EECS 504 PS7: Image Translation

Please provide the following information (e.g. Andrew Owens, ahowens):

[Your first name] [Your last name], [Your UMich uniqname]

Starting

Run the following code to import the modules you'll need. After your finish the assignment, remember to run all ce save the note book to your local machine as a .ipynb file for Canvas submission.

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
import os
import time
import itertools

import torch
import torchvision
from torchvision import datasets, models, transforms
import torch.nn as nn
import torch.optim as optim
from torch.autograd import Variable
import torch.nn.functional as F

print("PyTorch Version: ",torch.__version__)
print("Torchvision Version: ",torchvision.__version__)
# Detect if we have a GPU available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
if torch.cuda.is_available():
    print("Using the GPU!")
else:
    print("WARNING: Could not find GPU! Using CPU only. If you want to enable GPU, please to go
```

Problem 7.1 pix2pix

You will build pix2pix for image translation. Here is the website of pix2pix: https://phillipi.github.io/pix2pix/

Read the paper and github repo to understand how it implement.

In this question, you will need to:

1. Contruct dataloaders for train/test datasets
2. Build Generator and Discriminator
3. Train pix2pix and visualize the result during training
4. Plot the loss of generator/discriminator v.s. interation

Step 0: Downloading the dataset.

```
# Download the CMP Facade Database
# Note: Restarting the runtime won't remove the downloaded dataset. You only need to re-download
!wget http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/facades.tar.gz

# Unzip the download dataset .zip file to your local colab dir
!tar -xf facades.tar.gz
```

Step 1: Build dataloaders for train and test

```
def load_data(path, subfolder, transform, batch_size, shuffle=True):
    """
    Data loader.

    Inputs:
    - path: path of the data.
    - subfolder: subfolder of the data.
    - transform: data transformation.
    - batch_size: the size of the batch
    - shuffle: if true, shuffle the data

    Outputs:
    - torch Dataloader
    """

    #++++++++++++++++++++++++++++++++++++++++++++++++#
    #++++++++++++++++++++++++++++++++++++++++++++++++#
    # ============== YOUR CODE HERE ============== #
    #Hint: Use torch.utils.data.DataLoader

    # delete start
    dset = datasets.ImageFolder(path, transform)
    ind = dset.class_to_idx[subfolder]
    n = 0
    for i in range(dset.__len__()):
        if ind != dset.imgs[n][1]:
            del dset.imgs[n]
            n -= 1

        n += 1
    return torch.utils.data.DataLoader(dset, batch_size=batch_size, shuffle=shuffle)
    # delete end
```

```
    # ============== END OF CODE ================= #
    #++++++++++++++++++++++++++++++++++++++++++++++#
    #++++++++++++++++++++++++++++++++++++++++++++++#


# data_loader
transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
])
train_loader = load_data('./facades', 'train', transform, 1, shuffle=True)
test_loader = load_data('./facades', 'val', transform, 10, shuffle=False)

#Sample Output used for visualization
test = test_loader.__iter__().__next__()[0]
img_size = test.size()[2]
fixed_y_ = test[:, :, :, 0:img_size]
fixed_x_ = test[:, :, :, img_size:]
print(len(train_loader))
print(len(test_loader))
```

```
# plot sample image
example = train_loader.__iter__().__next__()[0][0].numpy().transpose((1, 2, 0))
mean = np.array([0.5, 0.5, 0.5])
std = np.array([0.5, 0.5, 0.5])
example = std * example + mean
plt.imshow(example)
plt.show()
```

Step 2: Build Generator and Discriminator

Based on the paper, the architectures of network are as following:

Generator architectures:

U-net encoder:

C64-C128-C256-C512-C512-C512-C512-C512

U-net decoder:

CD512-CD1024-CD1024-C1024-C1024-C512-C256-C128

After the last layer in the decoder, a convolution is applied to map to the number of output channels, followed by a
function. As an exception to the above notation, BatchNorm is not applied to the first C64 layer in the encoder. All

in the encoder are leaky, with slope 0.2, while ReLUs in the decoder are not leaky.

Discriminator architectures

The 70 × 70 discriminator architecture is:

C64-C128-C256-C512

After the last layer, a convolution is applied to map to a 1-dimensional output, followed by a Sigmoid function. As a
exception to the above notation, BatchNorm is not applied to the first C64 layer. All ReLUs are leaky, with slope 0.2

```
def normal_init(m, mean, std):
    """
    Helper function. Initialize parameter with given mean and std.
    """
    if isinstance(m, nn.ConvTranspose2d) or isinstance(m, nn.Conv2d):
        #++++++++++++++++++++++++++++++++++++++++++++++#
        #++++++++++++++++++++++++++++++++++++++++++++++#
        # ============== YOUR CODE HERE ============== #
        # delete start
        m.weight.data.normal_(mean, std)
        m.bias.data.zero_()
        # delete end
        # ============== END OF CODE ================= #
        #++++++++++++++++++++++++++++++++++++++++++++++#
        #++++++++++++++++++++++++++++++++++++++++++++++#


class generator(nn.Module):
    # initializers
    def __init__(self):
        super(generator, self).__init__()
        #++++++++++++++++++++++++++++++++++++++++++++++#
        #++++++++++++++++++++++++++++++++++++++++++++++#
        # ============== YOUR CODE HERE ============== #
        # delete start
        # Unet encoder
        self.conv1 = nn.Conv2d(3, 64, 4, 2, 1)
        self.conv2 = nn.Conv2d(64, 64 * 2, 4, 2, 1)
        self.conv2_bn = nn.BatchNorm2d(64 * 2)
        self.conv3 = nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1)
        self.conv3_bn = nn.BatchNorm2d(64 * 4)
        self.conv4 = nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1)
        self.conv4_bn = nn.BatchNorm2d(64 * 8)
        self.conv5 = nn.Conv2d(64 * 8, 64 * 8, 4, 2, 1)
        self.conv5_bn = nn.BatchNorm2d(64 * 8)
        self.conv6 = nn.Conv2d(64 * 8, 64 * 8, 4, 2, 1)
        self.conv6_bn = nn.BatchNorm2d(64 * 8)
        self.conv7 = nn.Conv2d(64 * 8, 64 * 8, 4, 2, 1)
        self.conv7_bn = nn.BatchNorm2d(64 * 8)
        self.conv8 = nn.Conv2d(64 * 8, 64 * 8, 4, 2, 1)
        # self.conv8_bn = nn.BatchNorm2d(d * 8)

        # Unet decoder
        self.deconv1 = nn.ConvTranspose2d(64 * 8, 64 * 8, 4, 2, 1)
        self.deconv1_bn = nn.BatchNorm2d(64 * 8)
        self.deconv2 = nn.ConvTranspose2d(64 * 8 * 2, 64 * 8, 4, 2, 1)
        self.deconv2_bn = nn.BatchNorm2d(64 * 8)
        self.deconv3 = nn.ConvTranspose2d(64 * 8 * 2, 64 * 8, 4, 2, 1)
        self.deconv3_bn = nn.BatchNorm2d(64 * 8)
        self.deconv4 = nn.ConvTranspose2d(64 * 8 * 2, 64 * 8, 4, 2, 1)
```

```
        self.deconv4_bn = nn.BatchNorm2d(64 * 8)
        self.deconv5 = nn.ConvTranspose2d(64 * 8 * 2, 64 * 4, 4, 2, 1)
        self.deconv5_bn = nn.BatchNorm2d(64 * 4)
        self.deconv6 = nn.ConvTranspose2d(64 * 4 * 2, 64 * 2, 4, 2, 1)
        self.deconv6_bn = nn.BatchNorm2d(64 * 2)
        self.deconv7 = nn.ConvTranspose2d(64 * 2 * 2, 64, 4, 2, 1)
        self.deconv7_bn = nn.BatchNorm2d(64)
        self.deconv8 = nn.ConvTranspose2d(64 * 2, 3, 4, 2, 1)
        # delete end
        # ============== END OF CODE ================= #
        #++++++++++++++++++++++++++++++++++++++++++++++#
        #++++++++++++++++++++++++++++++++++++++++++++++#

    # weight_init
    def weight_init(self, mean, std):
        for m in self._modules:
            normal_init(self._modules[m], mean, std)

    # forward method
    def forward(self, input):
        #++++++++++++++++++++++++++++++++++++++++++++++#
        #++++++++++++++++++++++++++++++++++++++++++++++#
        # ============== YOUR CODE HERE ============== #
        # delete start
        e1 = self.conv1(input)
        e2 = self.conv2_bn(self.conv2(F.leaky_relu(e1, 0.2)))
        e3 = self.conv3_bn(self.conv3(F.leaky_relu(e2, 0.2)))
        e4 = self.conv4_bn(self.conv4(F.leaky_relu(e3, 0.2)))
        e5 = self.conv5_bn(self.conv5(F.leaky_relu(e4, 0.2)))
        e6 = self.conv6_bn(self.conv6(F.leaky_relu(e5, 0.2)))
        e7 = self.conv7_bn(self.conv7(F.leaky_relu(e6, 0.2)))
        e8 = self.conv8(F.leaky_relu(e7, 0.2))
        # e8 = self.conv8_bn(self.conv8(F.leaky_relu(e7, 0.2)))
        d1 = F.dropout(self.deconv1_bn(self.deconv1(F.relu(e8))), 0.5, training=True)
        d1 = torch.cat([d1, e7], 1)
        d2 = F.dropout(self.deconv2_bn(self.deconv2(F.relu(d1))), 0.5, training=True)
        d2 = torch.cat([d2, e6], 1)
        d3 = F.dropout(self.deconv3_bn(self.deconv3(F.relu(d2))), 0.5, training=True)
        d3 = torch.cat([d3, e5], 1)
        d4 = self.deconv4_bn(self.deconv4(F.relu(d3)))
        # d4 = F.dropout(self.deconv4_bn(self.deconv4(F.relu(d3))), 0.5)
        d4 = torch.cat([d4, e4], 1)
        d5 = self.deconv5_bn(self.deconv5(F.relu(d4)))
        d5 = torch.cat([d5, e3], 1)
        d6 = self.deconv6_bn(self.deconv6(F.relu(d5)))
        d6 = torch.cat([d6, e2], 1)
        d7 = self.deconv7_bn(self.deconv7(F.relu(d6)))
        d7 = torch.cat([d7, e1], 1)
        d8 = self.deconv8(F.relu(d7))
        o = torch.tanh(d8)
        # delete end
        # ============== END OF CODE ================= #
        #++++++++++++++++++++++++++++++++++++++++++++++#
        #++++++++++++++++++++++++++++++++++++++++++++++#

        return o

class discriminator(nn.Module):
    # initializers
    def __init__(self):
```

```
        super(discriminator, self).__init__()
        #++++++++++++++++++++++++++++++++++++++++++++++#
        #++++++++++++++++++++++++++++++++++++++++++++++#
        # ============== YOUR CODE HERE ============== #
        # delete start
        self.conv1 = nn.Conv2d(6, 64, 4, 2, 1)
        self.conv2 = nn.Conv2d(64, 64 * 2, 4, 2, 1)
        self.conv2_bn = nn.BatchNorm2d(64 * 2)
        self.conv3 = nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1)
        self.conv3_bn = nn.BatchNorm2d(64 * 4)
        self.conv4 = nn.Conv2d(64 * 4, 64 * 8, 4, 1, 1)
        self.conv4_bn = nn.BatchNorm2d(64 * 8)
        self.conv5 = nn.Conv2d(64 * 8, 1, 4, 1, 1)
        # delete end
        # ============== END OF CODE ================= #
        #++++++++++++++++++++++++++++++++++++++++++++++#
        #++++++++++++++++++++++++++++++++++++++++++++++#

    # weight_init
    def weight_init(self, mean, std):
        for m in self._modules:
            normal_init(self._modules[m], mean, std)

    # forward method
    def forward(self, input, label):
        #++++++++++++++++++++++++++++++++++++++++++++++#
        #++++++++++++++++++++++++++++++++++++++++++++++#
        # ============== YOUR CODE HERE ============== #
        # delete start
        x = torch.cat([input, label], 1)
        x = F.leaky_relu(self.conv1(x), 0.2)
        x = F.leaky_relu(self.conv2_bn(self.conv2(x)), 0.2)
        x = F.leaky_relu(self.conv3_bn(self.conv3(x)), 0.2)
        x = F.leaky_relu(self.conv4_bn(self.conv4(x)), 0.2)
        x = torch.sigmoid(self.conv5(x))
        # delete end
        # ============== END OF CODE ================= #
        #++++++++++++++++++++++++++++++++++++++++++++++#
        #++++++++++++++++++++++++++++++++++++++++++++++#


        return x
```

Step 3: Train

In this section, complete the function train. Then train two model: one with only L1 loss, the other with c=100.

```
# Helper function for showing result.
def process_image(img):
    return (img.cpu().data.numpy().transpose(1, 2, 0) + 1) / 2

def show_result(G, x_, y_, num_epoch):
    predict_images = G(x_)

    fig, ax = plt.subplots(x_.size()[0], 3, figsize=(10,30))

    for i in range(x_.size()[0]):
```

```python
        ax[i, 0].get_xaxis().set_visible(False)
        ax[i, 0].get_yaxis().set_visible(False)
        ax[i, 1].get_xaxis().set_visible(False)
        ax[i, 1].get_yaxis().set_visible(False)
        ax[i, 2].get_xaxis().set_visible(False)
        ax[i, 2].get_yaxis().set_visible(False)
        ax[i, 0].cla()
        ax[i, 0].imshow(process_image(x_[i]))
        ax[i, 1].cla()
        ax[i, 1].imshow(process_image(predict_images[i]))
        ax[i, 2].cla()
        ax[i, 2].imshow(process_image(y_[i]))

    plt.tight_layout()
    label_epoch = 'Epoch {0}'.format(num_epoch)
    fig.text(0.5, 0, label_epoch, ha='center')
    label_input = 'Input'
    fig.text(0.18, 1, label_input, ha='center')
    label_output = 'Output'
    fig.text(0.5, 1, label_output, ha='center')
    label_truth = 'Ground truth'
    fig.text(0.81, 1, label_truth, ha='center')

    plt.show()

# Helper function for counting number of trainable parameters.
def count_params(model):
    '''
    Counts the number of trainable parameters in PyTorch.
    Args:
        model: PyTorch model.
    Returns:
        num_params: int, number of trainable parameters.
    '''
    num_params = sum([item.numel() for item in model.parameters() if item.requires_grad])
    return num_params


# Hint: you could use following loss to complete following function
BCE_loss = nn.BCELoss().cuda()
L1_loss = nn.L1Loss().cuda()

def train(G, D, num_epochs = 20, only_L1 = False):
    hist_D_losses = []
    hist_G_losses = []
    #++++++++++++++++++++++++++++++++++++++++++++++++#
    #++++++++++++++++++++++++++++++++++++++++++++++++#
    # ============== YOUR CODE HERE ============== #
    # Adam optimizer
    G_optimizer = optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))
    D_optimizer = optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))
    #++++++++++++++++++++++++++++++++++++++++++++++++#
    #++++++++++++++++++++++++++++++++++++++++++++++++#
    # ============== YOUR CODE HERE ============== #

    print('training start!')
    start_time = time.time()
    for epoch in range(num_epochs):
        print('Start training epoch %d' % (epoch + 1))
        D_losses = []
        G_losses = []
```

```
        epoch_start_time = time.time()
        num_iter = 0
        for x_, _ in train_loader:

            y_ = x_[:, :, :, 0:img_size]
            x_ = x_[:, :, :, img_size:]

            x_, y_ = Variable(x_.cuda()), Variable(y_.cuda())
            # train discriminator D
            #++++++++++++++++++++++++++++++++++++++++++++++#
            #++++++++++++++++++++++++++++++++++++++++++++++#
            # ============== YOUR CODE HERE ============== #
            #delete start
            D.zero_grad()

            D_result = D(x_, y_).squeeze()
            D_real_loss = BCE_loss(D_result, Variable(torch.ones(D_result.size()).cuda()))

            G_result = G(x_)
            D_result = D(x_, G_result).squeeze()
            D_fake_loss = BCE_loss(D_result, Variable(torch.zeros(D_result.size()).cuda()))

            D_train_loss = (D_real_loss + D_fake_loss) * 0.5
            D_train_loss.backward()
            D_optimizer.step()
            # delete after =
            loss_D = D_train_loss.data
            # delete end
            #++++++++++++++++++++++++++++++++++++++++++++++#
            #++++++++++++++++++++++++++++++++++++++++++++++#
            # ============== YOUR CODE HERE ============== #

            D_losses.append(loss_D)
            hist_D_losses.append(loss_D)

            # train generator G
            #++++++++++++++++++++++++++++++++++++++++++++++#
            #++++++++++++++++++++++++++++++++++++++++++++++#
            # ============== YOUR CODE HERE ============== #
            # delete start
            G.zero_grad()

            G_result = G(x_)
            D_result = D(x_, G_result).squeeze()

            if only_L1:
                G_train_loss = L1_loss(G_result, y_)
            else:
                G_train_loss = BCE_loss(D_result, Variable(torch.ones(D_result.size()).cuda()))

            G_train_loss.backward()
            G_optimizer.step()
            # delete after =
            loss_G = G_train_loss.data
            # delete end
            #++++++++++++++++++++++++++++++++++++++++++++++#
            #++++++++++++++++++++++++++++++++++++++++++++++#
            # ============== YOUR CODE HERE ============== #

            G_losses.append(loss_G)
```

```
            G_losses.append(loss_G)
            hist_G_losses.append(loss_G)
            num_iter += 1


        epoch_end_time = time.time()
        per_epoch_ptime = epoch_end_time - epoch_start_time

        print('[%d/%d] - using time: %.2f' % ((epoch + 1), num_epochs, per_epoch_ptime))
        print('loss of discriminator D: %.3f' % (torch.mean(torch.FloatTensor(D_losses))))
        print('loss of generator G: %.3f' % (torch.mean(torch.FloatTensor(G_losses))))
        print('Sample Image:')
        show_result(G, Variable(fixed_x_.cuda(), volatile=True), fixed_y_, (epoch+1))

    end_time = time.time()
    total_ptime = end_time - start_time

    return hist_D_losses, hist_G_losses
```

In this part, train your model with c=100 with at leat 20 epoch.

```
# Define network
G_100 = generator()
D_100 = discriminator()
G_100.weight_init(mean=0.0, std=0.02)
D_100.weight_init(mean=0.0, std=0.02)
G_100.cuda()
D_100.cuda()
G_100.train()
D_100.train()

#Report the architectures of your network
print(G_100)
print('Number of trainable parameters {}'.format(count_params(G_100)))

print(D_100)
print('Number of trainable parameters {}'.format(count_params(D_100)))
```

```
#training
# TODO: change_num_epochs if you want
hist_D_100_losses, hist_G_100_losses = train(G_100, D_100, num_epochs = 20, only_L1 = False)
```

In this part, train your model with only L1 loss with at leat 10 epoch.

```
# Define network
G_L1 = generator()
D_L1 = discriminator()
G_L1.weight_init(mean=0.0, std=0.02)
D_L1.weight_init(mean=0.0, std=0.02)
G_L1.cuda()
D_L1.cuda()
G_L1.train()
D_L1.train()

#training
# TODO: change_num_epochs if you want
hist_D_L1_losses, hist_G_L1_losses = train(G_L1, D_L1, num_epochs = 10, only_L1 = True)
```

Step 4: Viulization

In this section, plot the G/D loss history v.s. Iteration of model with c=100 in seperate plot.

```
# plot the G/D loss history v.s. Iteration in one plot
#++++++++++++++++++++++++++++++++++++++++++++++#
#++++++++++++++++++++++++++++++++++++++++++++++#
# ============= YOUR CODE HERE ============= #
plt.clf()
#plt.plot(range(len(hist_D_100_losses)), hist_D_100_losses, label='D_loss')
plt.plot(range(len(hist_G_100_losses)), hist_G_100_losses, label='G_loss')

plt.xlabel('Iter')
plt.ylabel('Loss')

plt.legend(loc=4)
plt.grid(True)
plt.tight_layout()
plt.show()


plt.clf()
plt.plot(range(len(hist_D_100_losses)), hist_D_100_losses, label='D_loss')
```

```
#plt.plot(range(len(hist_G_100_losses)), hist_G_100_losses, label='G_loss')

plt.xlabel('Iter')
plt.ylabel('Loss')

plt.legend(loc=4)
plt.grid(True)
plt.tight_layout()
plt.show()
#+++++++++++++++++++++++++++++++++++++++++++++++#
#+++++++++++++++++++++++++++++++++++++++++++++++#
# ============== YOUR CODE HERE ============== #
```

In this section, plot the G/D loss history v.s. Iteration of model with only L1 loss in seperate plot.

```
# plot the G/D loss history v.s. Iteration in one plot
#+++++++++++++++++++++++++++++++++++++++++++++++#
#+++++++++++++++++++++++++++++++++++++++++++++++#
# ============== YOUR CODE HERE ============== #
plt.clf()
plt.plot(range(len(hist_D_L1_losses)), hist_D_L1_losses, label='D_loss')
#plt.plot(range(len(hist_G_L1_losses)), hist_G_L1_losses, label='G_loss')

plt.xlabel('Iter')
plt.ylabel('Loss')

plt.legend(loc=4)
plt.grid(True)
plt.tight_layout()
plt.show()

plt.clf()
#plt.plot(range(len(hist_D_L1_losses)), hist_D_L1_losses, label='D_loss')
plt.plot(range(len(hist_G_L1_losses)), hist_G_L1_losses, label='G_loss')

plt.xlabel('Iter')
plt.ylabel('Loss')

plt.legend(loc=4)
plt.grid(True)
plt.tight_layout()
plt.show()
#+++++++++++++++++++++++++++++++++++++++++++++++#
#+++++++++++++++++++++++++++++++++++++++++++++++#
# ============== YOUR CODE HERE ============== #
```