Colab FAQ and Using GPU

For some basic overview and features offered in Colab notebooks, check out: Overview of Colaboratory Features.

You need to use the Colab GPU for this assignment by selecting:

> **Runtime $\rightarrow$ Change runtime type $\rightarrow$ Hardware Accelerator: GPU**

Download CIFAR and Colour dictionary

We will use the CIFAR-10 data set, which consists of images of size 32x32 pixels. For most of the questions we will use a subset of the dataset. To make the problem easier, we will only use the "Horse" category from this data set. Now let's learn to colour some horses!

The data loading script is included below. It can take up to a couple of minutes to download everything the first time.

All files are stored at `/content/csc413/a2/data/` folder.

Programming Assignment 2: Convolutional Neural Networks

Based on an assignment by Lisa Zhang

For CSC413/2516 in Winter 2021 with Professors Jimmy Ba and Bo Wang

Version 1.1

Changes by Version:

- Q.1 of D.1: add hints for the `train` function
- Q.2 of D.1: add step by step instructions for the layer replacement and gradient freezing
- Q.1 of D.2: add hints for the `compute_iou_loss` function
- Q.2 of D.2: add step by step instructions for the layer replacement and gradient freezing

Submission: You must submit two files through MarkUs: a PDF file containing your writeup, titled *a2-writeup.pdf*, and your code file *a2-cnn.ipynb*. Your writeup must be typeset.

The programming assignments are individual work. See the Course Syllabus for detailed policies.

Introduction:
This assignment will focus on the applications of convolutional neural networks in various image processing tasks. First, we will train a convolutional neural network for a task known as image

colourization. Given a greyscale image, we will predict the colour at each pixel. This a difficult problem for many reasons, one of which being that it is ill-posed: for a single greyscale image, there can be multiple, equally valid colourings.

In the second half of the assignment, we will perform fine-tuning on a pre-trained semantic segmentation model. Semantic segmentation attempts to clusters the areas of an image which belongs to the same object (label), and treats each pixel as a classification problem. We will fine-tune a pre-trained conv net featuring dilated convolution to segment flowers from the [Oxford17 flower dataset]

Helper code

You can ignore the restart warning.

```
######################################################################
# Setup working directory
######################################################################
%mkdir -p /content/csc413/a2/
%cd /content/csc413/a2

######################################################################
# Helper functions for loading data
######################################################################
# adapted from
# https://github.com/fchollet/keras/blob/master/keras/datasets/cifar10.py

import os
import pickle
import sys
import tarfile

import numpy as np
from PIL import Image
from six.moves.urllib.request import urlretrieve


def get_file(fname, origin, untar=False, extract=False, archive_format="auto", cache_d
    datadir = os.path.join(cache_dir)
    if not os.path.exists(datadir):
        os.makedirs(datadir)

    if untar:
        untar_fpath = os.path.join(datadir, fname)
        fpath = untar_fpath + ".tar.gz"
    else:
        fpath = os.path.join(datadir, fname)

    print("File path: %s" % fpath)
    if not os.path.exists(fpath):
```

```
    if not os.path.exists(fpath):
        print("Downloading data from", origin)

        error_msg = "URL fetch failure on {}: {} -- {}"
        try:
            try:
                urlretrieve(origin, fpath)
            except URLError as e:
                raise Exception(error_msg.format(origin, e.errno, e.reason))
            except HTTPError as e:
                raise Exception(error_msg.format(origin, e.code, e.msg))
        except (Exception, KeyboardInterrupt) as e:
            if os.path.exists(fpath):
                os.remove(fpath)
            raise

    if untar:
        if not os.path.exists(untar_fpath):
            print("Extracting file.")
            with tarfile.open(fpath) as archive:
                archive.extractall(datadir)
        return untar_fpath

    if extract:
        _extract_archive(fpath, datadir, archive_format)

    return fpath


def load_batch(fpath, label_key="labels"):
    """Internal utility for parsing CIFAR data.
    # Arguments
        fpath: path the file to parse.
        label_key: key for label data in the retrieve
            dictionary.
    # Returns
        A tuple `(data, labels)`.
    """
    f = open(fpath, "rb")
    if sys.version_info < (3,):
        d = pickle.load(f)
    else:
        d = pickle.load(f, encoding="bytes")
        # decode utf8
        d_decoded = {}
        for k, v in d.items():
            d_decoded[k.decode("utf8")] = v
        d = d_decoded
    f.close()
    data = d["data"]
    labels = d[label_key]
```

```
    data = data.reshape(data.shape[0], 3, 32, 32)
    return data, labels


def load_cifar10(transpose=False):
    """Loads CIFAR10 dataset.
    # Returns
        Tuple of Numpy arrays: `(x_train, y_train), (x_test, y_test)`.
    """
    dirname = "cifar-10-batches-py"
    origin = "http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"
    path = get_file(dirname, origin=origin, untar=True)

    num_train_samples = 50000

    x_train = np.zeros((num_train_samples, 3, 32, 32), dtype="uint8")
    y_train = np.zeros((num_train_samples,), dtype="uint8")

    for i in range(1, 6):
        fpath = os.path.join(path, "data_batch_" + str(i))
        data, labels = load_batch(fpath)
        x_train[(i - 1) * 10000 : i * 10000, :, :, :] = data
        y_train[(i - 1) * 10000 : i * 10000] = labels

    fpath = os.path.join(path, "test_batch")
    x_test, y_test = load_batch(fpath)

    y_train = np.reshape(y_train, (len(y_train), 1))
    y_test = np.reshape(y_test, (len(y_test), 1))

    if transpose:
        x_train = x_train.transpose(0, 2, 3, 1)
        x_test = x_test.transpose(0, 2, 3, 1)
    return (x_train, y_train), (x_test, y_test)
```

```
/content/csc413/a2
```

Download files

This may take 1 or 2 mins for the first time.

```
# Download cluster centers for k-means over colours
colours_fpath = get_file(
    fname="colours", origin="http://www.cs.toronto.edu/~jba/kmeans_colour_a2.tar.gz",
)
# Download CIFAR dataset
m = load_cifar10()
```

```
File path: data/colours.tar.gz
File path: data/cifar-10-batches-py.tar.gz
```

Image Colourization as Classification

We will select a subset of 24 colours and frame colourization as a pixel-wise classification problem, where we label each pixel with one of 24 colours. The 24 colours are selected using k-means clustering over colours, and selecting cluster centers.

This was already done for you, and cluster centers are provided in http://www.cs.toronto.edu/~jba/kmeans_colour_a2.tar.gz, which was downloaded by the helper functions above. For simplicity, we will measure distance in RGB space. This is not ideal but reduces the software dependencies for this assignment.

Helper code

```
"""
Colourization of CIFAR-10 Horses via classification.
"""
import argparse
import math
import time

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as npr
import scipy.misc
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable

# from load_data import load_cifar10

HORSE_CATEGORY = 7
```

Data related code

```
def get_rgb_cat(xs, colours):
    """
    Get colour categories given RGB values. This function doesn't
    actually do the work, instead it splits the work into smaller
    chunks that can fit into memory, and calls helper function
    _get_rgb_cat

    Args:
```

```
      xs: float numpy array of RGB images in [B, C, H, W] format
      colours: numpy array of colour categories and their RGB values
    Returns:
      result: int numpy array of shape [B, 1, H, W]
    """
    if np.shape(xs)[0] < 100:
        return _get_rgb_cat(xs)
    batch_size = 100
    nexts = []
    for i in range(0, np.shape(xs)[0], batch_size):
        next = _get_rgb_cat(xs[i : i + batch_size, :, :, :], colours)
        nexts.append(next)
    result = np.concatenate(nexts, axis=0)
    return result


def _get_rgb_cat(xs, colours):
    """
    Get colour categories given RGB values. This is done by choosing
    the colour in `colours` that is the closest (in RGB space) to
    each point in the image `xs`. This function is a little memory
    intensive, and so the size of `xs` should not be too large.

    Args:
      xs: float numpy array of RGB images in [B, C, H, W] format
      colours: numpy array of colour categories and their RGB values
    Returns:
      result: int numpy array of shape [B, 1, H, W]
    """
    num_colours = np.shape(colours)[0]
    xs = np.expand_dims(xs, 0)
    cs = np.reshape(colours, [num_colours, 1, 3, 1, 1])
    dists = np.linalg.norm(xs - cs, axis=2)  # 2 = colour axis
    cat = np.argmin(dists, axis=0)
    cat = np.expand_dims(cat, axis=1)
    return cat


def get_cat_rgb(cats, colours):
    """
    Get RGB colours given the colour categories

    Args:
      cats: integer numpy array of colour categories
      colours: numpy array of colour categories and their RGB values
    Returns:
      numpy tensor of RGB colours
    """
    return colours[cats]
```

```
def process(xs, ys, max_pixel=256.0, downsize_input=False):
    """
    Pre-process CIFAR10 images by taking only the horse category,
    shuffling, and have colour values be bound between 0 and 1

    Args:
      xs: the colour RGB pixel values
      ys: the category labels
      max_pixel: maximum pixel value in the original data
    Returns:
      xs: value normalized and shuffled colour images
      grey: greyscale images, also normalized so values are between 0 and 1
    """
    xs = xs / max_pixel
    xs = xs[np.where(ys == HORSE_CATEGORY)[0], :, :, :]
    npr.shuffle(xs)

    grey = np.mean(xs, axis=1, keepdims=True)

    if downsize_input:
        downsize_module = nn.Sequential(
            nn.AvgPool2d(2),
            nn.AvgPool2d(2),
            nn.Upsample(scale_factor=2),
            nn.Upsample(scale_factor=2),
        )
        xs_downsized = downsize_module.forward(torch.from_numpy(xs).float())
        xs_downsized = xs_downsized.data.numpy()
        return (xs, xs_downsized)
    else:
        return (xs, grey)


def get_batch(x, y, batch_size):
    """
    Generated that yields batches of data

    Args:
      x: input values
      y: output values
      batch_size: size of each batch
    Yields:
      batch_x: a batch of inputs of size at most batch_size
      batch_y: a batch of outputs of size at most batch_size
    """
    N = np.shape(x)[0]
    assert N == np.shape(y)[0]
    for i in range(0, N, batch_size):
        batch_x = x[i : i + batch_size, :, :, :]
        batch_y = y[i : i + batch_size, :, :, :]
        yield (batch_x, batch_y)
```

Torch helper

```
def get_torch_vars(xs, ys, gpu=False):
    """
    Helper function to convert numpy arrays to pytorch tensors.
    If GPU is used, move the tensors to GPU.

    Args:
      xs (float numpy tenosor): greyscale input
      ys (int numpy tenosor): categorical labels
      gpu (bool): whether to move pytorch tensor to GPU
    Returns:
      Variable(xs), Variable(ys)
    """
    xs = torch.from_numpy(xs).float()
    ys = torch.from_numpy(ys).long()
    if gpu:
        xs = xs.cuda()
        ys = ys.cuda()
    return Variable(xs), Variable(ys)


def compute_loss(criterion, outputs, labels, batch_size, num_colours):
    """
    Helper function to compute the loss. Since this is a pixelwise
    prediction task we need to reshape the output and ground truth
    tensors into a 2D tensor before passing it in to the loss criteron.

    Args:
      criterion: pytorch loss criterion
      outputs (pytorch tensor): predicted labels from the model
      labels (pytorch tensor): ground truth labels
      batch_size (int): batch size used for training
      num_colours (int): number of colour categories
    Returns:
      pytorch tensor for loss
    """

    loss_out = outputs.transpose(1, 3).contiguous().view([batch_size * 32 * 32, num_cc
    loss_lab = labels.transpose(1, 3).contiguous().view([batch_size * 32 * 32])
    return criterion(loss_out, loss_lab)


def run_validation_step(
    cnn,
    criterion,
    test_grey,
    test_rgb_cat,
    batch_size,
```

```
        colours,
        plotpath=None,
        visualize=True,
        downsize_input=False
    ):
        correct = 0.0
        total = 0.0
        losses = []
        num_colours = np.shape(colours)[0]
        for i, (xs, ys) in enumerate(get_batch(test_grey, test_rgb_cat, batch_size)):
            images, labels = get_torch_vars(xs, ys, args.gpu)
            outputs = cnn(images)

            val_loss = compute_loss(
                criterion, outputs, labels, batch_size=args.batch_size, num_colours=num_cc
            )
            losses.append(val_loss.data.item())

            _, predicted = torch.max(outputs.data, 1, keepdim=True)
            total += labels.size(0) * 32 * 32
            correct += (predicted == labels.data).sum()

        if plotpath:  # only plot if a path is provided
            plot(
                xs,
                ys,
                predicted.cpu().numpy(),
                colours,
                plotpath,
                visualize=visualize,
                compare_bilinear=downsize_input,
            )

        val_loss = np.mean(losses)
        val_acc = 100 * correct / total
        return val_loss, val_acc
```

Visualization

```
    def plot(input, gtlabel, output, colours, path, visualize, compare_bilinear=False):
        """
        Generate png plots of input, ground truth, and outputs

        Args:
          input: the greyscale input to the colourization CNN
          gtlabel: the grouth truth categories for each pixel
          output: the predicted categories for each pixel
          colours: numpy array of colour categories and their RGB values
          path: output path
          visualize: display the figures inline or save the figures in path
```

```
    """
    grey = np.transpose(input[:10, :, :, :], [0, 2, 3, 1])
    gtcolor = get_cat_rgb(gtlabel[:10, 0, :, :], colours)
    predcolor = get_cat_rgb(output[:10, 0, :, :], colours)

    img_stack = [np.hstack(np.tile(grey, [1, 1, 1, 3])), np.hstack(gtcolor), np.hstack

    if compare_bilinear:
        downsize_module = nn.Sequential(
            nn.AvgPool2d(2),
            nn.AvgPool2d(2),
            nn.Upsample(scale_factor=2, mode="bilinear"),
            nn.Upsample(scale_factor=2, mode="bilinear"),
        )
        gt_input = np.transpose(
            gtcolor,
            [
                0,
                3,
                1,
                2
            ],
        )
        color_bilinear = downsize_module.forward(torch.from_numpy(gt_input).float())
        color_bilinear = np.transpose(color_bilinear.data.numpy(), [0, 2, 3, 1])
        img_stack = [
            np.hstack(np.transpose(input[:10, :, :, :], [0, 2, 3, 1])),
            np.hstack(gtcolor),
            np.hstack(predcolor),
            np.hstack(color_bilinear),
        ]
    img = np.vstack(img_stack)

    plt.grid(None)
    plt.imshow(img, vmin=0.0, vmax=1.0)
    if visualize:
        plt.show()
    else:
        plt.savefig(path)


def toimage(img, cmin, cmax):
    return Image.fromarray((img.clip(cmin, cmax) * 255).astype(np.uint8))


def plot_activation(args, cnn):
    # LOAD THE COLOURS CATEGORIES
    colours = np.load(args.colours, allow_pickle=True)[0]
    num_colours = np.shape(colours)[0]

    (x_train, y_train), (x_test, y_test) = load_cifar10()
    test_rgb, test_grey = process(x_test, y_test, downsize_input=args.downsize_input)
```

```
test_rgb_cat = get_rgb_cat(test_rgb, colours)

# Take the idnex of the test image
id = args.index
outdir = "outputs/" + args.experiment_name + "/act" + str(id)
if not os.path.exists(outdir):
    os.makedirs(outdir)
images, labels = get_torch_vars(
    np.expand_dims(test_grey[id], 0), np.expand_dims(test_rgb_cat[id], 0)
)
cnn.cpu()
outputs = cnn(images)
_, predicted = torch.max(outputs.data, 1, keepdim=True)
predcolor = get_cat_rgb(predicted.cpu().numpy()[0, 0, :, :], colours)
img = predcolor
toimage(predcolor, cmin=0, cmax=1).save(os.path.join(outdir, "output_%d.png" % id)

if not args.downsize_input:
    img = np.tile(np.transpose(test_grey[id], [1, 2, 0]), [1, 1, 3])
else:
    img = np.transpose(test_grey[id], [1, 2, 0])
toimage(img, cmin=0, cmax=1).save(os.path.join(outdir, "input_%d.png" % id))

img = np.transpose(test_rgb[id], [1, 2, 0])
toimage(img, cmin=0, cmax=1).save(os.path.join(outdir, "input_%d_gt.png" % id))

def add_border(img):
    return np.pad(img, 1, "constant", constant_values=1.0)

def draw_activations(path, activation, imgwidth=4):
    img = np.vstack(
        [
            np.hstack(
                [
                    add_border(filter)
                    for filter in activation[i * imgwidth : (i + 1) * imgwidth, :,
                ]
            )
            for i in range(activation.shape[0] // imgwidth)
        ]
    )
    scipy.misc.imsave(path, img)

for i, tensor in enumerate([cnn.out1, cnn.out2, cnn.out3, cnn.out4, cnn.out5]):
    draw_activations(
        os.path.join(outdir, "conv%d_out_%d.png" % (i + 1, id)), tensor.data.cpu(
    )
print("visualization results are saved to %s" % outdir)
```

Training

```
class AttrDict(dict):
    def __init__(self, *args, **kwargs):
        super(AttrDict, self).__init__(*args, **kwargs)
        self.__dict__ = self


def train(args, cnn=None):
    # Set the maximum number of threads to prevent crash in Teaching Labs
    # TODO: necessary?
    torch.set_num_threads(5)
    # Numpy random seed
    npr.seed(args.seed)

    # Save directory
    save_dir = "outputs/" + args.experiment_name

    # LOAD THE COLOURS CATEGORIES
    colours = np.load(args.colours, allow_pickle=True, encoding="bytes")[0]
    num_colours = np.shape(colours)[0]
    # INPUT CHANNEL
    num_in_channels = 1 if not args.downsize_input else 3
    # LOAD THE MODEL
    if cnn is None:
        Net = globals()[args.model]
        cnn = Net(args.kernel, args.num_filters, num_colours, num_in_channels)

    # LOSS FUNCTION
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(cnn.parameters(), lr=args.learn_rate)

    # DATA
    print("Loading data...")
    (x_train, y_train), (x_test, y_test) = load_cifar10()

    print("Transforming data...")
    train_rgb, train_grey = process(x_train, y_train, downsize_input=args.downsize_in
    train_rgb_cat = get_rgb_cat(train_rgb, colours)
    test_rgb, test_grey = process(x_test, y_test, downsize_input=args.downsize_input)
    test_rgb_cat = get_rgb_cat(test_rgb, colours)

    # Create the outputs folder if not created already
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    print("Beginning training ...")
    if args.gpu:
        cnn.cuda()
    start = time.time()

    train_losses = []
```

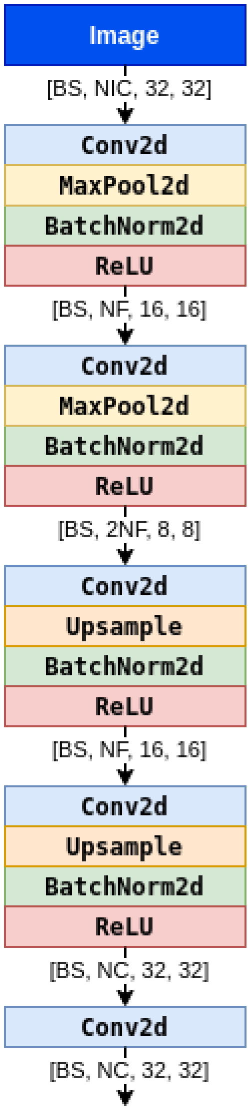
```
    valid_losses = []
    valid_accs = []
    for epoch in range(args.epochs):
        # Train the Model
        cnn.train()  # Change model to 'train' mode
        losses = []
        for i, (xs, ys) in enumerate(get_batch(train_grey, train_rgb_cat, args.batch_s
            images, labels = get_torch_vars(xs, ys, args.gpu)
            # Forward + Backward + Optimize
            optimizer.zero_grad()
            outputs = cnn(images)

            loss = compute_loss(
                criterion, outputs, labels, batch_size=args.batch_size, num_colours=nu
            )
            loss.backward()
            optimizer.step()
            losses.append(loss.data.item())

        # plot training images
        if args.plot:
            _, predicted = torch.max(outputs.data, 1, keepdim=True)
            plot(
                xs,
                ys,
                predicted.cpu().numpy(),
                colours,
                save_dir + "/train_%d.png" % epoch,
                args.visualize,
                args.downsize_input,
            )

        # plot training images
        avg_loss = np.mean(losses)
        train_losses.append(avg_loss)
        time_elapsed = time.time() - start
        print(
            "Epoch [%d/%d], Loss: %.4f, Time (s): %d"
            % (epoch + 1, args.epochs, avg_loss, time_elapsed)
        )

        # Evaluate the model
        cnn.eval()  # Change model to 'eval' mode (BN uses moving mean/var).
        val_loss, val_acc = run_validation_step(
            cnn,
            criterion,
            test_grey,
            test_rgb_cat,
            args.batch_size,
            colours,
            save_dir + "/test_%d.png" % epoch,
            args.visualize,
```

```
            args.downsize_input,
        )

        time_elapsed = time.time() - start
        valid_losses.append(val_loss)
        valid_accs.append(val_acc)
        print(
            "Epoch [%d/%d], Val Loss: %.4f, Val Acc: %.1f%%, Time(s): %.2f"
            % (epoch + 1, args.epochs, val_loss, val_acc, time_elapsed)
        )

    # Plot training curve
    plt.figure()
    plt.plot(train_losses, "ro-", label="Train")
    plt.plot(valid_losses, "go-", label="Validation")
    plt.legend()
    plt.title("Loss")
    plt.xlabel("Epochs")
    plt.savefig(save_dir + "/training_curve.png")

    if args.checkpoint:
        print("Saving model...")
        torch.save(cnn.state_dict(), args.checkpoint)

    return cnn
```

Part A: Pooling and Upsampling (2 pts)

Question 1

Complete the `PoolUpsampleNet` CNN model following the architecture described in the assignment handout.



In the diagram above, we denote the number of filters as **NF**. Further layers double the number of filters, denoted as **2NF**. In the final layers, the number of filters will be equivalent to the number of colour classes, denoted as **NC**. Consequently, your constructed neural network should define the number of input/output layers with respect to the variables `num_filters` and `num_colours`, as opposed to a constant value.

The specific modules to use are listed below. If parameters are not otherwise specified, use the default PyTorch parameters.

- [`nn.Conv2d`]() — The number of input filters should match the second dimension of the *input* tensor (e.g. the first `nn.Conv2d` layer has **NIC** input filters). The number of output filters should match the second dimension of the *output* tensor (e.g. the first `nn.Conv2d` layer has **NF** output filters). Set kernel size to parameter `kernel`. Set padding to the `padding` variable included in the starter code.
- [`nn.MaxPool2d`]() — Use `kernel_size=2` for all layers.

- [nn.BatchNorm2d]() — The number of features is specified after the hyphen in the diagram as a multiple of **NF** or **NC**.
- [nn.Upsample]() — Use `scaling_factor=2` for all layers.
- [nn.ReLU]()

We recommend grouping each block of operations (those adjacent without whitespace in the diagram) into [nn.Sequential]() containers. Grouping up relevant operations will allow for easier implementation of the `forward` method.

```
class PoolUpsampleNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        padding = kernel // 2

        ############### YOUR CODE GOES HERE ###############
        self.padding = padding
        self.kernel = kernel
        self.num_filters = num_filters
        self.num_colours = num_colours
        self.num_in_channels = num_in_channels

        self.block1 = nn.Sequential(
            nn.Conv2d(
                in_channels=self.num_in_channels,
                out_channels=self.num_filters,
                kernel_size=self.kernel,
                padding=self.padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_features=self.num_filters),
            nn.ReLU()
        )

        self.block2 = nn.Sequential(
            nn.Conv2d(
                in_channels=self.num_filters,
                out_channels=2*self.num_filters,
                kernel_size=self.kernel,
                padding=self.padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_features=2*self.num_filters),
            nn.ReLU()
        )

        self.block3 = nn.Sequential(
            nn.Conv2d(
                in_channels=2*self.num_filters,
```

```
                out_channels=self.num_filters,
                kernel_size=self.kernel,
                padding=self.padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=self.num_filters),
            nn.ReLU()
        )
        self.block4 = nn.Sequential(
            nn.Conv2d(
                in_channels=self.num_filters,
                out_channels=self.num_colours,
                kernel_size=self.kernel,
                padding=self.padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=self.num_colours),
            nn.ReLU()
        )

        self.block5 = nn.Conv2d(
            in_channels=self.num_colours,
            out_channels=self.num_colours,
            kernel_size=self.kernel,
            padding=self.padding
        )
        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        x = self.block5(x)
        return x
        ###################################################
```

Question 2

Run main training loop of `PoolUpsampleNet`. This will train the CNN for a few epochs using the cross-entropy objective. It will generate some images showing the trained result at the end. Do these results look good to you? Why or why not?

```
args = AttrDict()
args_dict = {
    "gpu": True,
    "valid": False,
    "checkpoint": "",
    "colours": "./data/colours/colour_kmeans24_cat7.npy",
    "model": "PoolUpsampleNet",
    "kernel": 3,
```

```
        "kernel": 3,
        "num_filters": 32,
        'learn_rate':0.001,
        "batch_size": 100,
        "epochs": 25,
        "seed": 0,
        "plot": True,
        "experiment_name": "colourization_cnn",
        "visualize": False,
        "downsize_input": False,
    }
    args.update(args_dict)
    cnn = train(args)
```

```
Loading data...
File path: data/cifar-10-batches-py.tar.gz
Transforming data...
Beginning training ...
Epoch [1/25], Loss: 2.4080, Time (s): 0
Epoch [1/25], Val Loss: 2.1114, Val Acc: 28.5%, Time(s): 1.05
Epoch [2/25], Loss: 1.9867, Time (s): 1
Epoch [2/25], Val Loss: 1.8911, Val Acc: 34.1%, Time(s): 2.01
Epoch [3/25], Loss: 1.8702, Time (s): 2
Epoch [3/25], Val Loss: 1.8064, Val Acc: 35.8%, Time(s): 3.00
Epoch [4/25], Loss: 1.8110, Time (s): 3
Epoch [4/25], Val Loss: 1.7633, Val Acc: 36.8%, Time(s): 4.03
Epoch [5/25], Loss: 1.7729, Time (s): 4
Epoch [5/25], Val Loss: 1.7315, Val Acc: 37.6%, Time(s): 5.09
Epoch [6/25], Loss: 1.7455, Time (s): 5
Epoch [6/25], Val Loss: 1.7067, Val Acc: 38.2%, Time(s): 6.19
Epoch [7/25], Loss: 1.7241, Time (s): 7
Epoch [7/25], Val Loss: 1.6906, Val Acc: 38.6%, Time(s): 7.33
Epoch [8/25], Loss: 1.7071, Time (s): 8
Epoch [8/25], Val Loss: 1.6779, Val Acc: 38.8%, Time(s): 8.49
Epoch [9/25], Loss: 1.6933, Time (s): 9
Epoch [9/25], Val Loss: 1.6675, Val Acc: 39.1%, Time(s): 9.68
Epoch [10/25], Loss: 1.6817, Time (s): 10
Epoch [10/25], Val Loss: 1.6578, Val Acc: 39.4%, Time(s): 10.90
Epoch [11/25], Loss: 1.6716, Time (s): 11
Epoch [11/25], Val Loss: 1.6503, Val Acc: 39.6%, Time(s): 12.16
Epoch [12/25], Loss: 1.6629, Time (s): 13
Epoch [12/25], Val Loss: 1.6436, Val Acc: 39.7%, Time(s): 13.45
Epoch [13/25], Loss: 1.6546, Time (s): 14
Epoch [13/25], Val Loss: 1.6394, Val Acc: 39.8%, Time(s): 14.77
Epoch [14/25], Loss: 1.6469, Time (s): 15
Epoch [14/25], Val Loss: 1.6349, Val Acc: 39.8%, Time(s): 16.14
Epoch [15/25], Loss: 1.6398, Time (s): 17
Epoch [15/25], Val Loss: 1.6261, Val Acc: 40.0%, Time(s): 17.54
Epoch [16/25], Loss: 1.6329, Time (s): 18
Epoch [16/25], Val Loss: 1.6176, Val Acc: 40.3%, Time(s): 18.99
```

Question 3

See assignment handout.

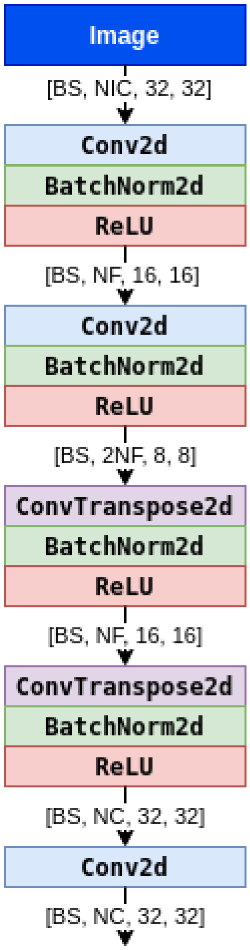
Part B: Strided and Transposed Convolutions (3 pts)

```
Epoch [22/25], Loss: 1.6012, Time (s): 27
```

Question 1

Complete the `ConvTransposeNet` CNN model following the architecture described in the assignment handout.

```
Epoch [25/25], Val Loss: 1.5869, Val Acc: 41.3%, Time(s): 33.22
```



An excellent visualization of convolutions and transposed convolutions with strides can be found here: https://github.com/vdumoulin/conv_arithmetic.

The specific modules to use are listed below. If parameters are not otherwise specified, use the default PyTorch parameters.

- `nn.Conv2d` — The number of input and output filters, and the kernel size, should be set in the same way as Part A. For the first two `nn.Conv2d` layers, set `stride` to 2 and set `padding` to 1.
- `nn.BatchNorm2d` — The number of features should be specified in the same way as for Part A.
- `nn.ConvTranspose2d` — The number of input filters should match the second dimension of the *input* tensor. The number of output filters should match the second dimension of the *output* tensor. Set `kernel_size` to parameter `kernel`. Set `stride` to 2, and set both `padding` and `output_padding` to 1.
- `nn.ReLU`

```
class ConvTransposeNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()
```

```
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        ############### YOUR CODE GOES HERE ###############
        self.padding = padding
        self.kernel = kernel
        self.num_filters = num_filters
        self.num_colours = num_colours
        self.num_in_channels = num_in_channels
        self.output_padding = output_padding
        self.stride = stride

        self.block1 = nn.Sequential(
            nn.Conv2d(
                in_channels =self.num_in_channels,
                out_channels =self.num_filters,
                stride = self.stride,
                kernel_size=self.kernel,
                padding= self.padding),
            nn.BatchNorm2d(num_features=self.num_filters),
            nn.ReLU()
        )

        self.block2 = nn.Sequential(
            nn.Conv2d(
                in_channels=self.num_filters,
                out_channels=2*self.num_filters,
                stride = self.stride,
                kernel_size=self.kernel,
                padding= self.padding),
            nn.BatchNorm2d(num_features=2*self.num_filters),
            nn.ReLU()
        )

        self.block3 = nn.Sequential(
            nn.ConvTranspose2d(
                in_channels=2*self.num_filters,
                out_channels=self.num_filters,
                stride = self.stride,
                kernel_size=self.kernel,
                padding= 1,
                output_padding = self.output_padding),
            nn.BatchNorm2d(num_features=self.num_filters),
            nn.ReLU()
        )

        self.block4 = nn.Sequential(
            nn.ConvTranspose2d(
```

```
                in_channels=self.num_filters,
                out_channels=self.num_colours,
                stride = self.stride,
                kernel_size=self.kernel,
                padding = 1,
                output_padding = self.output_padding),
            nn.BatchNorm2d(num_features=self.num_colours),
            nn.ReLU()
        )

        self.block5 = nn.Conv2d(
            in_channels=self.num_colours,
            out_channels =self.num_colours,
            kernel_size=self.kernel,
            padding = self.padding)

        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        x1 = self.block1(x)
        x2 = self.block2(x1)
        x3 = self.block3(x2)
        x4 = self.block4(x3)
        x5 = self.block5(x4)
        return x5
        ###################################################
```

Question 2

Train the model for at least 25 epochs using a batch size of 100 and a kernel size of 3. Plot the training curve, and include this plot in your write-up. How do the results compare to the previous model?

```
args = AttrDict()
args_dict = {
    "gpu": True,
    "valid": False,
    "checkpoint": "",
    "colours": "./data/colours/colour_kmeans24_cat7.npy",
    "model": "ConvTransposeNet",
    "kernel": 3,
    "num_filters": 32,
    'learn_rate':0.001,
    "batch_size": 100,
    "epochs": 25,
    "seed": 0,
    "plot": True,
    "experiment_name": "colourization_cnn",
```

```
        "visualize": False,
        "downsize_input": False,
    }
    args.update(args_dict)
    cnn = train(args)
```

```
Loading data...
File path: data/cifar-10-batches-py.tar.gz
Transforming data...
Beginning training ...
Epoch [1/25], Loss: 2.4785, Time (s): 1
Epoch [1/25], Val Loss: 2.0723, Val Acc: 31.0%, Time(s): 1.49
Epoch [2/25], Loss: 1.8689, Time (s): 2
Epoch [2/25], Val Loss: 1.7521, Val Acc: 37.8%, Time(s): 2.85
Epoch [3/25], Loss: 1.7075, Time (s): 4
Epoch [3/25], Val Loss: 1.6411, Val Acc: 40.3%, Time(s): 4.25
Epoch [4/25], Loss: 1.6247, Time (s): 5
Epoch [4/25], Val Loss: 1.5742, Val Acc: 41.8%, Time(s): 5.68
Epoch [5/25], Loss: 1.5636, Time (s): 6
Epoch [5/25], Val Loss: 1.5206, Val Acc: 43.2%, Time(s): 7.15
Epoch [6/25], Loss: 1.5151, Time (s): 8
Epoch [6/25], Val Loss: 1.4697, Val Acc: 44.9%, Time(s): 8.65
Epoch [7/25], Loss: 1.4757, Time (s): 9
Epoch [7/25], Val Loss: 1.4316, Val Acc: 46.1%, Time(s): 10.18
Epoch [8/25], Loss: 1.4429, Time (s): 11
Epoch [8/25], Val Loss: 1.4023, Val Acc: 46.9%, Time(s): 11.75
Epoch [9/25], Loss: 1.4146, Time (s): 13
Epoch [9/25], Val Loss: 1.3738, Val Acc: 47.8%, Time(s): 13.36
Epoch [10/25], Loss: 1.3894, Time (s): 14
Epoch [10/25], Val Loss: 1.3495, Val Acc: 48.5%, Time(s): 15.00
Epoch [11/25], Loss: 1.3666, Time (s): 16
Epoch [11/25], Val Loss: 1.3214, Val Acc: 49.6%, Time(s): 16.67
Epoch [12/25], Loss: 1.3457, Time (s): 18
Epoch [12/25], Val Loss: 1.2992, Val Acc: 50.3%, Time(s): 18.39
```

Questions 3 – 5

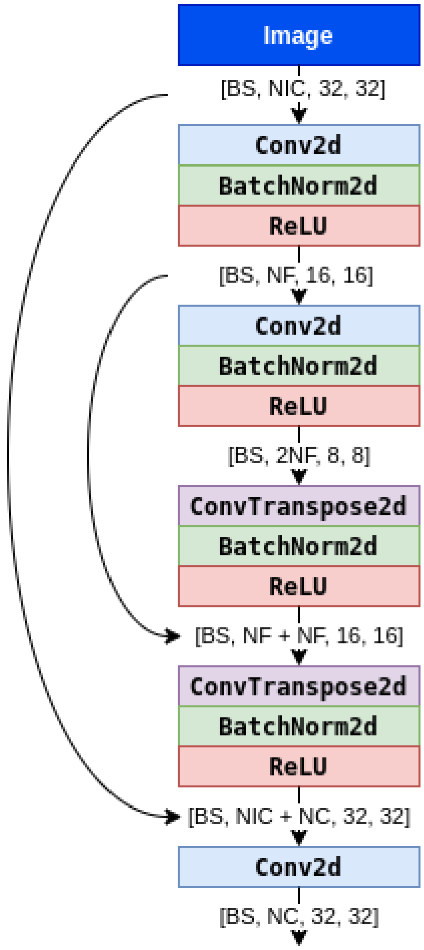
See assignment handout.

```
Epoch [15/25], Val Loss: 1.2444, Val Acc: 52.2%, Time(s): 23.71
```

Part C. Skip Connections (1 pts)

A skip connection in a neural network is a connection which skips one or more layer and connects to a later layer. We will introduce skip connections to our previous model.

Question 1



In this question, we will be adding a skip connection from the first layer to the last, second layer to the second last, etc. That is, the final convolution should have both the output of the previous layer and the initial greyscale input as input. This type of skip-connection is introduced by [Ronneberger et al.[2015]](), and is called a "UNet".

Just like the `ConvTransposeNet` class that you have completed in the previous part, complete the `__init__` and `forward` methods methods of the `UNet` class below.

Hint: You will need to use the function `torch.cat`.

```
class UNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        ############### YOUR CODE GOES HERE ###############
        self.padding = padding
        self.kernel = kernel
```

```
        self.num_filters = num_filters
        self.num_colours = num_colours
        self.num_in_channels = num_in_channels
        self.output_padding = output_padding
        self.stride = stride

        self.block1 = nn.Sequential(
            nn.Conv2d(
                in_channels =self.num_in_channels,
                out_channels =self.num_filters,
                stride = self.stride,
                kernel_size=self.kernel,
                padding= self.padding),
            nn.BatchNorm2d(num_features=self.num_filters),
            nn.ReLU()
        )

        self.block2 = nn.Sequential(
            nn.Conv2d(
                in_channels=self.num_filters,
                out_channels=2*self.num_filters,
                stride = self.stride,
                kernel_size=self.kernel,
                padding= self.padding),
            nn.BatchNorm2d(num_features=2*self.num_filters),
            nn.ReLU()
        )

        self.block3 = nn.Sequential(
            nn.ConvTranspose2d(
                in_channels=2*self.num_filters,
                out_channels=self.num_filters,
                stride = self.stride,
                kernel_size=self.kernel,
                padding= 1,
                output_padding = self.output_padding),
            nn.BatchNorm2d(num_features=self.num_filters),
            nn.ReLU()
        )

        self.block4 = nn.Sequential(
            nn.ConvTranspose2d(
                in_channels=2*self.num_filters,
                out_channels=self.num_colours,
                stride = self.stride,
                kernel_size=self.kernel,
                padding = 1,
                output_padding = self.output_padding),
            nn.BatchNorm2d(num_features=self.num_colours),
            nn.ReLU()
        )
```

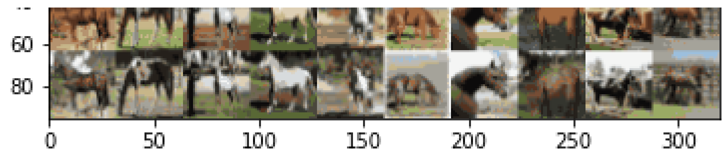
```
        self.block5 = nn.Conv2d(
            in_channels=self.num_colours+ self.num_in_channels,
            out_channels =self.num_colours,
            kernel_size=self.kernel,
            padding = self.padding)

        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        x1 = self.block1(x)
        x2 = self.block2(x1)
        x3 = self.block3(x2)

        x4_in = torch.cat((x1, x3), dim=1)
        x4_out = self.block4(x4_in)

        x5_in = torch.cat((x, x4_out), dim=1)
        x5_out = self.block5(x5_in)
        return x5_out
        ###################################################
```

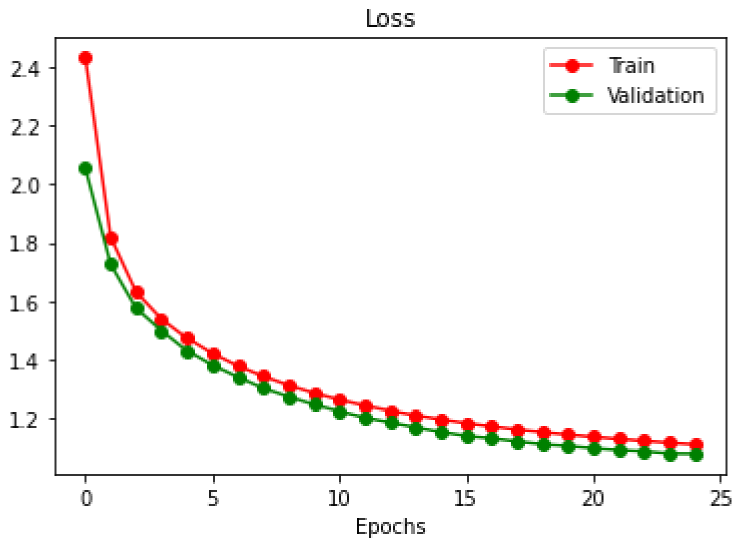
Question 2

Train the model for at least 25 epochs using a batch size of 100 and a kernel size of 3. Plot the training curve, and include this plot in your write-up.

```
args = AttrDict()
args_dict = {
    "gpu": True,
    "valid": False,
    "checkpoint": "",
    "colours": "./data/colours/colour_kmeans24_cat7.npy",
    "model": "UNet",
    "kernel": 3,
    "num_filters": 32,
    'learn_rate':0.001,
    "batch_size": 100,
    "epochs": 25,
    "seed": 0,
    "plot": True,
    "experiment_name": "colourization_cnn",
    "visualize": False,
    "downsize_input": False,
}
args.update(args_dict)
cnn = train(args)
```

```
Loading data...
File path: data/cifar-10-batches-py.tar.gz
Transforming data...
Beginning training ...
Epoch [1/25], Loss: 2.4335, Time (s): 1
Epoch [1/25], Val Loss: 2.0563, Val Acc: 31.9%, Time(s): 1.51
Epoch [2/25], Loss: 1.8192, Time (s): 2
Epoch [2/25], Val Loss: 1.7261, Val Acc: 37.1%, Time(s): 2.90
Epoch [3/25], Loss: 1.6317, Time (s): 4
Epoch [3/25], Val Loss: 1.5759, Val Acc: 41.9%, Time(s): 4.34
Epoch [4/25], Loss: 1.5392, Time (s): 5
Epoch [4/25], Val Loss: 1.4994, Val Acc: 44.1%, Time(s): 5.80
Epoch [5/25], Loss: 1.4734, Time (s): 7
Epoch [5/25], Val Loss: 1.4301, Val Acc: 46.5%, Time(s): 7.31
Epoch [6/25], Loss: 1.4207, Time (s): 8
Epoch [6/25], Val Loss: 1.3803, Val Acc: 48.1%, Time(s): 8.86
Epoch [7/25], Loss: 1.3779, Time (s): 10
Epoch [7/25], Val Loss: 1.3396, Val Acc: 49.4%, Time(s): 10.44
Epoch [8/25], Loss: 1.3422, Time (s): 11
Epoch [8/25], Val Loss: 1.3014, Val Acc: 50.7%, Time(s): 12.04
Epoch [9/25], Loss: 1.3117, Time (s): 13
Epoch [9/25], Val Loss: 1.2728, Val Acc: 51.6%, Time(s): 13.68
Epoch [10/25], Loss: 1.2854, Time (s): 15
Epoch [10/25], Val Loss: 1.2460, Val Acc: 52.6%, Time(s): 15.35
Epoch [11/25], Loss: 1.2626, Time (s): 16
Epoch [11/25], Val Loss: 1.2234, Val Acc: 53.3%, Time(s): 17.06
Epoch [12/25], Loss: 1.2427, Time (s): 18
Epoch [12/25], Val Loss: 1.2019, Val Acc: 53.9%, Time(s): 18.79
Epoch [13/25], Loss: 1.2252, Time (s): 20
Epoch [13/25], Val Loss: 1.1857, Val Acc: 54.5%, Time(s): 20.56
Epoch [14/25], Loss: 1.2097, Time (s): 21
Epoch [14/25], Val Loss: 1.1689, Val Acc: 55.0%, Time(s): 22.36
Epoch [15/25], Loss: 1.1958, Time (s): 23
Epoch [15/25], Val Loss: 1.1532, Val Acc: 55.5%, Time(s): 24.18
Epoch [16/25], Loss: 1.1834, Time (s): 25
Epoch [16/25], Val Loss: 1.1397, Val Acc: 55.9%, Time(s): 26.06
Epoch [17/25], Loss: 1.1721, Time (s): 27
Epoch [17/25], Val Loss: 1.1316, Val Acc: 56.1%, Time(s): 27.96
Epoch [18/25], Loss: 1.1620, Time (s): 29
Epoch [18/25], Val Loss: 1.1213, Val Acc: 56.4%, Time(s): 29.93
Epoch [19/25], Loss: 1.1527, Time (s): 31
Epoch [19/25], Val Loss: 1.1113, Val Acc: 56.7%, Time(s): 31.90
Epoch [20/25], Loss: 1.1443, Time (s): 33
Epoch [20/25], Val Loss: 1.1056, Val Acc: 56.8%, Time(s): 33.89
Epoch [21/25], Loss: 1.1365, Time (s): 35
Epoch [21/25], Val Loss: 1.0978, Val Acc: 56.9%, Time(s): 35.90
Epoch [22/25], Loss: 1.1293, Time (s): 37
Epoch [22/25], Val Loss: 1.0913, Val Acc: 57.1%, Time(s): 37.95
Epoch [23/25], Loss: 1.1226, Time (s): 39
Epoch [23/25], Val Loss: 1.0870, Val Acc: 57.2%, Time(s): 40.03
Epoch [24/25], Loss: 1.1165, Time (s): 41
Epoch [24/25], Val Loss: 1.0791, Val Acc: 57.4%, Time(s): 42.16
Epoch [25/25], Loss: 1.1107, Time (s): 43
Epoch [25/25], Val Loss: 1.0785, Val Acc: 57.3%, Time(s): 44.35
```





Question 3

See assignment handout.

Image Segmentation as Classification

In the previous two parts, we worked on training models for image colourization. Now we will switch gears and perform semantic segmentation by fine-tuning a pre-trained model.

Semantic segmentation can be considered as a pixel-wise classification problem where we need to predict the class label for each pixel. Fine-tuning is often used when you only have limited labeled data.

Here, we take a pre-trained model on the Microsoft COCO dataset and fine-tune it to perform segmentation with the classes it was never trained on. To be more specific, we use **deeplabv3** pre-trained model and fine-tune it on the Oxford17 flower dataset.

We simplify the task to be a binary semantic segmentation task (background and flower). In the following code, you will first see some examples from the Oxford17 dataset and load the finetune the model by truncating the last layer of the network and replacing it with a randomly initialized convolutional layer. Note that we only update the weights of the newly introduced layer.

Helper code

Below helper functions are provided for setting up the dataset and visualization.

```
import time

import cv2
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from PIL import Image
from scipy.io import loadmat
from torch.utils.data import DataLoader, Dataset
```

Data related code

```
# Dataset helper function
def read_image(path):
    im = cv2.imread(str(path))
    return cv2.cvtColor(im, cv2.COLOR_BGR2RGB)


def normalize(im):
    """Normalizes images with Imagenet stats."""
    imagenet_stats = np.array([[0.485, 0.456, 0.406], [0.229, 0.224, 0.225]])
    return (im / 255.0 - imagenet_stats[0]) / imagenet_stats[1]


def denormalize(img):
    imagenet_stats = np.array([[0.485, 0.456, 0.406], [0.229, 0.224, 0.225]])
    return img * imagenet_stats[1] + imagenet_stats[0]


# Mainly imported from https://colab.research.google.com/drive/1KzGRSNQpP4BonRKj3ZwGM
class CUB(Dataset):
    def __init__(self, files_path, split, train=True):

        self.files_path = files_path
        self.split = split
        if train:
            filenames = (
                list(self.split["trn1"][0])
                + list(self.split["trn2"][0])
                + list(self.split["trn3"][0])
            )
        else:
            # We only use `val1` for validation
            filenames = self.split["val1"][0]

        valid_filenames = []
        for i in filenames:
```

```
            img_name = "image_%04d.jpg" % int(i)
            if os.path.exists(os.path.join(files_path, "jpg", img_name)) and os.path.e
                os.path.join(files_path, "trimaps", img_name.replace("jpg", "png"))
            ):
                valid_filenames.append(img_name)

        self.valid_filenames = valid_filenames
        self.num_files = len(valid_filenames)

    def __len__(self):
        return self.num_files

    def __getitem__(self, index):

        filename = self.valid_filenames[index]

        # Load the image
        path = os.path.join(self.files_path, "jpg", filename)
        x = read_image(path)  # H*W*c
        x = cv2.resize(x, (224, 224))
        x = normalize(x)
        x = np.rollaxis(x, 2)  # To meet torch's input specification(c*H*W)

        # Load the segmentation mask
        path = os.path.join(self.files_path, "trimaps", filename.replace("jpg", "png")
        y = read_image(path)
        y = cv2.resize(y, (224, 224))  # H*W*c

        return x, y


def initialize_loader(train_batch_size=64, val_batch_size=64):
    split = loadmat("datasplits.mat")
    train_dataset = CUB("./", split, train=True)
    valid_dataset = CUB("./", split, train=False)
    train_loader = DataLoader(
        train_dataset, batch_size=train_batch_size, shuffle=True, num_workers=4, drop_
    )
    valid_loader = DataLoader(valid_dataset, batch_size=val_batch_size, num_workers=4)
    return train_loader, valid_loader
```

Visualization

```
def visualize_dataset(dataloader):
    """Imshow for Tensor."""
    x, y = next(iter(dataloader))

    fig = plt.figure(figsize=(10, 5))
    for i in range(4):
        inp = x[i]
```

```
        inp = x[i]
        inp = inp.numpy().transpose(1, 2, 0)
        inp = denormalize(inp)
        mask = y[i] / 255.0

        ax = fig.add_subplot(2, 2, i + 1, xticks=[], yticks=[])
        plt.imshow(np.concatenate([inp, mask], axis=1))


def plot_prediction(args, model, is_train, index_list=[0], plotpath=None, title=None):

    train_loader, valid_loader = initialize_loader()
    loader = train_loader if is_train else valid_loader

    images, masks = next(iter(loader))
    images = images.float()
    if args.gpu:
        images = images.cuda()

    with torch.no_grad():
        outputs = model(images)["out"]
    output_predictions = outputs.argmax(1)

    # create a color pallette, selecting a color for each class
    palette = torch.tensor([2 ** 25 - 1, 2 ** 15 - 1, 2 ** 21 - 1])
    colors = torch.as_tensor([i for i in range(21)])[:, None] * palette
    colors = (colors % 255).numpy().astype("uint8")
    colors = [i for color in colors for i in color]

    for index in index_list:

        r = Image.fromarray(output_predictions[index].byte().cpu().numpy())
        r.putpalette(colors)

        fig = plt.figure(figsize=(10, 5))
        if title:
            plt.title(title)

        ax = fig.add_subplot(1, 3, 1, xticks=[], yticks=[])
        plt.imshow(denormalize(images[index].cpu().numpy().transpose(1, 2, 0)))

        ax = fig.add_subplot(1, 3, 2, xticks=[], yticks=[])
        plt.imshow(r)

        ax = fig.add_subplot(1, 3, 3, xticks=[], yticks=[])
        plt.imshow(masks[index])

        if plotpath:
            plt.savefig(plotpath)
            plt.close()
```

Download dataset and initialize DataLoader

Download the [Oxford17 Flower] by running the code below. It will takes around 1 minutes for the first time.

```
import os

if not os.path.exists("17flowers.tgz"):
    print("Downloading flower dataset")
    !wget https://www.robots.ox.ac.uk/~vgg/data/flowers/17/17flowers.tgz
    !tar xvzf 17flowers.tgz
if not os.path.exists("trimaps.tgz"):
    !wget https://www.robots.ox.ac.uk/~vgg/data/flowers/17/trimaps.tgz
    !tar xvzf trimaps.tgz
if not os.path.exists("datasplits.mat"):
    !wget https://www.robots.ox.ac.uk/~vgg/data/flowers/17/datasplits.mat
```

```
trimaps/image_0722.png
trimaps/image_0723.png
trimaps/image_0724.png
trimaps/image_0725.png
trimaps/image_0726.png
trimaps/image_0727.png
trimaps/image_0728.png
trimaps/image_0729.png
trimaps/image_0730.png
trimaps/image_0731.png
trimaps/image_0732.png
trimaps/image_0733.png
trimaps/image_0734.png
trimaps/image_0735.png
trimaps/image_0736.png
trimaps/image_0737.png
trimaps/image_0738.png
trimaps/image_0739.png
trimaps/image_0740.png
trimaps/image_0741.png
trimaps/image_0743.png
trimaps/image_0744.png
trimaps/image_0745.png
trimaps/image_0746.png
trimaps/image_0747.png
trimaps/image_0748.png
trimaps/image_0749.png
trimaps/image_0750.png
trimaps/image_0751.png
trimaps/image_0752.png
trimaps/image_0753.png
trimaps/image_0754.png
trimaps/image_0755.png
trimaps/image_0756.png
trimaps/image_0757.png
trimaps/image_0758.png
trimaps/image_0759.png
```

```
trimaps/image_0761.png
trimaps/image_0762.png
trimaps/image_0763.png
trimaps/image_0764.png
trimaps/image_0766.png

trimaps/image_0767.png
trimaps/image_0768.png
trimaps/image_0769.png
trimaps/image_0770.png
trimaps/image_0771.png
trimaps/image_0772.png
trimaps/image_0773.png
trimaps/image_0774.png
trimaps/image_0775.png
trimaps/image_0776.png
trimaps/image_0777.png
trimaps/image_0778.png
trimaps/image_0779.png
trimaps/image_0780.png
trimaps/image_0781.png
trimaps/image_0782.png
trimaps/image_0783.png
trimaps/image_0784.png
```

Run the code below to initialize `DataLoader` and visualize few examples

```
train_loader, valid_loader = initialize_loader()
visualize_dataset(train_loader)
```

Load pre-trained model

Pytorch [Hub](#) supports publishing pre-trained models(model definitions and pre-trained weights) to a github repository by adding a simple hubconf.py file. Run the code below to download [deeplabv3](#).

```
# For further details, please refer to: https://arxiv.org/pdf/1706.05587.pds
model = torch.hub.load("pytorch/vision:v0.5.0", "deeplabv3_resnet101", pretrained=True
print(model)
```

```
Downloading: "https://github.com/pytorch/vision/archive/v0.5.0.zip" to /root/.ca
Downloading: "https://download.pytorch.org/models/resnet101-5d3b4d8f.pth" to /ro
```

100% 170M/170M [00:02<00:00, 68.7MB/s]

```
Downloading: "https://download.pytorch.org/models/deeplabv3_resnet101_coco-586e9
```

100% 233M/233M [00:03<00:00, 80.2MB/s]

```
DeepLabV3(
  (backbone): IntermediateLayerGetter(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bi
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mo
    (layer1): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
        (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
        (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
        (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        (relu): ReLU(inplace=True)
      )
    )
    (layer2): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
```

```
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        (relu): ReLU(inplace=True)
      )
      (3): Bottleneck(
        (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        (relu): ReLU(inplace=True)
      )
    )
    (layer3): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
```

```
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (6): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (7): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (8): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
```

```
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (9): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (10): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (11): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (12): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (13): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (14): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
      (relu): ReLU(inplace=True)
    )
    (15): Bottleneck(
```

```
        (15): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
          (relu): ReLU(inplace=True)
        )
        (16): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
          (relu): ReLU(inplace=True)
        )
        (17): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
          (relu): ReLU(inplace=True)
        )
        (18): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
          (relu): ReLU(inplace=True)
        )
        (19): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
          (relu): ReLU(inplace=True)
        )
        (20): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
          (relu): ReLU(inplace=True)
        )
        (21): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
          (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
```

```
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
        (relu): ReLU(inplace=True)
      )
      (22): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2,
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
        (relu): ReLU(inplace=True)
      )
    )
    (layer4): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(2,
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_run
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_run
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(4,
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_run
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(4,
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_run
        (relu): ReLU(inplace=True)
      )
```

Helper functions for training

Below are few functions helpful for model training.

```
        (0): Sequential(
```

```
def compute_loss(pred, gt):
    loss = F.cross_entropy(pred, gt)
    return loss
```

```
# from https://www.kaggle.com/iezepov/fast-iou-scoring-metric-in-pytorch-and-numpy
def iou_pytorch(outputs, labels):

    SMOOTH = 1e-6
    # You can comment out this line if you are passing tensors of equal shape
    # But if you are passing output from UNet or something it will most probably
    # be with the BATCH x 1 x H x W shape
    outputs = torch.argmax(outputs, 1)
    outputs = outputs.squeeze(1)  # BATCH x 1 x H x W => BATCH x H x W

    intersection = (outputs & labels).float().sum((1, 2))  # Will be zero if Truth=0 o
    union = (outputs | labels).float().sum((1, 2))  # Will be zero if both are 0

    iou = (intersection + SMOOTH) / (union + SMOOTH)  # We smooth our devision to avoi

    thresholded = (
        torch.clamp(20 * (iou - 0.5), 0, 10).ceil() / 10
    )  # This is equal to comparing with thresolds

    return (
        thresholded.mean()
    )  # Or thresholded.mean() if you are interested in average across the batch


def convert_to_binary(masks, thres=0.5):
    binary_masks = (
        (masks[:, 0, :, :] == 128) & (masks[:, 1, :, :] == 0) & (masks[:, 2, :, :] ==
    ) + 0.0
    return binary_masks.long()

def run_validation_step(args, epoch, model, loader, plotpath=None):

    model.eval()  # Change model to 'eval' mode (BN uses moving mean/var).

    losses = []
    ious = []
    with torch.no_grad():
        for i, (images, masks) in enumerate(loader):
            permute_masks = masks.permute(0, 3, 1, 2)  # to match the input size: B, C
            binary_masks = convert_to_binary(permute_masks)
            if args.gpu:
                images = images.cuda()
                binary_masks = binary_masks.cuda()
            output = model(images.float())
            pred_seg_masks = output["out"]

            output_predictions = pred_seg_masks[0].argmax(0)
            if args.loss == 'cross-entropy':
                loss = compute_loss(pred_seg_masks, binary_masks)
            else:
```

```
            loss = compute_iou_loss(pred_seg_masks, binary_masks)
            iou = iou_pytorch(pred_seg_masks, binary_masks)
            losses.append(loss.data.item())
            ious.append(iou.data.item())

        val_loss = np.mean(losses)
        val_iou = np.mean(ious)

    if plotpath:
        plot_prediction(
            args, model, False, index_list=[0], plotpath=plotpath, title="Val_%d" % ep
        )

    return val_loss, val_iou
```

Part D.1. Finetune Semantic Segmentation Model with Cross Entropy Loss (2 pts)

Question 1.

For this assignment, we want to fine-tune only the last layer in our downloaded deeplabv3. We do this by keeping track of weights we want to update in `learned_parameters`.

Use the PyTorch utility [`Model.named_parameters()`](), which returns an iterator over all the weight matrices of the model.

The last layer weights have names prefix `classifier.4`. We will select the corresponding weights then passing them to `learned_parameters`.

Complete the `train` function in Part D.1 of the notebook by adding 3 lines of code where indicated.

```
def train(args, model):

    # Set the maximum number of threads to prevent crash in Teaching Labs
    torch.set_num_threads(5)
    # Numpy random seed
    np.random.seed(args.seed)

    # Save directory
    # Create the outputs folder if not created already
    save_dir = "outputs/" + args.experiment_name
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    learned_parameters = []
    # We only learn the last layer and freeze all the other weights
    ################ Code goes here ######################
```

```
    # Around 3 lines of code
    # Hint:
    # - use a for loop to loop over all model.named_parameters()
    # - append the parameters (both weights and biases) of the last layer (prefix: cla
    for name, param in model.named_parameters():
        if name.startswith("classifier.4"):
            learned_parameters.append(param)
    ######################################################


    # Adam only updates learned_parameters
    optimizer = torch.optim.Adam(learned_parameters, lr=args.learn_rate)

    train_loader, valid_loader = initialize_loader(args.train_batch_size, args.val_bat
    print(
        "Train set: {}, Test set: {}".format(
            train_loader.dataset.num_files, valid_loader.dataset.num_files
        )
    )

    print("Beginning training ...")
    if args.gpu:
        model.cuda()

    start = time.time()
    trn_losses = []
    val_losses = []
    val_ious = []
    best_iou = 0

    for epoch in range(args.epochs):

        # Train the Model
        model.train()  # Change model to 'train' mode
        start_tr = time.time()

        losses = []
        for i, (images, masks) in enumerate(train_loader):
            permute_masks = masks.permute(0, 3, 1, 2)  # to match the input size: B, C
            binary_masks = convert_to_binary(permute_masks)  # B, H, W
            if args.gpu:
                images = images.cuda()
                binary_masks = binary_masks.cuda()

            # Forward + Backward + Optimize
            optimizer.zero_grad()
            output = model(images.float())
            pred_seg_masks = output["out"]

            _, pred_labels = torch.max(pred_seg_masks, 1, keepdim=True)
            if args.loss == 'cross-entropy':
                loss = compute_loss(pred_seg_masks, binary_masks)
```

```
        else:
            loss = compute_iou_loss(pred_seg_masks, binary_masks)
        loss.backward()
        optimizer.step()
        losses.append(loss.data.item())

    # plot training images
    if args.plot:
        plot_prediction(
            args,
            model,
            True,
            index_list=[0],
            plotpath=save_dir + "/train_%d.png" % epoch,
            title="Train_%d" % epoch,
        )

    # plot training images
    trn_loss = np.mean(losses)
    trn_losses.append(trn_loss)
    time_elapsed = time.time() - start_tr
    print(
        "Epoch [%d/%d], Loss: %.4f, Time (s): %d"
        % (epoch + 1, args.epochs, trn_loss, time_elapsed)
    )

    # Evaluate the model
    start_val = time.time()
    val_loss, val_iou = run_validation_step(
        args, epoch, model, valid_loader, save_dir + "/val_%d.png" % epoch
    )

    if val_iou > best_iou:
        best_iou = val_iou
        torch.save(
            model.state_dict(), os.path.join(save_dir, args.checkpoint_name + "-be
        )

    time_elapsed = time.time() - start_val
    print(
        "Epoch [%d/%d], Loss: %.4f, mIOU: %.4f, Validation time (s): %d"
        % (epoch + 1, args.epochs, val_loss, val_iou, time_elapsed)
    )

    val_losses.append(val_loss)
    val_ious.append(val_iou)

# Plot training curve
plt.figure()
plt.plot(trn_losses, "ro-", label="Train")
plt.plot(val_losses, "go-", label="Validation")
```

```
    plt.legend()
    plt.title("Loss")
    plt.xlabel("Epochs")
    plt.savefig(save_dir + "/training_curve.png")

    # Plot validation iou curve
    plt.figure()
    plt.plot(val_ious, "ro-", label="mIOU")
    plt.legend()
    plt.title("mIOU")
    plt.xlabel("Epochs")
    plt.savefig(save_dir + "/val_iou_curve.png")

    print("Saving model...")
    torch.save(
        model.state_dict(),
        os.path.join(save_dir, args.checkpoint_name + "-{}-last.ckpt".format(args.epoc
    )

    print("Best model achieves mIOU: %.4f" % best_iou)
```

Question 2.

For fine-tuning we also want to

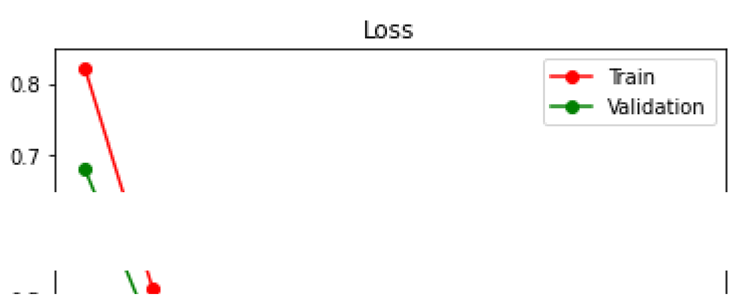
- use `Model.requires_grad_()` to prevent back-prop through all the layers that should be frozen
- replace the last layer with a new `nn.Conv2d` with appropriate input output channels and kernel sizes. Since we are performing binary segmentation for this assignment, this new layer should have 2 output channels.

Complete the script below by adding around 4 lines of code and train the model.

```
class AttrDict(dict):
    def __init__(self, *args, **kwargs):
        super(AttrDict, self).__init__(*args, **kwargs)
        self.__dict__ = self

args = AttrDict()
# You can play with the hyperparameters here, but to finish the assignment,
# there is no need to tune the hyperparameters here.
args_dict = {
    "gpu": True,
    "checkpoint_name": "finetune-segmentation",
    "learn_rate": 0.05,
    "train_batch_size": 128,
    "val_batch_size": 256,
    "epochs": 10,
    "loss": 'cross-entropy',
```

```
    "seed": 0,
    "plot": True,
    "experiment_name": "finetune-segmentation",
}
args.update(args_dict)

# Truncate the last layer and replace it with the new one.
# To avoid `CUDA out of memory` error, you might find it useful (sometimes required)
#   to set the `requires_grad`=False for some layers
################ YOUR CODE GOES HERE ######################
# Around 4 lines of code
# Hint:
# - replace the classifier.4 layer with the new Conv2d layer (1 line)
# - no need to consider the aux_classifier module (just treat it as don't care)
# - freeze the gradient of other layers (3 lines)
model.classifier[4] = nn.Conv2d(256, 2, kernel_size=(1, 1), stride=(1, 1))

for name, param in model.named_parameters():
    if not name.startswith("classifier.4"):
        param.requires_grad = False
######################################################


# Clear the cache in GPU
torch.cuda.empty_cache()
train(args, model)
```

```
Train set: 1280, Test set: 212
Beginning training ...
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [1/10], Loss: 0.8203, Time (s): 44
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [1/10], Loss: 0.6803, mIOU: 0.0802, Validation time (s): 12
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [2/10], Loss: 0.5141, Time (s): 47
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [2/10], Loss: 0.4502, mIOU: 0.1887, Validation time (s): 13
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [3/10], Loss: 0.3883, Time (s): 47
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [3/10], Loss: 0.3215, mIOU: 0.2439, Validation time (s): 13
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [4/10], Loss: 0.3344, Time (s): 47
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [4/10], Loss: 0.3114, mIOU: 0.2406, Validation time (s): 12
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [5/10], Loss: 0.3112, Time (s): 48
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [5/10], Loss: 0.2948, mIOU: 0.3033, Validation time (s): 13
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [6/10], Loss: 0.3037, Time (s): 47
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [6/10], Loss: 0.2816, mIOU: 0.3146, Validation time (s): 13
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [7/10], Loss: 0.2977, Time (s): 47
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [7/10], Loss: 0.2813, mIOU: 0.2967, Validation time (s): 12
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [8/10], Loss: 0.2918, Time (s): 46
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [8/10], Loss: 0.2756, mIOU: 0.3425, Validation time (s): 13
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [9/10], Loss: 0.2886, Time (s): 46
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [9/10], Loss: 0.2692, mIOU: 0.3255, Validation time (s): 12
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [10/10], Loss: 0.2837, Time (s): 48
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [10/10], Loss: 0.2705, mIOU: 0.3127, Validation time (s): 12
Saving model...
Best model achieves mIOU: 0.3425
```



Question 3.

Visualize predictions by running the code below.

```
plot_prediction(args, model, is_train=True, index_list=[0, 1, 2, 3])
```

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


```
plot_prediction(args, model, is_train=False, index_list=[0, 1, 2, 3])
```

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

Part D.2. Finetune Semantic Segmentation Model with IoU Loss (2 pts)

Question 1.

We will change the loss function from cross entropy used in part D.1 to the (soft) IoU loss.

Complete the `compute_iou_loss` function below by adding around 5 lines of code each where indicated.

```
def compute_iou_loss(pred, gt, SMOOTH=1e-6):
    # Compute the IoU between the pred and the gt (ground truth)
    ################ YOUR CODE GOES HERE ######################
    # Around 5 lines of code
    # Hint:
    # - apply softmax on pred along the channel dimension (dim=1)
    # - only have to compute IoU between gt and the foreground channel of pred
    # - no need to consider IoU for the background channel of pred
    # - extract foreground from the softmaxed pred (e.g., softmaxed_pred[:, 1, :, :])
    # - compute intersection between foreground and gt
    # - compute union between foreground and gt
    # - compute loss using the computed intersection and union
    ######################################################
    s = nn.Softmax(dim=1)
    softmaxed_pred = s(pred)
    foreground = softmaxed_pred[:, 1, :, :]
    intersection = foreground.mul(gt)
    union = (foreground + gt - intersection).sum()
    loss = 1- intersection.sum() / union

    return loss
```

Same as D.1, complete the script below by adding around 4 lines of code and train the model.

```
args = AttrDict()
# You can play with the hyperparameters here, but to finish the assignment,
# there is no need to tune the hyperparameters here.
args_dict = {
    "gpu": True,
    "checkpoint_name": "finetune-segmentation",
    "learn_rate": 0.05,
    "train_batch_size": 128,
    "val_batch_size": 256,
    "epochs": 10,
    "loss": 'iou',
    "seed": 0,
    "plot": True,
    "experiment_name": "finetune-segmentation",
}
args.update(args_dict)

# Truncate the last layer and replace it with the new one.
# To avoid `CUDA out of memory` error, you might find it useful (sometimes required)
#   to set the `requires_grad`=False for some layers
################ YOUR CODE GOES HERE ######################
# Around 4 lines of code
# Hint:
# - replace the classifier.4 layer with the new Conv2d layer (1 line)
```

```
# - no need to consider the aux_classifier module (just treat it as don't care)
# - freeze the gradient of other layers (3 lines)
model.classifier[4] = nn.Conv2d(256, 2, kernel_size=(1, 1), stride=(1, 1))

for name, param in model.named_parameters():
    if not name.startswith("classifier.4"):
        param.requires_grad = False
######################################################


# Clear the cache in GPU
torch.cuda.empty_cache()
train(args, model)
```

```
Train set: 1280, Test set: 212
Beginning training ...
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [1/10], Loss: 0.7019, Time (s): 49
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [1/10], Loss: 0.5850, mIOU: 0.1009, Validation time (s): 12
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [2/10], Loss: 0.5513, Time (s): 46
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [2/10], Loss: 0.5158, mIOU: 0.1778, Validation time (s): 12
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [3/10], Loss: 0.5096, Time (s): 47
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [3/10], Loss: 0.4912, mIOU: 0.2231, Validation time (s): 13
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [4/10], Loss: 0.4848, Time (s): 46
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [4/10], Loss: 0.4642, mIOU: 0.2241, Validation time (s): 12
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [5/10], Loss: 0.4628, Time (s): 48
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [5/10], Loss: 0.4437, mIOU: 0.2538, Validation time (s): 12
Epoch [6/10], Loss: 0.4428, Time (s): 46
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [6/10], Loss: 0.4323, mIOU: 0.2693, Validation time (s): 12
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [7/10], Loss: 0.4307, Time (s): 47
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [7/10], Loss: 0.4249, mIOU: 0.2825, Validation time (s): 13
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [8/10], Loss: 0.4202, Time (s): 46
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [8/10], Loss: 0.4147, mIOU: 0.2976, Validation time (s): 13
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Epoch [9/10], Loss: 0.4129, Time (s): 46
```

Question 2.

Visualize predictions by running the code below.

```
Epoch [10/10], Loss: 0.3965, mIOU: 0.3208, Validation time (s): 13
```

```
plot_prediction(args, model, is_train=True, index_list=[0, 1, 2, 3])
```

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

```
plot_prediction(args, model, is_train=False, index_list=[0, 1, 2, 3])
```

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