_k\})}{min(\{n_1, n_2, \cdots, n_k\})}$$

\noindent where n_i represents the number of images for class i. Therefore, the larger the imbalance factor β is, the harder it gets for doing long-tailed recognition on such data. With a $\beta = 100$ version of CIFAR-100, the head classes will have 500 training samples while the tail classes only have 5 training samples.

```
# create a custom dataset CIFAR30 from CIFAR100
class CIFAR30(torchvision.datasets.CIFAR100):
    # cifar100 has 100 classes, we only want 30
    cls_num = 30

    def __init__(self, root, imb_type='exp', imb_factor=0.01, rand_number=0, train=True,
                 transform=None, target_transform=None,
                 download=False, imbalanced=False):
        super(CIFAR30, self).__init__(root, train, transform, target_transform, download)
        np.random.seed(rand_number)

        self.remove_extra_class(self.cls_num)

        if self.train and imbalanced:
            img_num_list = self.get_img_num_per_cls(self.cls_num, imb_type, imb_factor)
            self.gen_imbalanced_data(img_num_list)

        self.update_num_per_cls()

    # remove extra classes to make it 30 classes
    def remove_extra_class(self, cls_num):
        new_data = []
        new_targets = []
        targets_np = np.array(self.targets, dtype=np.int64)
        classes = np.unique(targets_np)
        for i in range(cls_num):
            idx = np.where(targets_np == i)[0]
            new_data.append(self.data[idx, ...])
            new_targets.extend([i, ] * len(idx))
        new_data = np.vstack(new_data)
        self.data = new_data
        self.targets = new_targets
```

```python


    # get the number of images per class we desire
    def get_img_num_per_cls(self, cls_num, imb_type, imb_factor):
        img_max = len(self.data) / cls_num
        img_num_per_cls = []
        if imb_type == 'exp':
            for cls_idx in range(cls_num):
                num = img_max * (imb_factor ** (cls_idx / (cls_num - 1.0)))
                img_num_per_cls.append(int(num))
        elif imb_type == 'step':
            for cls_idx in range(cls_num // 2):
                img_num_per_cls.append(int(img_max))
            for cls_idx in range(cls_num // 2):
                img_num_per_cls.append(int(img_max * imb_factor))
        else:
            img_num_per_cls.extend([int(img_max)] * cls_num)
        return img_num_per_cls

    # generate imbalanced data from original dataset with given img_num_per_cls
    def gen_imbalanced_data(self, img_num_per_cls):
        new_data = []
        new_targets = []
        targets_np = np.array(self.targets)
        classes = np.unique(targets_np)
        self.num_per_cls_dict = dict()
        for the_class, the_img_num in zip(classes, img_num_per_cls):
            self.num_per_cls_dict[the_class] = the_img_num
            idx = np.where(targets_np == the_class)[0]
            np.random.shuffle(idx)
            selec_idx = idx[:the_img_num]
            new_data.append(self.data[selec_idx, ...])
            new_targets.extend([the_class, ] * the_img_num)
        new_data = np.vstack(new_data)
        self.data = new_data
        self.targets = new_targets

    def get_cls_num_list(self):
        cls_num_list = []
        for i in range(self.cls_num):
            cls_num_list.append(self.num_per_cls_dict[i])
        return cls_num_list

    def update_num_per_cls(self):
        targets_np = np.array(self.targets, dtype=np.int64)
        classes = np.unique(targets_np)
        self.num_per_cls_dict = dict()
        for cls in classes:
            self.num_per_cls_dict[cls] = len(np.where(targets_np == cls)[0])
```

We will also be adapting some extra transorms (augmentations) on our CIFAR-30:

```
# transforms for training and testing
training_transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.RandomHorizontalFlip(p=1),
     transforms.RandomAffine(degrees=60),
     transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
)
testing_transform = transforms.Compose(
    [transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
)

# create the datasets
cifar30_trainset = CIFAR30(root='./data', train=True, download=True,
                                transform=training_transform, imbalanced=False)
im_cifar30_trainset = CIFAR30(root='./data', train=True, download=True,
                                transform=training_transform, imbalanced=True)
cifar30_testset = CIFAR30(root='./data', train=False, download=True,
                                transform=testing_transform, imbalanced=False)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz to ./data/cifar-10
100%|██████████| 169001437/169001437 [00:12<00:00, 13424184.26it/s]
Extracting ./data/cifar-100-python.tar.gz to ./data
Files already downloaded and verified
Files already downloaded and verified
```

Compare the data (label) distribution of the three dataset `cifar30_trainset`, `im_cifar30_trainset`, and `cifar30_testset` we constructed:

1. Balanced Training Data
2. Imbalanced Training Data
3. Balanced Testing Data

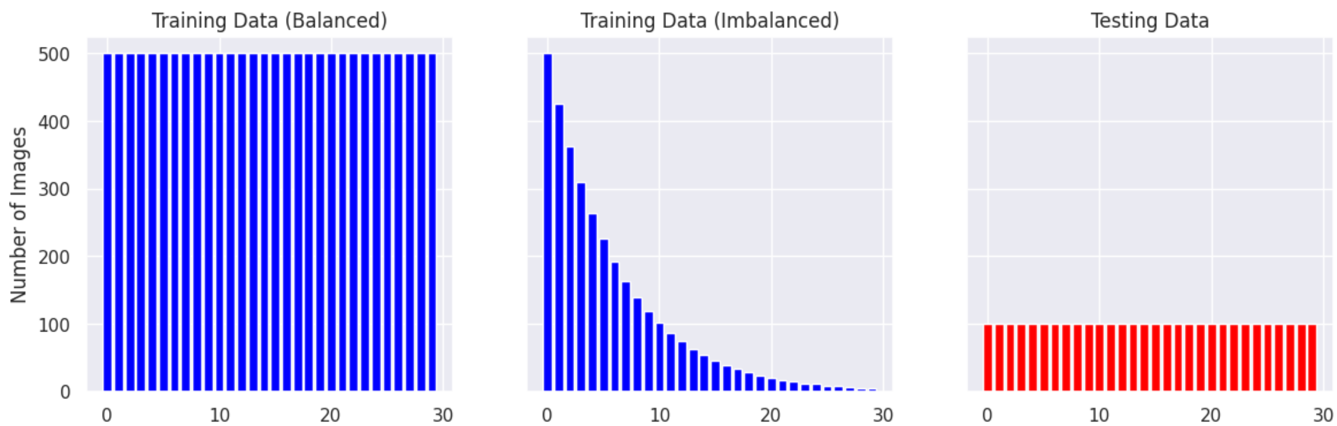
```
training_distribution = list(cifar30_trainset.num_per_cls_dict.values())
im_training_distribution = list(im_cifar30_trainset.num_per_cls_dict.values())
testing_distribution = list(cifar30_testset.num_per_cls_dict.values())
training_cls = list(im_cifar30_trainset.num_per_cls_dict.keys())

plt.subplots(1, 3, sharey=True, figsize=(14,4))

plt.subplot(1, 3, 1)
plt.bar(training_cls, training_distribution, color='blue')
plt.title('Training Data (Balanced)')
```

```
plt.ylabel('Number of Images')
plt.subplot(1, 3, 2)
plt.bar(training_cls, im_training_distribution, color='blue')
plt.title('Training Data (Imbalanced)')
plt.subplot(1, 3, 3)
plt.bar(training_cls, testing_distribution, color='red')
plt.title('Testing Data')
```

```
Text(0.5, 1.0, 'Testing Data')
```

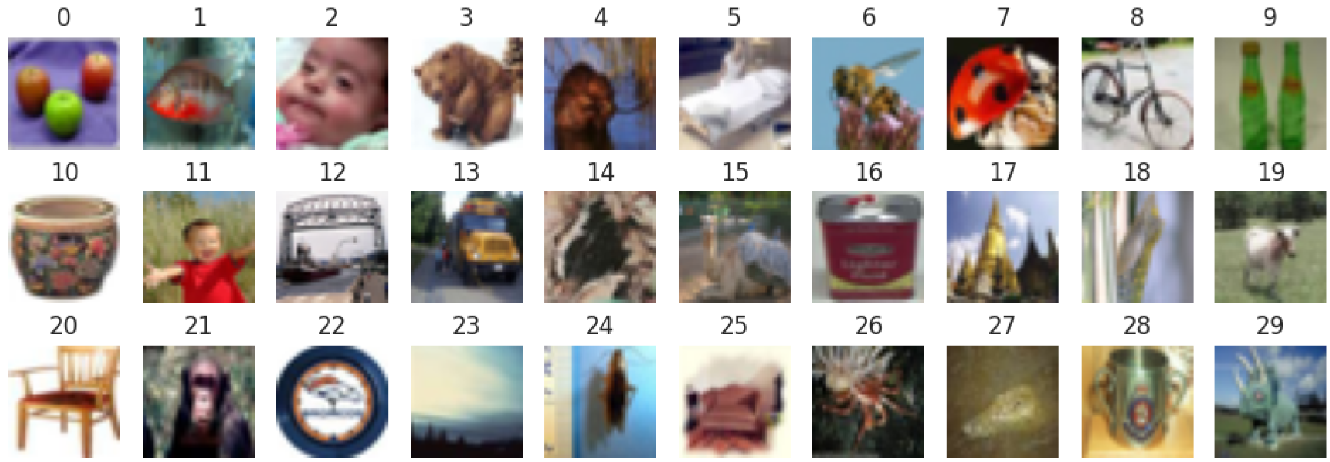


Show some images with labels (class names) from dataset.

```
def cifar_imshow(img):
  img = img / 2 + 0.5 # unnormalize the image
  npimg = img.numpy()
  return np.transpose(npimg, (1, 2, 0)) # reorganize the channel

# visualize some samples in the CIFAR-30 dataset
fig, axs = plt.subplots(3, 10, figsize = (12, 4))

# loop through subplots and images
for i, ax in enumerate(axs.flat):
  ax.imshow(cifar_imshow(cifar30_testset[i*100][0]))
  ax.axis('off')
  ax.set_title('{}'.format(cifar30_testset[i*100][1]))
```



1-a. Train CNN Baseline

Check whether your runtime is on GPU or not.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print('Using device:', device)
!nvidia-smi
```

```
Using device: cuda:0
Sat Nov  4 17:30:09 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   53C    P8    13W /  70W |      3MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

The CNN we will be using in this problem is called ResNet:

```
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(
            in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion*planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, in_planes, planes, stride=1):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3,
                               stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, self.expansion *
                               planes, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(self.expansion*planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion*planes)
            )

    def forward(self, x):
```

```
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.relu(self.bn2(self.conv2(out)))
        out = self.bn3(self.conv3(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=30):
        super(ResNet, self).__init__()
        self.in_planes = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        self.linear = nn.Linear(512*block.expansion, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out

my_cnn = ResNet(BasicBlock, [2, 2, 2, 2]).to(device)
```

Trainer and Tester code

```
def trainer(train_loader, valid_loader, model, config, device, weight=None):

    criterion = nn.CrossEntropyLoss(reduction='mean', weight=weight)
```

```
    optimizer = torch.optim.SGD(model.parameters(), lr=config['learning_rate'], momentum=0

    if not os.path.isdir('./models'):
        os.mkdir('./models') # Create directory of saving models.

    n_epochs, best_loss, step, early_stop_count = config['n_epochs'], math.inf, 0, 0

    for epoch in range(n_epochs):
        model.train() # Set your model to train mode.
        loss_record = []

        # tqdm is a package to visualize your training progress.
        train_pbar = tqdm(train_loader, position=0, leave=True)

        for x, y in train_pbar:
            optimizer.zero_grad()               # Set gradient to zero.
            x, y = x.to(device), y.to(device)   # Move your data to device.
            pred = model(x)
            loss = criterion(pred, y)
            loss.backward()                     # Compute gradient(backpropagation).
            optimizer.step()                    # Update parameters.
            step += 1
            loss_record.append(loss.detach().item())

            # Display current epoch number and loss on tqdm progress bar.
            train_pbar.set_description(f'Epoch [{epoch+1}/{n_epochs}]')
            train_pbar.set_postfix({'loss': loss.detach().item()})

        mean_train_loss = sum(loss_record)/len(loss_record)
        # writer.add_scalar('Loss/train', mean_train_loss, step)

        model.eval() # Set your model to evaluation mode.
        loss_record = []
        val_accuracy = []
        for x, y in valid_loader:
            x, y = x.to(device), y.to(device)
            with torch.no_grad():
                pred = model(x)
                loss = criterion(pred, y)

                _, predicted = torch.max(pred.data, 1)
                val_accuracy.append((predicted == y).sum().item() / predicted.size(0))
            loss_record.append(loss.item())
        print('Accuracy:', sum(val_accuracy)/len(val_accuracy))

    torch.save(model.state_dict(), config['save_path'])

def tester(test_loader, model, config, device):

    model.eval() # Set your model to evaluation mode.
    loss_record = []
```

```
    test_accuracy = []
    for x, y in test_loader:
        x, y = x.to(device), y.to(device)
        with torch.no_grad():
            pred = model(x)
            _, predicted = torch.max(pred.data, 1)
            test_accuracy.append((predicted == y).sum().item() / predicted.size(0))
    print(sum(test_accuracy)/len(test_accuracy))
```

Sample Config Dict

```
config = {
    'seed': 1968990,      # Your seed number, you can pick your lucky number. :)
    'valid_ratio': 0.2,   # validation_size = train_size * valid_ratio
    'n_epochs': 50,      # Number of epochs.
    'batch_size': 32,
    'learning_rate': 0.01,
    'early_stop': 20,    # If model has not improved for this many consecutive epochs, sto
    'save_path': './models/baseline_model.ckpt'  # Your model will be saved here.
}
```

Prepare the Dataloader

```
# Original CIFAR-30
cifar30_train_data, cifar30_valid_data = random_split(cifar30_trainset, [0.8, 0.2])

train_loader = torch.utils.data.DataLoader(cifar30_train_data, batch_size=config['batch_si
valid_loader = torch.utils.data.DataLoader(cifar30_valid_data, batch_size=config['batch_si

# Imbalanced CIFAR-30
im_cifar30_train_data, im_cifar30_valid_data = random_split(im_cifar30_trainset, [0.8, 0.2

im_train_loader = torch.utils.data.DataLoader(im_cifar30_train_data, batch_size=config['ba
im_valid_loader = torch.utils.data.DataLoader(im_cifar30_valid_data, batch_size=config['ba

# CIFAR-30 Testing (always balanced)
test_loader = torch.utils.data.DataLoader(cifar30_testset, batch_size=config['batch_size']
```

Train on original CIFAR-30

```
trainer(train_loader, valid_loader, my_cnn, config, device)
```

```
Epoch [23/50]: 100%|██████████| 375/375 [00:31<00:00, 11.87it/s, loss=0.943]
Accuracy: 0.583665780141844
Epoch [24/50]: 100%|██████████| 375/375 [00:31<00:00, 11.77it/s, loss=0.86]
Accuracy: 0.6031693262411347
Epoch [25/50]: 100%|██████████| 375/375 [00:30<00:00, 12.12it/s, loss=0.942]
Accuracy: 0.602947695035461
Epoch [26/50]: 100%|██████████| 375/375 [00:31<00:00, 11.85it/s, loss=0.816]
Accuracy: 0.5910904255319149
Epoch [27/50]: 100%|██████████| 375/375 [00:30<00:00, 12.32it/s, loss=0.559]
Accuracy: 0.6006205673758865
Epoch [28/50]: 100%|██████████| 375/375 [00:31<00:00, 12.02it/s, loss=0.877]
Accuracy: 0.5875443262411347
Epoch [29/50]: 100%|██████████| 375/375 [00:31<00:00, 12.06it/s, loss=0.758]
Accuracy: 0.5957446808510638
Epoch [30/50]: 100%|██████████| 375/375 [00:31<00:00, 12.06it/s, loss=1.17]
Accuracy: 0.6067154255319149
Epoch [31/50]: 100%|██████████| 375/375 [00:30<00:00, 12.17it/s, loss=0.764]
Accuracy: 0.596520390070922
Epoch [32/50]: 100%|██████████| 375/375 [00:31<00:00, 11.92it/s, loss=0.583]
Accuracy: 0.6124778368794326
Epoch [33/50]: 100%|██████████| 375/375 [00:31<00:00, 11.85it/s, loss=0.395]
Accuracy: 0.6164671985815603
Epoch [34/50]: 100%|██████████| 375/375 [00:31<00:00, 12.03it/s, loss=0.89]
Accuracy: 0.6170212765957447
Epoch [35/50]: 100%|██████████| 375/375 [00:31<00:00, 11.92it/s, loss=1]
Accuracy: 0.5980718085106383
Epoch [36/50]: 100%|██████████| 375/375 [00:31<00:00, 11.96it/s, loss=0.743]
Accuracy: 0.6072695035460993
Epoch [37/50]: 100%|██████████| 375/375 [00:31<00:00, 12.00it/s, loss=0.882]
Accuracy: 0.6263297872340425
Epoch [38/50]: 100%|██████████| 375/375 [00:31<00:00, 12.00it/s, loss=0.608]
Accuracy: 0.6105939716312057
Epoch [39/50]: 100%|██████████| 375/375 [00:31<00:00, 11.98it/s, loss=0.388]
Accuracy: 0.6344193262411347
Epoch [40/50]: 100%|██████████| 375/375 [00:30<00:00, 12.12it/s, loss=0.495]
Accuracy: 0.6052748226950355
Epoch [41/50]: 100%|██████████| 375/375 [00:31<00:00, 12.08it/s, loss=0.373]
Accuracy: 0.6107047872340425
Epoch [42/50]: 100%|██████████| 375/375 [00:30<00:00, 12.23it/s, loss=0.635]
Accuracy: 0.6237810283687943
Epoch [43/50]: 100%|██████████| 375/375 [00:30<00:00, 12.23it/s, loss=0.49]
Accuracy: 0.6204565602836879
Epoch [44/50]: 100%|██████████| 375/375 [00:30<00:00, 12.24it/s, loss=0.554]
Accuracy: 0.6155806737588653
Epoch [45/50]: 100%|██████████| 375/375 [00:30<00:00, 12.22it/s, loss=0.445]
Accuracy: 0.6365248226950355
Epoch [46/50]: 100%|██████████| 375/375 [00:31<00:00, 11.96it/s, loss=0.564]
Accuracy: 0.6278812056737589
Epoch [47/50]: 100%|██████████| 375/375 [00:31<00:00, 11.93it/s, loss=0.528]
Accuracy: 0.6257757092198581
Epoch [48/50]: 100%|██████████| 375/375 [00:31<00:00, 11.90it/s, loss=0.482]
Accuracy: 0.635084219858156
Epoch [49/50]: 100%|██████████| 375/375 [00:31<00:00, 12.03it/s, loss=0.364]
Accuracy: 0.6355274822695036
Epoch [50/50]: 100%|██████████| 375/375 [00:31<00:00, 11.83it/s, loss=0.399]
Accuracy: 0.6380762411347517
```

```
tester(test_loader, my_cnn, config, device)
```

```
0.6729831560283688
```

```
my_cnn = ResNet(BasicBlock, [2, 2, 2, 2]).to(device)
trainer(im_train_loader, im_valid_loader, my_cnn, config, device)
```

```
Epoch [1/50]: 100%|██████████| 85/85 [00:06<00:00, 12.25it/s, loss=2.22]
Accuracy: 0.3080357142857143
Epoch [2/50]: 100%|██████████| 85/85 [00:06<00:00, 12.21it/s, loss=3.29]
Accuracy: 0.30357142857142855
Epoch [3/50]: 100%|██████████| 85/85 [00:07<00:00, 11.84it/s, loss=2.05]
Accuracy: 0.28422619047619047
Epoch [4/50]: 100%|██████████| 85/85 [00:06<00:00, 12.92it/s, loss=2.64]
Accuracy: 0.31101190476190477
Epoch [5/50]: 100%|██████████| 85/85 [00:07<00:00, 11.96it/s, loss=4.87]
Accuracy: 0.3869047619047619
Epoch [6/50]: 100%|██████████| 85/85 [00:07<00:00, 11.94it/s, loss=2.99]
Accuracy: 0.43898809523809523
Epoch [7/50]: 100%|██████████| 85/85 [00:06<00:00, 12.85it/s, loss=1.19]
Accuracy: 0.3556547619047619
Epoch [8/50]: 100%|██████████| 85/85 [00:08<00:00, 10.40it/s, loss=1.29]
Accuracy: 0.3943452380952381
Epoch [9/50]: 100%|██████████| 85/85 [00:07<00:00, 11.87it/s, loss=0.536]
Accuracy: 0.3898809523809524
Epoch [10/50]: 100%|██████████| 85/85 [00:06<00:00, 12.86it/s, loss=3]
Accuracy: 0.3869047619047619
Epoch [11/50]: 100%|██████████| 85/85 [00:07<00:00, 11.80it/s, loss=1.37]
Accuracy: 0.38244047619047616
Epoch [12/50]: 100%|██████████| 85/85 [00:07<00:00, 11.81it/s, loss=1.09]
Accuracy: 0.4479166666666667
Epoch [13/50]: 100%|██████████| 85/85 [00:06<00:00, 12.78it/s, loss=1.15]
Accuracy: 0.41964285714285715
Epoch [14/50]: 100%|██████████| 85/85 [00:07<00:00, 11.94it/s, loss=1.91]
Accuracy: 0.4226190476190476
Epoch [15/50]: 100%|██████████| 85/85 [00:06<00:00, 12.16it/s, loss=1.11]
Accuracy: 0.34672619047619047
Epoch [16/50]: 100%|██████████| 85/85 [00:06<00:00, 12.76it/s, loss=1.96]
Accuracy: 0.44345238095238093
Epoch [17/50]: 100%|██████████| 85/85 [00:07<00:00, 11.76it/s, loss=3.12]
Accuracy: 0.40029761904761907
Epoch [18/50]: 100%|██████████| 85/85 [00:06<00:00, 12.36it/s, loss=3.95]
Accuracy: 0.24851190476190477
Epoch [19/50]: 100%|██████████| 85/85 [00:06<00:00, 12.68it/s, loss=1.04]
Accuracy: 0.4330357142857143
Epoch [20/50]: 100%|██████████| 85/85 [00:07<00:00, 11.76it/s, loss=3.29]
Accuracy: 0.40625
Epoch [21/50]: 100%|██████████| 85/85 [00:06<00:00, 12.67it/s, loss=1.93]
Accuracy: 0.34970238095238093
Epoch [22/50]: 100%|██████████| 85/85 [00:07<00:00, 11.91it/s, loss=1.53]
Accuracy: 0.4226190476190476
Epoch [23/50]: 100%|██████████| 85/85 [00:07<00:00, 11.82it/s, loss=0.827]
Accuracy: 0.39732142857142855
Epoch [24/50]: 100%|██████████| 85/85 [00:06<00:00, 12.34it/s, loss=0.763]
```

```
Accuracy: 0.46726190476190477
Epoch [25/50]: 100%|██████████| 85/85 [00:07<00:00, 11.78it/s, loss=1.64]
Accuracy: 0.35714285714285715
Epoch [26/50]: 100%|██████████| 85/85 [00:07<00:00, 11.67it/s, loss=1.86]
Accuracy: 0.47172619047619047
Epoch [27/50]: 100%|██████████| 85/85 [00:06<00:00, 12.72it/s, loss=3.94]
Accuracy: 0.3630952380952381
Epoch [28/50]: 100%|██████████| 85/85 [00:07<00:00, 12.07it/s, loss=1.9]
Accuracy: 0.44642857142857145
Epoch [29/50]: 100%|██████████| 85/85 [00:07<00:00, 11.67it/s, loss=1.97]
Accuracy: 0.43898809523809523
```

```
tester(test_loader, my_cnn, config, device)
```

```
0.2888962765957447
```

1-b. Implement Re-Weighting

Hint:

Notice there is a "weight" argument for the loss we use:

```
criterion = nn.CrossEntropyLoss(reduction='mean', weight=weight)
```

```
# Please do not modify the config
re_weighting_config = {
    'seed': 1968990,      # Your seed number, you can pick your lucky number. :)
    'select_all': True,   # Whether to use all features.
    'valid_ratio': 0.2,   # validation_size = train_size * valid_ratio
    'n_epochs': 50,      # Number of epochs.
    'batch_size': 32,
    'learning_rate': 0.001,
    'early_stop': 20,    # If model has not improved for this many consecutive epochs, sto
    'save_path': './models/re_weighting_model.ckpt'  # Your model will be saved here.
}

# TODO

# ENDS HERE
```

```
x = list(map(float, np.divide(im_training_distribution, sum(im_training_distribution))))

beta = np.ones(30) * min(x)/max(x)
print(beta)
weightVals = (1-beta)/(1-(beta**im_training_distribution))
weightVals = list(map(float, weightVals))

weight = torch.tensor(weightVals).to(device)
```

```
my_cnn_re_weighting = ResNet(BasicBlock, [2, 2, 2, 2]).to(device)

trainer(im_train_loader, im_valid_loader, my_cnn_re_weighting, re_weighting_config, device
```

```
Epoch [22/50]: 100%|██████████| 85/85 [00:07<00:00, 11.77it/s, loss=2.78]
Accuracy: 0.49255952380952384
Epoch [23/50]: 100%|██████████| 85/85 [00:06<00:00, 12.31it/s, loss=1.8]
Accuracy: 0.4895833333333333
Epoch [24/50]: 100%|██████████| 85/85 [00:06<00:00, 12.36it/s, loss=2.57]
Accuracy: 0.4851190476190476
Epoch [25/50]: 100%|██████████| 85/85 [00:07<00:00, 11.95it/s, loss=2.94]
```

```
Epoch [48/50]: 100%|██████████| 85/85 [00:07<00:00, 11.81it/s, loss=4.17]
Accuracy: 0.5
Epoch [49/50]: 100%|██████████| 85/85 [00:07<00:00, 11.97it/s, loss=2.97]
Accuracy: 0.4583333333333333
Epoch [50/50]: 100%|██████████| 85/85 [00:06<00:00, 12.47it/s, loss=0.385]
Accuracy: 0.5431547619047619
```

1-c. Evaluate Re-Weighting

```
tester(test_loader, my_cnn_re_weighting, re_weighting_config, device)
```

```
0.29388297872340424
```

As mentioned in office hours, this is a slight improvement over the un-weighted value.

1-d. Implement Re-Sampling

Hint:

Check out how sampler works in PyTorch's DataLoader!

```
sampler = WeightedRandomSampler(weights=sample_weights, num_samples=len(class_counts) * 500)
rs_train_loader = DataLoader(im_cifar30_trainset, batch_size=config['batch_size'], sampler=sampler)
```

```
# Please do not modify the config
config = {
    'seed': 1968990,      # Your seed number, you can pick your lucky number. :)
    'select_all': True,   # Whether to use all features.
    'valid_ratio': 0.2,   # validation_size = train_size * valid_ratio
    'n_epochs': 50,      # Number of epochs.
    'batch_size': 32,
    'learning_rate': 0.001,
    'early_stop': 20,    # If model has not improved for this many consecutive epochs, sto
    'save_path': './models/re_sampling_model.ckpt'  # Your model will be saved here.
}


# TODO

# ENDS HERE
```

```
from torch.utils.data import WeightedRandomSampler
```

```
sample_weights = np.ones(sum(im_training_distribution)) / sum(im_training_distribution)
len(sample_weights)
```

```
3362
```

```
sampler = WeightedRandomSampler(weights=sample_weights, num_samples=len(im_training_distri
rs_train_loader = DataLoader(im_cifar30_trainset, batch_size=config['batch_size'], sampler
rs_valid_loader = DataLoader(im_cifar30_valid_data, batch_size=config['batch_size'], sampl
```

```
my_cnn_re_sampling = ResNet(BasicBlock, [2, 2, 2, 2]).to(device)
# trainer(rs_train_loader, rs_valid_loader, my_cnn_re_sampling, config, device)
trainer(rs_train_loader, test_loader, my_cnn_re_sampling, config, device)
```

```
Epoch [42/50]: 100%|██████████| 469/469 [00:38<00:00, 12.05it/s, loss=0.0141]
Accuracy: 0.3324468085106383
Epoch [43/50]: 100%|██████████| 469/469 [00:38<00:00, 12.05it/s, loss=0.0586]
Accuracy: 0.3503989361702128
Epoch [44/50]: 100%|██████████| 469/469 [00:38<00:00, 12.07it/s, loss=0.104]
Accuracy: 0.34075797872340424
Epoch [45/50]: 100%|██████████| 469/469 [00:38<00:00, 12.11it/s, loss=0.174]
Accuracy: 0.3507313829787234
Epoch [46/50]: 100%|██████████| 469/469 [00:38<00:00, 12.09it/s, loss=0.0262]
Accuracy: 0.3394281914893617
Epoch [47/50]: 100%|██████████| 469/469 [00:39<00:00, 12.02it/s, loss=0.0052]
Accuracy: 0.3460771276595745
Epoch [48/50]: 100%|██████████| 469/469 [00:38<00:00, 12.08it/s, loss=0.0237]
Accuracy: 0.3527260638297872
Epoch [49/50]: 100%|██████████| 469/469 [00:39<00:00, 11.96it/s, loss=0.162]
Accuracy: 0.34873670212765956
Epoch [50/50]: 100%|██████████| 469/469 [00:40<00:00, 11.72it/s, loss=0.0149]
Accuracy: 0.34541223404255317
```

1-e. Evaluate Re-Sampling

```
tester(test_loader, my_cnn_re_sampling, config, device)
```

```
0.34541223404255317
```

```
Start coding or generate with AI.
```