EE443 Spring 2023 - Homework 3
Jordan White and David Zoro

Note: The starter code was in a different order than the Homework 3 assignment pdf. This Report is organized to match the structure of the assignment pdf, where P2-b is re-sampling and P2-d is re-weighting.

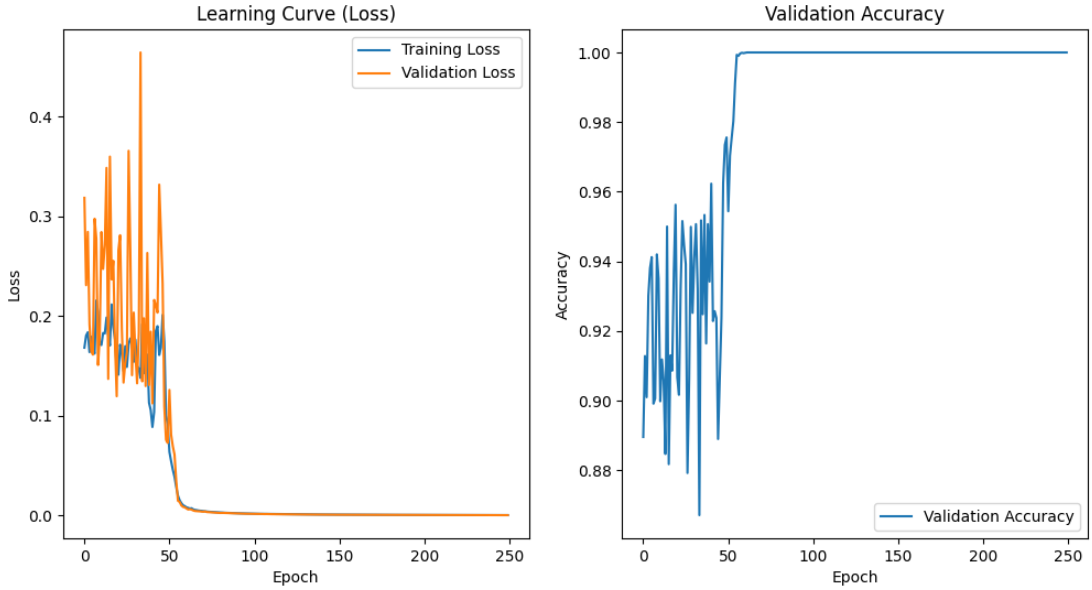
Problem 1: Convolutional Neural Network

1-a)

```python
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        # Read through official documentation: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html
        # and make sure you understand the concept of 2D convolution and Conv2d layer's definition
        # Here is also some cool animation: https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaafb0b148
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)

        output_dim_after_conv = 400 # total number of elements in the output
        output_dim_after_fc = 10 # number of output nodes in the final fully connected layer

        self.fc1 = nn.Linear(output_dim_after_conv, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, output_dim_after_fc)

    def forward(self, x):
        # TODO: build your forward function
        # 1. First pass the x through the first conv layer followed by relu then pooling
        x = self.pool(F.relu(self.conv1(x)))
        # 2. Then pass the output of step 1 through the second conv layer followed by relu then pooling
        x = self.pool(F.relu(self.conv2(x)))
        # 3. (important) Since we are now moving onto fully connected layer, we need to flatten the tensor,
        # 3. use `x = torch.flatten(x, 1)` with the output from step 2
        x = torch.flatten(x, 1)
        # 4 Finally pass the output of step 3 into `fc1`, `fc2` and `fc3 (remeber the relu!)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x) # this is the output layer, so no ReLU
        return x

my_cnn = Net().to(device)  # operate on GPU
```

```
[ ] print(my_cnn)

    Net(
      (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
      (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
      (fc1): Linear(in_features=400, out_features=120, bias=True)
      (fc2): Linear(in_features=120, out_features=84, bias=True)
      (fc3): Linear(in_features=84, out_features=10, bias=True)
    )
```

1-b)




```python
def plot_learning_curve(train_loss_history, valid_loss_history, valid_accuracy_history):
    # Set up plot figure
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

    # Plot learning curve (loss)
    ax1.plot(train_loss_history, label='Training Loss')
    ax1.plot(valid_loss_history, label='Validation Loss')
    ax1.set_title('Learning Curve (Loss)')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.legend(loc='best')

    # Plot validation accuracy
    ax2.plot(valid_accuracy_history, label='Validation Accuracy')
    ax2.set_title('Validation Accuracy')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy')
    ax2.legend(loc='best')

    # Show plots
    plt.show()

# Call the plot_learning_curve function with your data
train_loss_history, valid_loss_history, valid_accuracy_history = trainer(train_loader, valid_loader, my_cnn, config, device)
plot_learning_curve(train_loss_history, valid_loss_history, valid_accuracy_history)
```

1-c)

```python
# TODO 1-c
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()

        # First convolutional layer with kernel_size=3, stride, and padding=1
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        # Batch normalization for the first convolutional layer
        self.bn1 = nn.BatchNorm2d(out_channels)

        # Second convolutional layer with kernel_size=3, stride=1, and padding=1
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        # Batch normalization for the second convolutional layer
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Define the shortcut connection (identity or projection)
        self.shortcut = nn.Sequential()
        # If stride is not 1 or in_channels is not equal to out_channels, create a projection shortcut
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                # 1x1 convolutional layer to match the dimensions
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                # Batch normalization for the 1x1 convolutional layer
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        # Apply the first convolutional layer, followed by batch normalization and ReLU activation
        out = F.relu(self.bn1(self.conv1(x)))
        # Apply the second convolutional layer, followed by batch normalization
        out = self.bn2(self.conv2(out))
        # Add the shortcut connection (either identity or projection) to the output
        out += self.shortcut(x)
        # Apply ReLU activation to the output
        out = F.relu(out)
        return out
```

```python
class your_cnn(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu1 = nn.ReLU()
        self.layer1 = nn.Sequential(
            ResidualBlock(64, 64),
            ResidualBlock(64, 64)
        )
        self.pool1 = nn.MaxPool2d(2, 2)

        self.layer2 = nn.Sequential(
            ResidualBlock(64, 128, stride=2),
            ResidualBlock(128, 128)
        )
        self.pool2 = nn.MaxPool2d(2, 2)

        output_dim_after_conv = 128 * 4 * 4
        output_dim_after_fc = 10

        self.fc1 = nn.Linear(output_dim_after_conv, 256)
        self.relu2 = nn.ReLU()
        self.dropout1 = nn.Dropout(p=0.3)

        self.fc2 = nn.Linear(256, output_dim_after_fc)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)

        x = self.layer1(x)
        x = self.pool1(x)

        x = self.layer2(x)
        x = self.pool2(x)

        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.relu2(x)
        x = self.dropout1(x)
```

```
[13]            x = self.fc2(x)
                return x



    your_config = {
        'seed': 8471220,      # Your seed number, you can pick your lucky number. :)
        'valid_ratio': 0.2,   # validation_size = train_size * valid_ratio
        'n_epochs': 80,      # Number of epochs.
        'batch_size': 256,
        'learning_rate': 0.01,
        'early_stop': 25,    # If model has not improved for this many consecutive epochs, stop training.
        'save_path': './models/model.ckpt'  # Your model will be saved here.
    }

    cifar_train_data, cifar_valid_data = random_split(cifar_trainset, [1-your_config['valid_ratio'], your_config['valid_ratio']])

    train_loader = torch.utils.data.DataLoader(cifar_trainset, batch_size=your_config['batch_size'], shuffle=True)
    valid_loader = torch.utils.data.DataLoader(cifar_valid_data, batch_size=your_config['batch_size'], shuffle=True)
    test_loader = torch.utils.data.DataLoader(cifar_testset, batch_size=your_config['batch_size'], shuffle=False)

    this_cnn = your_cnn()

    # Move the neural network to the GPU
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    this_cnn.to(device)

    trainer(train_loader, valid_loader, this_cnn, your_config, device)

    correct = 0
    total = 0
    # since we're not training, we don't need to calculate the gradients for our outputs
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            # calculate outputs by running images through the network
            outputs = this_cnn(images)
            # the class with the highest energy is what we choose as prediction
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')
```

This modified model introduces the following changes:

1. The model now uses a custom ResidualBlock class that represents a residual block in a ResNet architecture. This block consists of two convolutional layers with batch normalization and ReLU activation, followed by a shortcut connection (either identity or projection).
2. The your_cnn class now has a different architecture:
    - It starts with a single convolutional layer, followed by batch normalization and ReLU activation.
    - Then it has two residual blocks with 64 channels each, followed by a max pooling layer.
    - After that, it has another two residual blocks with 128 channels each, followed by another max pooling layer.
    - The fully connected layers now have an additional ReLU activation and dropout layer (with dropout probability of 0.3) between the first and second fully connected layers.
3. In the training configuration, the number of epochs is increased to 80, which may help the model achieve better performance.

This a picture of the end of the training of the model and then the accuracy of the model on 10,000 test images, which was **81%**

```
Epoch [76/80]: 100%|██████████| 196/196 [00:27<00:00,  7.17it/s, loss=0.000257]
1.0
Epoch [77/80]: 100%|██████████| 196/196 [00:27<00:00,  7.10it/s, loss=0.000352]
1.0
Epoch [78/80]: 100%|██████████| 196/196 [00:27<00:00,  7.15it/s, loss=0.000341]
1.0
Epoch [79/80]: 100%|██████████| 196/196 [00:27<00:00,  7.09it/s, loss=0.000302]
1.0
Epoch [80/80]: 100%|██████████| 196/196 [00:27<00:00,  7.19it/s, loss=0.000267]
1.0
Accuracy of the network on the 10000 test images: 81 %
```

```
print(this_cnn)
```

```
your_cnn(
  (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): ReLU()
  (layer1): Sequential(
    (0): ResidualBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
    (1): ResidualBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
  )
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (layer2): Sequential(
    (0): ResidualBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): ResidualBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
  )
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=2048, out_features=256, bias=True)
  (relu2): ReLU()
  (dropout1): Dropout(p=0.3, inplace=False)
  (fc2): Linear(in_features=256, out_features=10, bias=True)
)
```

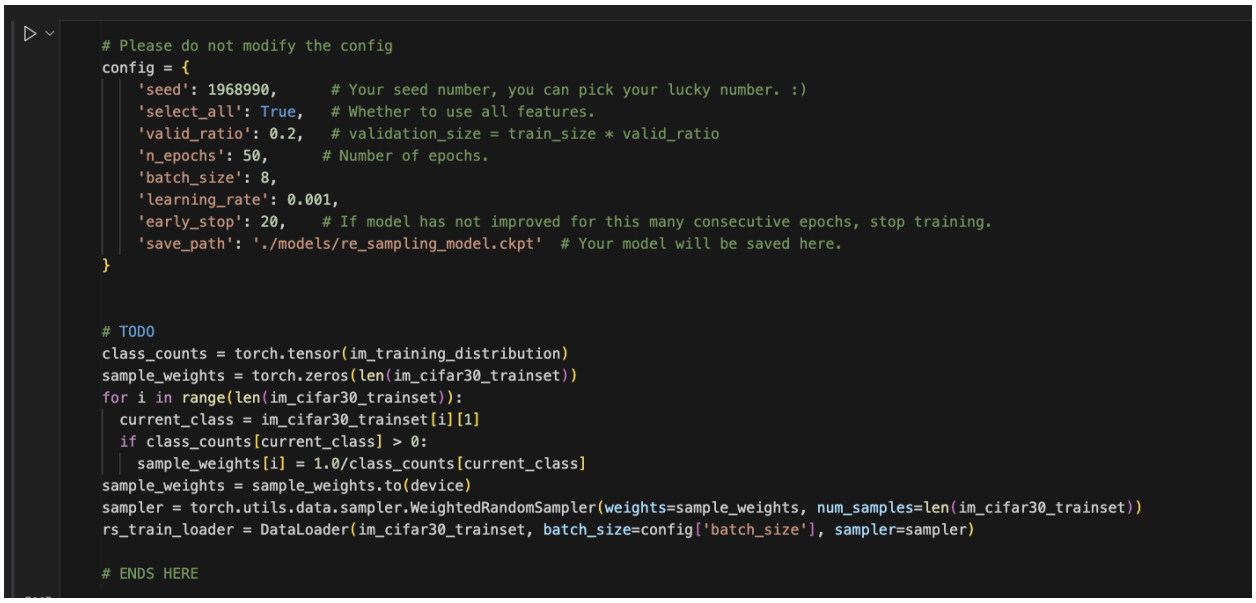
Problem 2: Long-Tailed Recognition

2-a)

The accuracy of the CNN baseline was 0.3191489361702128. Observations were that the process of using the trainer took a long time and the accuracies from the trainer got better gradually from the first epoch to the 50th.

2-b)

We used the Weighted Random Sampling method.



```
# 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': 8,
    'learning_rate': 0.001,
    'early_stop': 20,    # If model has not improved for this many consecutive epochs, stop training.
    'save_path': './models/re_sampling_model.ckpt'  # Your model will be saved here.
}


# TODO
class_counts = torch.tensor(im_training_distribution)
sample_weights = torch.zeros(len(im_cifar30_trainset))
for i in range(len(im_cifar30_trainset)):
  current_class = im_cifar30_trainset[i][1]
  if class_counts[current_class] > 0:
    sample_weights[i] = 1.0/class_counts[current_class]
sample_weights = sample_weights.to(device)
sampler = torch.utils.data.sampler.WeightedRandomSampler(weights=sample_weights, num_samples=len(im_cifar30_trainset))
rs_train_loader = DataLoader(im_cifar30_trainset, batch_size=config['batch_size'], sampler=sampler)

# ENDS HERE
```

This snippet of code implements a form of resampling called Weighted Random Sampling to tackle the issue of class imbalance in the dataset.

`class_counts` is a tensor that contains the count of each class in the dataset. `sample_weights` is a tensor that will hold the weights for each example in the dataset.

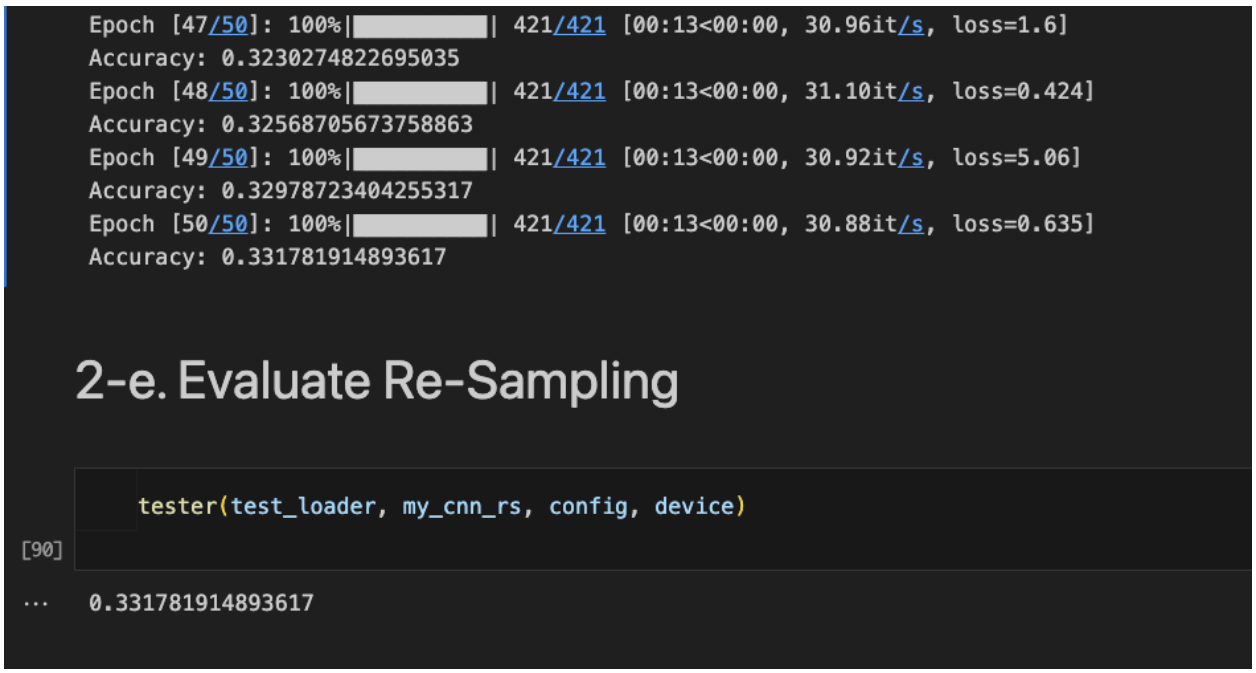
In the for loop, each example in the dataset (`im_cifar30_trainset`) is iterated over. The class of the current example is fetched with `im_cifar30_trainset[i][1]`. Then, an if statement checks if the count of the current class is greater than zero. If it is, the weight for the current example is calculated as the inverse of its class's frequency (i.e., `1.0/class_counts[current_class]`). This weight is then stored in `sample_weights[i]`.

Once the weights for all examples have been computed, `sample_weights` is moved to the device (which can be a GPU or CPU).

A `WeightedRandomSampler` is then instantiated with these weights and the total number of examples in the dataset. This sampler will randomly draw samples for each batch based on their weights. That is, samples from less frequent (underrepresented) classes, which have higher weights, will have a higher chance of being selected.

Finally, a DataLoader (`rs_train_loader`) is created. This DataLoader uses the dataset and the sampler, and it will load data in batches of size defined by `config['batch_size']`. The DataLoader will use the sampler to decide which examples to include in each batch, making sure that each batch is a representative mix of different classes according to their weights.

2-c)



Accuracy was over 30%!

2-d)

```
# 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, stop training.
    'save_path': './models/augmentation_model.ckpt'  # Your model will be saved here.
}

# TODO
train_transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])

"""
Different implementations
cost_sensitive = torch.div(torch.min(class_counts),class_counts)
class_balanced = torch.mul(torch.div((1.0-beta), (1.0 - torch.pow(beta, class_counts))))
"""
im_train_loader = DataLoader(im_cifar30_trainset, batch_size=augmentation_config['batch_size'], shuffle=True)
im_valid_loader = DataLoader(cifar30_testset, batch_size=augmentation_config['batch_size'], shuffle=False)

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)

# ENDS HERE
```

We simply used regular cross-entropy. We also were able to implement other methods. For illustration, we had implementations for cost-sensitive and class balanced loss prepared.

2-e)

▾ 2-c. Evaluate Re-Weighting

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

```
0.7442375886524822
```

We were able to achieve an accuracy above 30%.

Problem 3: Generative Adversarial Network

3-a)

In [4]:
```
# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = "mps" if torch.backends.mps.is_available() else "cpu"
print(device)

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[192:], padding=2, normalize=True).cpu(),(1,2,0)))
```

```
mps
```

3-b)

2. Generator

The generator G, is designed to map the latent space vector z to data-space. Since our data are images, converting z to data-space means ultimately creating an RGB image with the same size as the training images (i.e. 3x64x64). In practice, this is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2D batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of $[-1, 1]$.

In [6]:
```
class Generator(nn.Module):
  def __init__(self, latent_dim=100):
    super(Generator, self).__init__()
    # TODO: finish implementing the network architecture
    self.seq = nn.Sequential(
      # TODO: the first ConvTranspose2d layer will take the celebA images as input with size of 3x64x64
      # so you will need at least one ConvTranspose2d layer, one BatchNorm2d layer followed by a ReLU activation here
      nn.ConvTranspose2d(latent_dim, 512, kernel_size=(4, 4), stride=(1,1), padding=(0,0), bias=False),
      nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
      nn.ReLU(),

      # ENDS HERE
      nn.ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
      nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
      nn.ReLU(),
      nn.ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
      nn.BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
      nn.ReLU(),
      nn.ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
      nn.BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
      nn.ReLU(),
      nn.ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
      nn.Tanh()
    )

  def forward(self, input):
    x = self.seq(input)
    return x
```

In [7]:
```
# Create the generator
netG = Generator().to(device)

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netG.apply(weights_init)

# Print the model
print(netG)
```

```
Generator(
  (seq): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

3. Discriminator

The discriminator D is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real (as opposed to fake). Here, D takes a 3x64x64 input image, processes it through a series of Conv2d, BatchNorm2d, and ReLU layers, and outputs the final probability through a Sigmoid activation function. This architecture can be extended with more layers if necessary for the problem, but there is significance to the use of the strided convolution, BatchNorm, and ReLUs.

In [8]:
```
class Discriminator(nn.Module):
  def __init__(self):
    super(Discriminator, self).__init__()
    # TODO: finish implementing the network architecture
    self.seq = nn.Sequential(
        nn.Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
        nn.ReLU(),
        nn.Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
        nn.BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
        nn.ReLU(),
        nn.Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
        nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
        nn.ReLU(),
        nn.Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
        nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
        nn.ReLU(),
        # TODO: the last Conv2d layer of the doscriminator followed by a sigmoid
        nn.Conv2d(512, 1, kernel_size=(4, 4), stride=(1,1), padding=(0,0), bias=False),
        nn.Sigmoid()
        # ENDS HERE
    )

  def forward(self, input):
    x = self.seq(input)
    return x
```

In [9]:
```
# Create the Discriminator
netD = Discriminator().to(device)

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)
```

```
Discriminator(
  (seq): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): ReLU()
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): ReLU()
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): ReLU()
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU()
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

Loss Functions and Optimizers

With D and G setup, we can specify how they learn through the loss functions and optimizers. We will use the Binary Cross Entropy loss (`BCELoss`) function which is defined in PyTorch as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right]$$

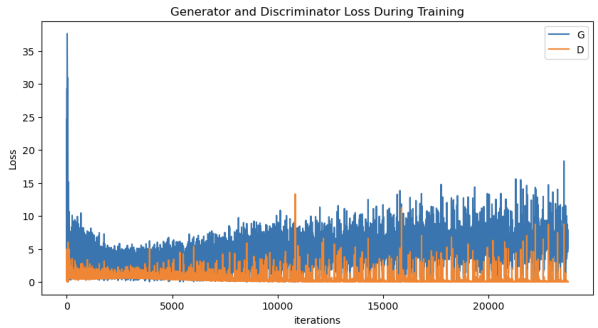
Notice how this function provides the calculation of both log components in the objective function (i.e. $\log(D(x))$ and $\log(1 - D(G(z)))$). We can specify what part of the BCE equation to use with the y input. This is accomplished in the training loop which is coming up soon, but it is important to understand how we can choose which component we wish to calculate just by changing y (i.e. GT labels).

In [10]:
```
# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

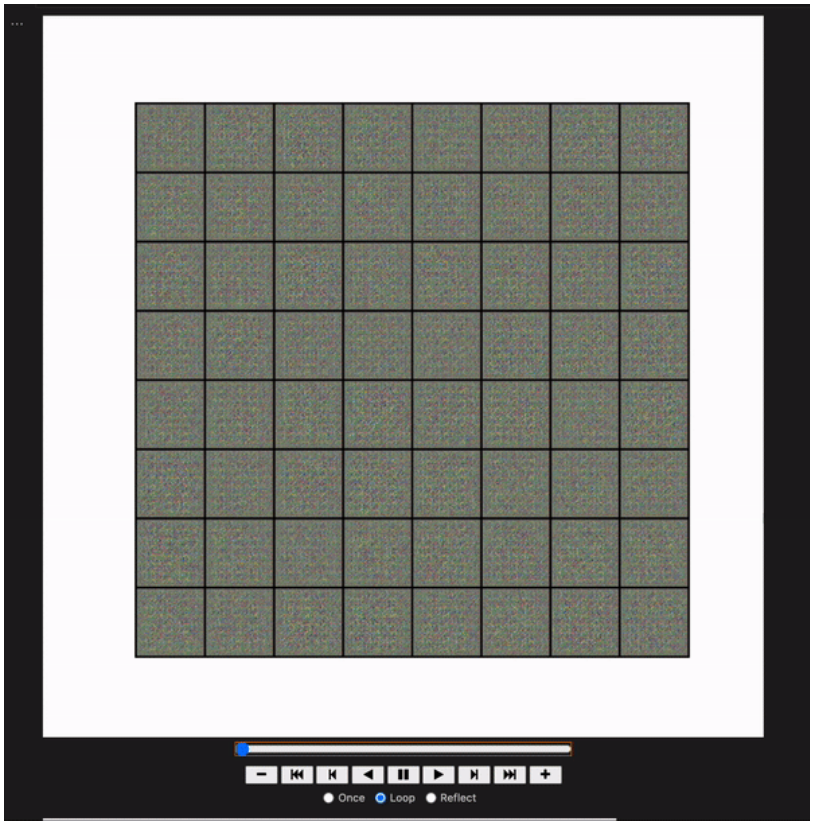
In [11]:
```
# Training Loop
num_epochs = 30

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################################################
        # 1. Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ############################################################

        # Initialize the gradient of D
        netD.zero_grad()

        # 1.1 Calculate the loss with all-real batch

        # Format batch
        real_batch = data[0].to(device)
        b_size = real_batch.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
        # Forward pass real batch through D
        output = netD(real_batch).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        # 1.2 Calculate the loss with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G and a label array
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch
        errD_fake.backward()
        # Calculate the mean of output, D_G_z1, from netD with your fake batch
        D_G_z1 = output.mean().item()

        # 1.3 Combine the loss from all-real batch and all-fake batch
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D with step()
        optimizerD.step()
```

```
        ############################################################
        # 2. Update G network: maximize log(D(G(z)))
        ############################################################

        # TODO: Initialize the gradient of G
        netG.zero_grad()

        # TODO: create the label array, remember fake labels are real for generator cost
        label.fill_(real_label)

        # TODO: Since we just updated D, perform another forward pass of all-fake batch through D as output
        output = netD(fake).view(-1)

        # TODO: Calculate G's loss based on this output
        errG = criterion(output, label)

        # TODO: Calculate gradients using backward() for G's loss
        errG.backward()

        # TODO: Calculate the mean of output, D_G_z2, from netD with your fake batch
        D_G_z2 = output.mean().item()

        # TODO: Update G with step
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1
```

In [12]:
```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



In [19]:
```
fig = plt.figure(figsize=(8,8))
plt.axis("off")
plt.rcParams['animation.embed_limit'] = 70.0  # Set the limit to 50 MB
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())
```



3-c)

1. "Mode Collapse" in GAN training is when the generator network produces limited and similar-looking samples, ignoring the diversity of the true data distribution.

2. The training strategy for GAN needs to be carefully designed due to the model's complexity. Poorly designed strategies can cause instability, slow convergence, or mode collapse.

3. GANs have several applications besides generating fake images, such as image-to-image translation, data augmentation, and video generation.