Paper Reproduction - "Image-to-Image Translation with Conditional Adversarial Networks"

By Xu Yingfu Zeng Liang Liu Hao

1. Introduction

Many problems in image processing, computer graphics and computer vision can be regarded as a problem that translate an input image to a new image with different style or format. Generative Adversarial Networks(GAN) is a of paramount method to implement the image translating. However, how effective image-conditional GANs can be as a general-purpose solition for image-to-image translation remains unclear. In this case, conditional GANs has been proposed in this paper [1] (https://arxiv.org/pdf/1611.07004.pdf (https://arxiv.org/pdf/1611.07004.pdf)) which is used to explore the effectness of image-condition GANs. And in this paper, it achives decent results on a wide variety of applications('Labels to Street Scene', 'Labels to Facade', 'BW to Color', 'Aerial to Map', 'Day to Night' and 'Edges to Photo'). For simplicity of expression, the method in [1] is called as "pix2pix" in the rest part of this report.

By inspired by this paper, we would like to reproduce this paper to gain a deeper insight by following three critria:

- New code variant (Rewrite existing code to be more efficient/readable)
- New data (Evaluating different datasets to obtain similar result)
- Hyperparamter check (Evaluating sensitivity to hyperparameters)

In this blog, we will walk through the following:

- section 2 Code Description for New code variant: we only illustrate the significant changed part of code
- section 3 New dataset: what dataset we choose and why choose this dataset
- section 4 The results of the New dataset
- section 5 Comparison on Different Network Architectures and Learning Rate
- section 6 Appendix

2. Code Description for New code variant

In this section, we rewrite the existing code from the readability and the efficiency two aspects.

- Readability: In the orginal code (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix) of paper [1], it mixes of "pix2pix" model code and "CycleGAN" model code which is quite difficult for readers to follow its logic. For example, the `Resnet` architecture is redundant in the "pix2pix" model. In this case, we pick up all "pix2pix" code and rewrite the code into modularity (dataset, dataloader, libraries, parameters, network, train, test)to give a clear logic.

- Efficiency: In the original code (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix) of paper [1], the dataset format is troublesome since the dataset we found are usually separate pictures and the picture size is unique. In this case, much time spents on changing picture size and combine pictures by `combine_A_and_B.py`. Inspired by the github code (https://github.com/mrzhu-cool/pix2pix-pytorch), we change the data format like the following. We don not need to change the data size and preprocess the dataset by `combine_A_and_B.py`.

```
1st class folder:       - facades
2nd class folder:     - test  - train
3rd class folder:       - a     - a
3rd class folder:       - b     - b
```

In the Appendix, the detailed explanation of the code will be inllustrated.

Besides, we noticed that there are several differences between the code and the description in the paper[1].

After the last layer, a convolution is applied to map to a 1-dimensional output, followed by a Sigmoid function. But in the code: "Note: Do not use sigmoid as the last layer of Discriminator. LSGAN needs no sigmoid."

For Loss GAN (the following Equation), CycleGAN replaces the negative log likelihood objective by a least-squares loss. This loss is more stable during training and generates higher quality results. Besides, in paper[1], the authors pointed out that previous approaches have found it beneficial to mix the GAN objective with a more traditional loss, such as L2 distance.

$$\begin{aligned}\mathcal{L}_{\mathrm{GAN}}\left(G, D_Y, X, Y\right) = & \mathbb{E}_{y \sim p_{\mathrm{data}}(y)}\left[\log D_Y(y)\right] \\ & + \mathbb{E}_{x \sim p_{\mathrm{data}}(x)}\left[\log(1 - D_Y(G(x))\right]\end{aligned}$$

In pix2pix, the 70 × 70 discriminator architecture is: C64-C128-C256-C512. Why it is called as 70 × 70 discriminator has been explained well in https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/issues/39 (https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/issues/39).

3. New dataset

pix2pix model has achieved impressive results in many image-to-image translation applications. In order to further explored the ability of pix2pix, we have designed 3 new application scenarios including the recovery of blurry images, Chinese calligraphy transformation, and turning depth images into RGB images. 3 new datasets were created for these applications respectively. In this section, the dataset creation process is introduced.

Blurry images

It is difficult to collect paired blurry and clear images so we had to take advantage of the existing image dataset. We applied multiple blurring to the images from PASCAL Visual Object Classes Challenge 2007, which consists of 2501 training images, 2510 validation images and 4952 test images regarding 20 common objects. For each image in the original dataset, 9 blurry images were created by gaussian blur, motion blur, and disk blur with random filter size and intensity as the example below. The new images are paired with the original one. The original partition is maintained. Thus, the new dataset has 22509 training images and 22590 validation images and 44568 test images.

Method	**-**	**-**
Gaussian blur	size in [3,15]	std in [1,5]
Motion blur	length in [8, 20]	angle in [0 360]
Disk blur	radius in [2,10]	

Chinese calligraphy

The Chinese calligraphy dataset is for training a model to transform characters of printing style to calligraphy. The images were generated from calligraphy fonts and those images of the same characters were made into pair. The dataset consists of 300 paired images for training, 100 for validation, and 100 for test.

Depth dataset

The depth grayscale images and RGB scene images are sampled from the SYNTHIA-AL (ICCV Workshops 2019) dataset downloaded from http://synthia-dataset.net/downloads/ (http://synthia-dataset.net/downloads/). SYNTHIA is a dataset that has been generated with the purpose of aiding semantic segmentation and related scene understanding problems in the context of driving scenarios. SYNTHIA consists of a collection of photo-realistic frames rendered from a virtual city and comes with precise pixel-level semantic annotations.

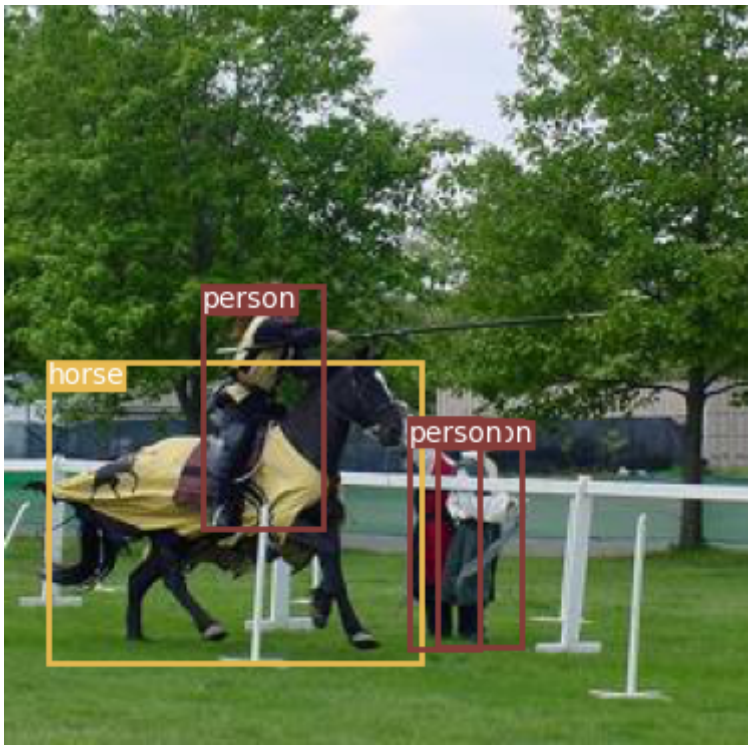
4. The results of the New dataset

4.1 The results of the deblurring dataset

In the following figures, the deblurring results of the default pix2pix project on a small test set are shown. The default pix2pix uses unet_256 as the architecture of the generator network. The learning rate is 0.0002 and decay linearly to zero in 10 epochs after it holds 0.0002 for 10 epochs.

This small test set is made up of 9 different kinds of artificial blur of the same image.

Original sharp image:



Original blurry images:

Results after 1 epoch of training:

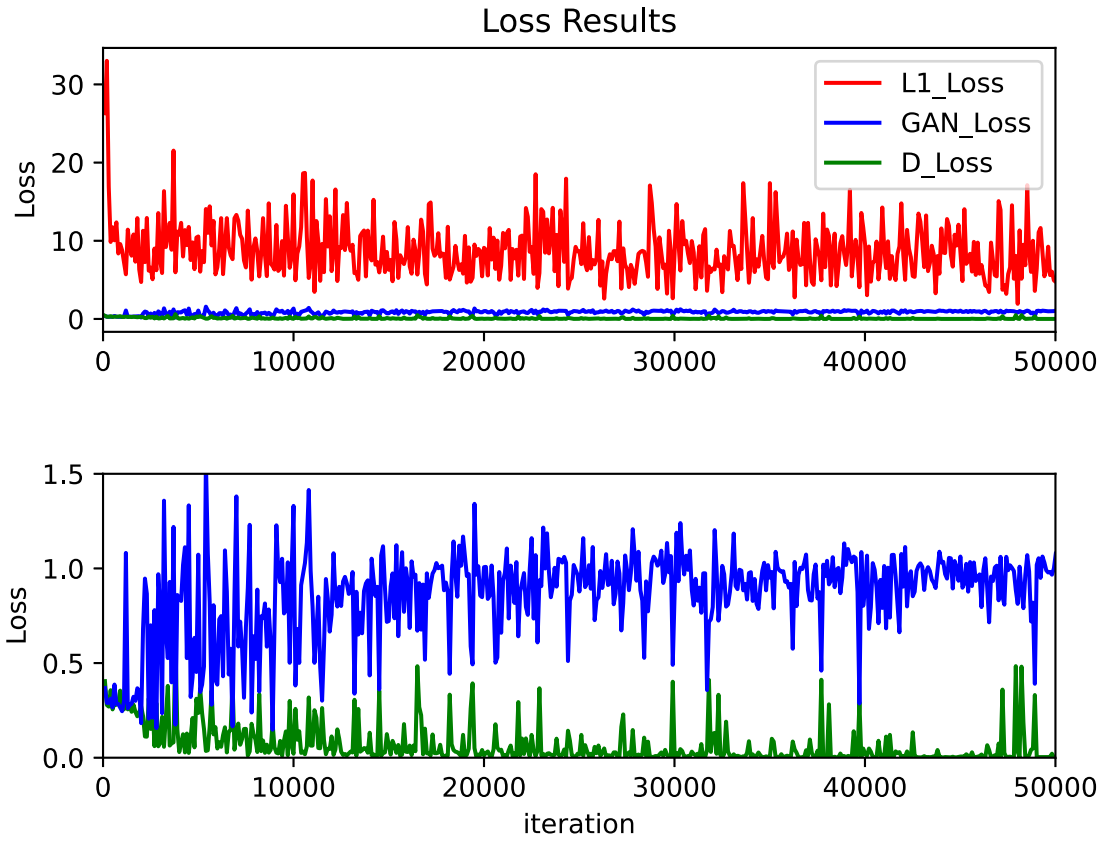
Results after 5 epochs of training:

Results after 10 epochs of training:

Results after 15 epochs of training:

Results after 20 epochs of training:

The following figure shows how the items of loss changes in the training.



4.2 The results of the depth dataset

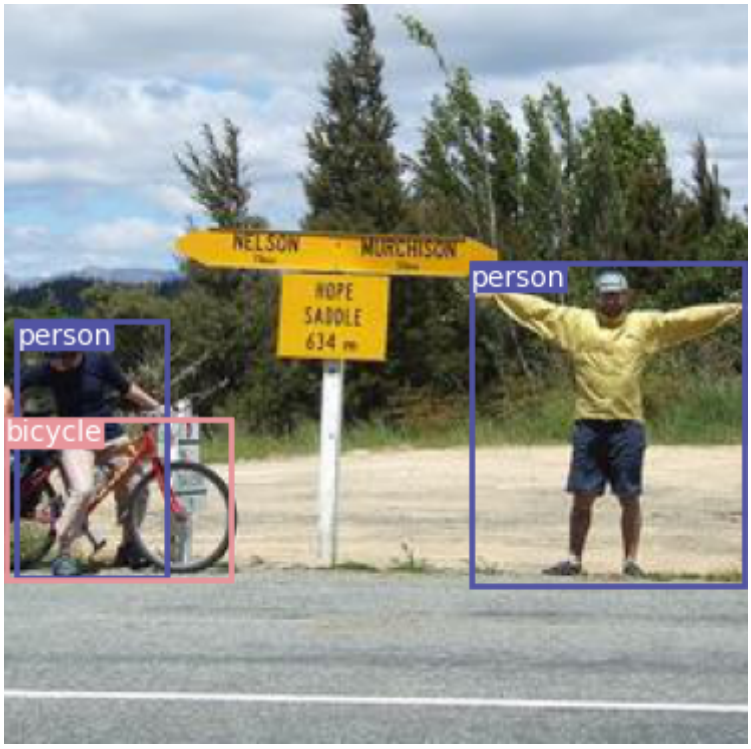
The results are shown in the "Comparison on Different Network Architectures and Learning Rate" part.

5. Comparison on Different Network Architectures and Learning Rate

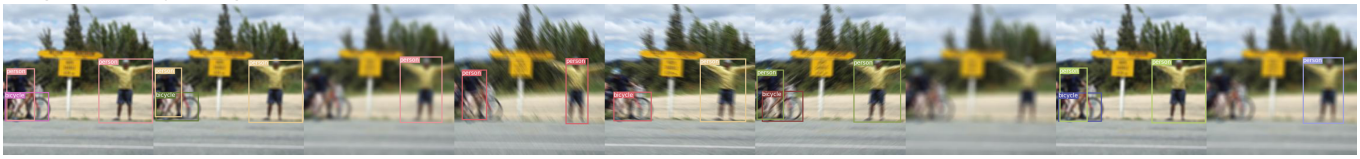
5.1 Deblurring

In the following figures, the deblurring results of 3 different models on a new small test set are shown. This test set is made up of 9 different kinds of artificial blur of the same image. The first three images from left have gaussian blur. The three images in the middle have motion blur. The three images on the right have disk blur.

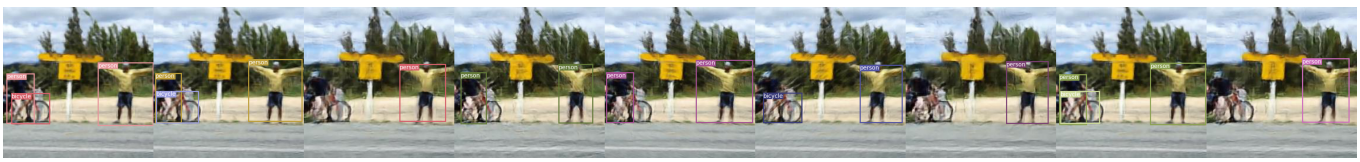
Shape image:



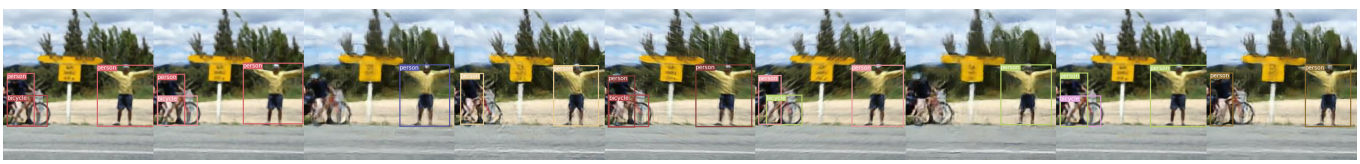
Original blurry images:



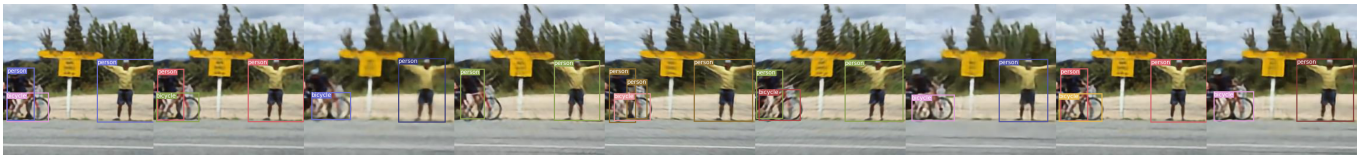
Results of unet 0.0002:



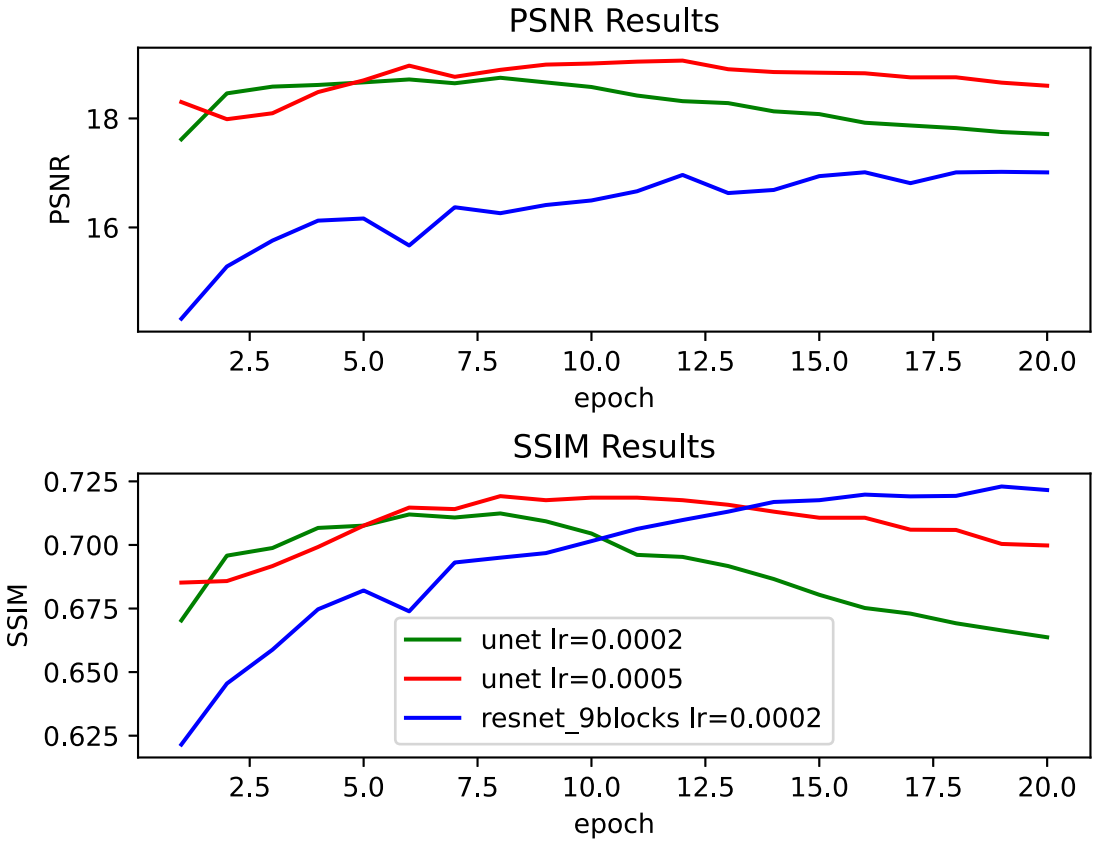
Results of unet 0.0005:



Results of resnet 0.0002:



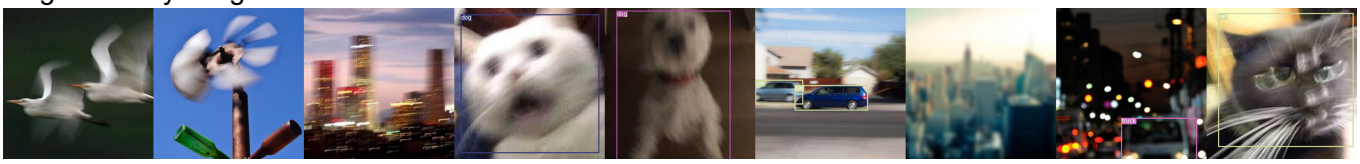
The following figure shows how the average PSNR and SSIM of the generated images of 3 different models in the visualization set changes with epoch.



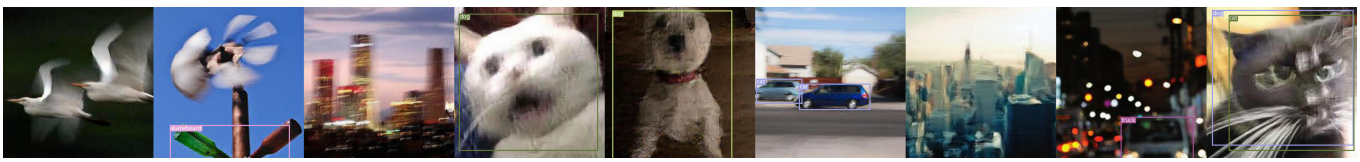
The average PSNR and SSIM of different models on the test set will be added later on 15 April 2020.

In the following figures, the deblurring results of 3 different models on a new small test set are shown. This test set is made up by originally blurry images found on the internet.

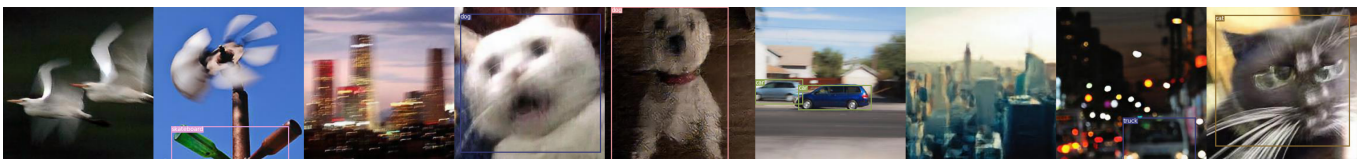
Original blurry images:



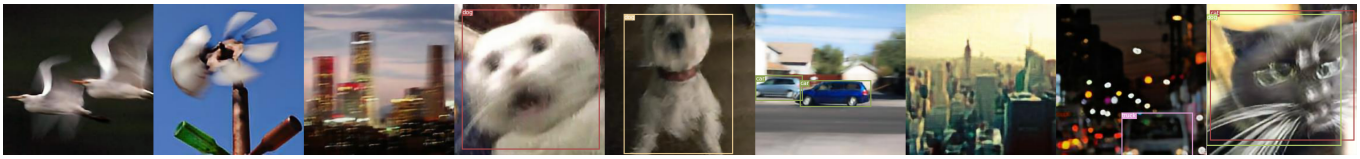
Results of unet 0.0002:



Results of unet 0.0005:



Results of resnet 0.0002:



5.2 Generating RGB images from graysacle depth images

In the following figures, the generated RGB images from input graysacle depth images of 2 different models on a small test set are shown. This test set is selected from the visualization dataset of depth dataset. Each image in this set has a unique scene in the simulatied city. These 7 kinds of scene are also included in the traning set.

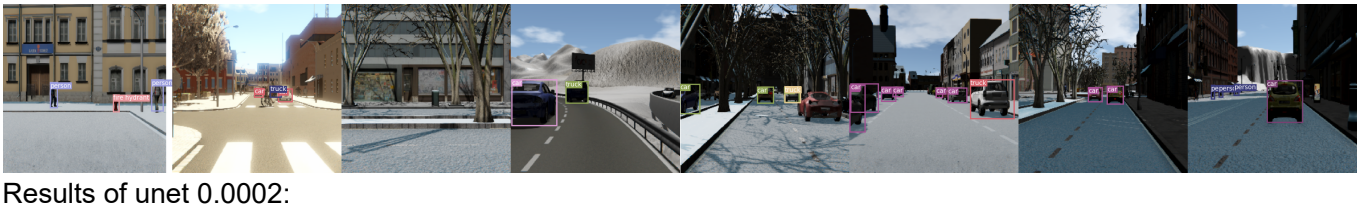
Original RGB images:

Results of unet 0.0002:

Results of resnet 0.0002:

In the following figures, the generated RGB images of 2 different models on a new small test set are shown. This test set is selected from the SYNTHIA-SF-19 dataset. Each image in this set has a unique scene in the simulatied city. These 7 kinds of scene are NOT included in the traning set. Input grayscale depth images:

Original RGB images:



Results of unet 0.0002:

Results of resnet 0.0002:

The average PSNR and SSIM of different models on the test set will be added later on 15 April 2020.

6. Appendix

In the below, we provide two possible ways to run our code to see the result.

- 1: you have to connect the colab to the google drive for accessing the datasets.
- 2: you direct download the github dataset folders(recommended)

Github clone:

In [0]:

```
!git clone https://github.com/YingfuXu/pix2pixCourseProject
```

Library used

In [0]:

```
from __future__ import print_function
import os
from os import listdir
from os.path import join
from math import log10
import torch
import torch.nn as nn
import torch.optim as optim
from torch.nn import init
from torch.optim import lr_scheduler
from torch.utils.data import DataLoader
import torch.backends.cudnn as cudnn
import functools
from matplotlib import pyplot as plt
import torch.utils.data as data
import numpy as np
from PIL import Image
import torchvision.transforms as transforms
import random
```

setting parameters

In [0]:

```
# change the dataset in the current directory
dataset = 'facades'

batch_size = 1
test_batch_size=1
# direction of the dataset
direction='b2a'
# number of channels
input_nc=3
output_nc=3
# the number of filters in the first convolution layer
ngf=64
ndf=64

epoch_count=1
niter=100
niter_decay=100

lr=0.0002
lr_policy='lambda'
lr_decay_iters=30
beta1=0.5

threads=0
seed=123
lamb=10

torch.manual_seed(seed)
# this is the cpu version
# device = torch.device("cpu")

# if using the gpu, open the following code
torch.cuda.manual_seed(seed)
device = torch.device("cuda: 0")
```

Funtions for loading dataset and preprocessing

In [0]:

```
def is_image_file(filename):
    return any(filename.endswith(extension) for extension in [".png", ".jpg", ".jpeg"])

def load_img(filepath):
    img = Image.open(filepath).convert('RGB')
    img = img.resize((256, 256), Image.BICUBIC)
    return img

def save_img(image_tensor, filename):
    image_numpy = image_tensor.float().numpy()
    image_numpy = (np.transpose(image_numpy, (1, 2, 0)) + 1) / 2.0 * 255.0
    image_numpy = image_numpy.clip(0, 255)
    image_numpy = image_numpy.astype(np.uint8)
    image_pil = Image.fromarray(image_numpy)
    image_pil.save(filename)
    print("Image saved as {}".format(filename))


# Inherit the data.Dataset and create a new dataset for getting each item easily
class DatasetFromFolder(data.Dataset):

    def __init__(self, image_dir, direction):
        super(DatasetFromFolder, self).__init__()
        self.direction = direction
        self.a_path = join(image_dir, "a")
        self.b_path = join(image_dir, "b")
        self.image_filenames = [x for x in listdir(self.a_path) if is_image_file(x)]

        transform_list = [transforms.ToTensor(),
                          transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
        self.transform = transforms.Compose(transform_list)

    def __getitem__(self, index):
        # covert into RGB picture, do they cover the original pictures?
        # Answer: this is just converting the original pictures
        # join : combine each paths in the list
        # Preprocessing of the pictures
        a = Image.open(join(self.a_path, self.image_filenames[index])).convert('RGB')
        b = Image.open(join(self.b_path, self.image_filenames[index])).convert('RGB')
        # Resize
        a = a.resize((286, 286), Image.BICUBIC)
        b = b.resize((286, 286), Image.BICUBIC)
        # To tensor
        a = transforms.ToTensor()(a)
        b = transforms.ToTensor()(b)
        # add a offset to the picture
        w_offset = random.randint(0, max(0, 286 - 256 - 1))
        h_offset = random.randint(0, max(0, 286 - 256 - 1))

        a = a[:, h_offset:h_offset + 256, w_offset:w_offset + 256]
        b = b[:, h_offset:h_offset + 256, w_offset:w_offset + 256]
        # Normalize
        a = transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))(a)
        b = transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))(b)


        if random.random() < 0.5:
            idx = [i for i in range(a.size(2) - 1, -1, -1)]
            idx = torch.LongTensor(idx)
```

```
            a = a.index_select(2, idx)
            b = b.index_select(2, idx)

        if self.direction == "a2b":
            return a, b
        else:
            return b, a

    def __len__(self):
        return len(self.image_filenames)
```

Design Network

In [0]:

```
class Inconv(nn.Module):
    def __init__(self, in_ch, out_ch, norm_layer, use_bias):
        super(Inconv, self).__init__()
        self.inconv = nn.Sequential(
            nn.ReflectionPad2d(3),
            nn.Conv2d(in_ch, out_ch, kernel_size=7, padding=0,
                      bias=use_bias),
            norm_layer(out_ch),
            nn.ReLU(True)
        )

    def forward(self, x):
        x = self.inconv(x)
        return x
# ngf, ngf * 2, norm_layer, use_bias
class Down(nn.Module):
    def __init__(self, in_ch, out_ch, norm_layer, use_bias):
        super(Down, self).__init__()
        self.down = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size=3,
                      stride=2, padding=1, bias=use_bias),
            norm_layer(out_ch),
            nn.ReLU(True)
        )

    def forward(self, x):
        x = self.down(x)
        return x

class Up(nn.Module):
    def __init__(self, in_ch, out_ch, norm_layer, use_bias):
        super(Up, self).__init__()
        self.up = nn.Sequential(
            # nn.Upsample(scale_factor=2, mode='nearest'),
            # nn.Conv2d(in_ch, out_ch,
            #           kernel_size=3, stride=1,
            #           padding=1, bias=use_bias),
            nn.ConvTranspose2d(in_ch, out_ch,
                               kernel_size=3, stride=2,
                               padding=1, output_padding=1,
                               bias=use_bias),
            norm_layer(out_ch),
            nn.ReLU(True)
        )

    def forward(self, x):
        x = self.up(x)
        return x


class Outconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(Outconv, self).__init__()
        self.outconv = nn.Sequential(
            nn.ReflectionPad2d(3),
            nn.Conv2d(in_ch, out_ch, kernel_size=7, padding=0),
            nn.Tanh()
        )
```

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

def init_net(net, init_type='normal', init_gain=0.02, gpu_id='cuda:0'):

    with torch.no_grad():
        net.to(gpu_id)
        init_weights(net, init_type, gain=init_gain)
    return net
```

gernerator

In [0]:

```
class UnetSkipConnectionBlock(nn.Module):
    """Defines the Unet submodule with skip connection.
        X ---------------------identity----------------------
        |-- downsampling -- |submodule| -- upsampling --|
    """

    def __init__(self, outer_nc, inner_nc, input_nc=None,
                 submodule=None, outermost=False, innermost=False, norm_layer=nn.BatchNorm2d,
use_dropout=False):
        """Construct a Unet submodule with skip connections.

        Parameters:
            outer_nc (int) -- the number of filters in the outer conv layer
            inner_nc (int) -- the number of filters in the inner conv layer
            input_nc (int) -- the number of channels in input images/features
            submodule (UnetSkipConnectionBlock) -- previously defined submodules
            outermost (bool)    -- if this module is the outermost module
            innermost (bool)    -- if this module is the innermost module
            norm_layer          -- normalization layer
            use_dropout (bool)  -- if use dropout layers.
        """
        super(UnetSkipConnectionBlock, self).__init__()
        self.outermost = outermost
        if type(norm_layer) == functools.partial:
            use_bias = norm_layer.func == nn.InstanceNorm2d
        else:
            use_bias = norm_layer == nn.InstanceNorm2d
        if input_nc is None:
            input_nc = outer_nc
        downconv = nn.Conv2d(input_nc, inner_nc, kernel_size=4,
                             stride=2, padding=1, bias=use_bias)
        downrelu = nn.LeakyReLU(0.2, True)
        downnorm = norm_layer(inner_nc)
        uprelu = nn.ReLU(True)
        upnorm = norm_layer(outer_nc)

        if outermost:
            upconv = nn.ConvTranspose2d(inner_nc * 2, outer_nc,
                                        kernel_size=4, stride=2,
                                        padding=1)
            down = [downconv]
            up = [uprelu, upconv, nn.Tanh()]
            model = down + [submodule] + up
        elif innermost:
            upconv = nn.ConvTranspose2d(inner_nc, outer_nc,
                                        kernel_size=4, stride=2,
                                        padding=1, bias=use_bias)
            down = [downrelu, downconv]
            up = [uprelu, upconv, upnorm]
            model = down + up
        else:
            upconv = nn.ConvTranspose2d(inner_nc * 2, outer_nc,
                                        kernel_size=4, stride=2,
                                        padding=1, bias=use_bias)
            down = [downrelu, downconv, downnorm]
            up = [uprelu, upconv, upnorm]

            if use_dropout:
                model = down + [submodule] + up + [nn.Dropout(0.5)]
```

```
            else:
                model = down + [submodule] + up

        self.model = nn.Sequential(*model)

    def forward(self, x):
        if self.outermost:
            return self.model(x)
        else:   # add skip connections
            return torch.cat([x, self.model(x)], 1)



class UnetGenerator(nn.Module):
    """Create a Unet-based generator"""

    def __init__(self, input_nc, output_nc, num_downs, ngf=64, norm_layer=nn.BatchNorm2d, use_dr
opout=False):
        """Construct a Unet generator
        Parameters:
            input_nc (int)  -- the number of channels in input images
            output_nc (int) -- the number of channels in output images
            num_downs (int) -- the number of downsamplings in UNet. For example, # if |num_downs
| == 7,
                                image of size 128x128 will become of size 1x1 # at the bottlenec
k
            ngf (int)       -- the number of filters in the last conv layer
            norm_layer      -- normalization layer

        We construct the U-Net from the innermost layer to the outermost layer.
        It is a recursive process.
        """
        super(UnetGenerator, self).__init__()
        # construct unet structure
        unet_block = UnetSkipConnectionBlock(ngf * 8, ngf * 8, input_nc=None, submodule=None,
                                             norm_layer=norm_layer, innermost=True)  # add the
 innermost layer
        for i in range(num_downs - 5):          # add intermediate layers with ngf * 8 filters
            unet_block = UnetSkipConnectionBlock(ngf * 8, ngf * 8, input_nc=None, submodule=une
t_block,
                                                 norm_layer=norm_layer, use_dropout=use_dropout)
        # gradually reduce the number of filters from ngf * 8 to ngf
        unet_block = UnetSkipConnectionBlock(ngf * 4, ngf * 8, input_nc=None, submodule=unet_bl
ock,
                                             norm_layer=norm_layer)
        unet_block = UnetSkipConnectionBlock(ngf * 2, ngf * 4, input_nc=None, submodule=unet_bl
ock,
                                             norm_layer=norm_layer)
        unet_block = UnetSkipConnectionBlock(ngf, ngf * 2, input_nc=None, submodule=unet_block,
                                             norm_layer=norm_layer)
        self.model = UnetSkipConnectionBlock(output_nc, ngf, input_nc=input_nc, submodule=unet_b
lock,
                                             outermost=True, norm_layer=norm_layer)  # add the
 outermost layer

    def forward(self, input):
        """Standard forward"""
        return self.model(input)


def define_G(input_nc, output_nc, ngf, norm='batch', use_dropout=False, init_type='normal', ini
```

```
t_gain=0.02, gpu_id='cuda:0'):
    net = None
    norm_layer = get_norm_layer(norm_type=norm)

#     net = ResnetGenerator(input_nc, output_nc, ngf, norm_layer=norm_layer, use_dropout=use_dro
pout, n_blocks=9)
    net = UnetGenerator(input_nc, output_nc, 8, ngf, norm_layer=norm_layer, use_dropout=use_drop
out)
    return init_net(net, init_type, init_gain, gpu_id)
```

Discriminator

In [0]:

```
class NLayerDiscriminator(nn.Module):
    def __init__(self, input_nc, ndf=64, n_layers=3, norm_layer=nn.BatchNorm2d, use_sigmoid=Fal
se):
        super(NLayerDiscriminator, self).__init__()
        if type(norm_layer) == functools.partial:
            use_bias = norm_layer.func == nn.InstanceNorm2d
        else:
            use_bias = norm_layer == nn.InstanceNorm2d

        kw = 4
        padw = 1
        sequence = [
            nn.Conv2d(input_nc, ndf, kernel_size=kw, stride=2, padding=padw),
            nn.LeakyReLU(0.2, True)
        ]

        nf_mult = 1
        nf_mult_prev = 1
        for n in range(1, n_layers):
            nf_mult_prev = nf_mult
            nf_mult = min(2**n, 8)
            sequence += [
                nn.Conv2d(ndf * nf_mult_prev, ndf * nf_mult,
                          kernel_size=kw, stride=2, padding=padw, bias=use_bias),
                norm_layer(ndf * nf_mult),
                nn.LeakyReLU(0.2, True)
            ]

        nf_mult_prev = nf_mult
        nf_mult = min(2**n_layers, 8)
        sequence += [
            nn.Conv2d(ndf * nf_mult_prev, ndf * nf_mult,
                      kernel_size=kw, stride=1, padding=padw, bias=use_bias),
            norm_layer(ndf * nf_mult),
            nn.LeakyReLU(0.2, True)
        ]

        sequence += [nn.Conv2d(ndf * nf_mult, 1, kernel_size=kw, stride=1, padding=padw)]

        if use_sigmoid:
            sequence += [nn.Sigmoid()]

        self.model = nn.Sequential(*sequence)

    def forward(self, input):
        return self.model(input)


class PixelDiscriminator(nn.Module):
    def __init__(self, input_nc, ndf=64, norm_layer=nn.BatchNorm2d, use_sigmoid=False):
        super(PixelDiscriminator, self).__init__()
        if type(norm_layer) == functools.partial:
            use_bias = norm_layer.func == nn.InstanceNorm2d
        else:
            use_bias = norm_layer == nn.InstanceNorm2d

        self.net = [
            nn.Conv2d(input_nc, ndf, kernel_size=1, stride=1, padding=0),
            nn.LeakyReLU(0.2, True),
```

```
            nn.Conv2d(ndf, ndf * 2, kernel_size=1, stride=1, padding=0, bias=use_bias),
            norm_layer(ndf * 2),
            nn.LeakyReLU(0.2, True),
            nn.Conv2d(ndf * 2, 1, kernel_size=1, stride=1, padding=0, bias=use_bias)]

        if use_sigmoid:
            self.net.append(nn.Sigmoid())

        self.net = nn.Sequential(*self.net)

    def forward(self, input):
        return self.net(input)

def define_D(input_nc, ndf, netD,
             n_layers_D=3, norm='batch', use_sigmoid=False, init_type='normal', init_gain=0.02,
gpu_id='cuda:0'):
    net = None
    norm_layer = get_norm_layer(norm_type=norm)

    if netD == 'basic':
        net = NLayerDiscriminator(input_nc, ndf, n_layers=3, norm_layer=norm_layer, use_sigmoid=
use_sigmoid)
    elif netD == 'n_layers':
        net = NLayerDiscriminator(input_nc, ndf, n_layers_D, norm_layer=norm_layer, use_sigmoid=
use_sigmoid)
    elif netD == 'pixel':
        net = PixelDiscriminator(input_nc, ndf, norm_layer=norm_layer, use_sigmoid=use_sigmoid)
    else:
        raise NotImplementedError('Discriminator model name [%s] is not recognized' % net)

    return init_net(net, init_type, init_gain, gpu_id)
```

init_param, loss fn, optimizer

In [0]:

```
def init_weights(net, init_type='normal', gain=0.02):
    def init_func(m):
        classname = m.__class__.__name__
        if hasattr(m, 'weight') and (classname.find('Conv') != -1 or classname.find('Linear') !
= -1):
            if init_type == 'normal':
                init.normal_(m.weight.data, 0.0, gain)
            elif init_type == 'xavier':
                init.xavier_normal_(m.weight.data, gain=gain)
            elif init_type == 'kaiming':
                init.kaiming_normal_(m.weight.data, a=0, mode='fan_in')
            elif init_type == 'orthogonal':
                init.orthogonal_(m.weight.data, gain=gain)
            else:
                raise NotImplementedError('initialization method [%s] is not implemented' %
init_type)
            if hasattr(m, 'bias') and m.bias is not None:
                init.constant_(m.bias.data, 0.0)
        elif classname.find('BatchNorm2d') != -1:
            init.normal_(m.weight.data, 1.0, gain)
            init.constant_(m.bias.data, 0.0)

    print('initialize network with %s' % init_type)
    net.apply(init_func)



def get_norm_layer(norm_type='instance'):
    if norm_type == 'batch':
        norm_layer = functools.partial(nn.BatchNorm2d, affine=True)
    elif norm_type == 'instance':
        norm_layer = functools.partial(nn.InstanceNorm2d, affine=False, track_running_stats=Fal
se)
    elif norm_type == 'switchable':
        norm_layer = SwitchNorm2d
    elif norm_type == 'none':
        norm_layer = None
    else:
        raise NotImplementedError('normalization layer [%s] is not found' % norm_type)
    return norm_layer


class GANLoss(nn.Module):
    def __init__(self, use_lsgan=True, target_real_label=1.0, target_fake_label=0.0):
        super(GANLoss, self).__init__()
        self.register_buffer('real_label', torch.tensor(target_real_label))
        self.register_buffer('fake_label', torch.tensor(target_fake_label))
        if use_lsgan:
            self.loss = nn.MSELoss()
        else:
            # binary cross entropy
            self.loss = nn.BCELoss()

    def get_target_tensor(self, input, target_is_real):
        if target_is_real:
            target_tensor = self.real_label
        else:
            target_tensor = self.fake_label
        return target_tensor.expand_as(input)
```

```
    def __call__(self, input, target_is_real):
        target_tensor = self.get_target_tensor(input, target_is_real)
        return self.loss(input, target_tensor)

def update_learning_rate(scheduler, optimizer):
    scheduler.step()
    lr = optimizer.param_groups[0]['lr']
    print('learning rate = %.7f' % lr)

# learning rate decay
def get_scheduler(optimizer):
    if lr_policy == 'lambda':
        def lambda_rule(epoch):
            lr_l = 1.0 - max(0, epoch + epoch_count - niter) / float(niter_decay + 1)
            return lr_l
        scheduler = lr_scheduler.LambdaLR(optimizer, lr_lambda=lambda_rule)
    elif lr_policy == 'step':
        scheduler = lr_scheduler.StepLR(optimizer, step_size=lr_decay_iters, gamma=0.1)
    elif lr_policy == 'plateau':
        scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.2, threshold=
0.01, patience=5)
    elif lr_policy == 'cosine':
        scheduler = lr_scheduler.CosineAnnealingLR(optimizer, T_max=opt.niter, eta_min=0)
    else:
        return NotImplementedError('learning rate policy [%s] is not implemented', opt.lr_po
licy)
    return scheduler
```

train

In [0]:

```
dataroot1 = "datasets/facades/train"
dataroot2 = "datasets/facades/test"
# As for windows, delete num_works parameter
training_data_loader = DataLoader(dataset=DatasetFromFolder(dataroot1, direction), num_workers=t
hreads, batch_size=batch_size, shuffle=True)
testing_data_loader = DataLoader(dataset=DatasetFromFolder(dataroot2, direction), num_workers=th
reads, batch_size=test_batch_size)


print('===> Building models')

'''loading the generator and discriminator'''
net_g = define_G(input_nc, output_nc, ngf, 'batch', False, 'normal', 0.02, gpu_id=device)
net_d = define_D(input_nc + output_nc, ndf, 'basic', gpu_id=device)

'''set loss fn'''
criterionGAN = GANLoss().to(device)
criterionL1 = nn.L1Loss().to(device)
criterionMSE = nn.MSELoss().to(device)

'''setup optimizer'''
optimizer_g = optim.Adam(net_g.parameters(), lr=lr, betas=(beta1, 0.999))
optimizer_d = optim.Adam(net_d.parameters(), lr=lr, betas=(beta1, 0.999))

'''set the learning rate adjust policy'''
net_g_scheduler = get_scheduler(optimizer_g)
net_d_scheduler = get_scheduler(optimizer_d)

'''training process'''
for epoch in range(epoch_count, niter + niter_decay + 1):

    for iteration, batch in enumerate(training_data_loader, 1):
        # forward
        real_a, real_b = batch[0].to(device), batch[1].to(device)
        fake_b = net_g(real_a)

        ########################
        # (1) Update D network
        ########################

        optimizer_d.zero_grad()

        # D train with fake
        fake_ab = torch.cat((real_a, fake_b), 1)
        pred_fake = net_d.forward(fake_ab.detach())
        loss_d_fake = criterionGAN(pred_fake, False)

        # D train with real
        real_ab = torch.cat((real_a, real_b), 1)
        pred_real = net_d.forward(real_ab)
        loss_d_real = criterionGAN(pred_real, True)

        # Combined D loss
        loss_d = (loss_d_fake + loss_d_real) * 0.5

        loss_d.backward()

        optimizer_d.step()
```

```
        ########################
        # (2) Update G network
        ########################

        optimizer_g.zero_grad()

        # First, G(A) should fake the discriminator
        fake_ab = torch.cat((real_a, fake_b), 1)
        pred_fake = net_d.forward(fake_ab)
        loss_g_gan = criterionGAN(pred_fake, True)

        # Second, G(A) = B
        loss_g_l1 = criterionL1(fake_b, real_b) * lamb

        loss_g = loss_g_gan + loss_g_l1

        loss_g.backward()

        optimizer_g.step()

        print("===> Epoch[{}]({}/{}): Loss_D: {:.4f} Loss_G: {:.4f}".format(
            epoch, iteration, len(training_data_loader), loss_d.item(), loss_g.item()))

    update_learning_rate(net_g_scheduler, optimizer_g)
    update_learning_rate(net_d_scheduler, optimizer_d)

    # test
    avg_psnr = 0
    for batch in testing_data_loader:
        input, target = batch[0].to(device), batch[1].to(device)

        prediction = net_g(input)
        mse = criterionMSE(prediction, target)
        psnr = 10 * log10(1 / mse.item())
        avg_psnr += psnr
    print("===> Avg. PSNR: {:.4f} dB".format(avg_psnr / len(testing_data_loader)))

    #checkpoint
    if epoch % 50 == 0:
        if not os.path.exists("checkpoint"):
            os.mkdir("checkpoint")
        if not os.path.exists(os.path.join("checkpoint", dataset)):
            os.mkdir(os.path.join("checkpoint", dataset))
        net_g_model_out_path = "checkpoint/{}/netG_model_epoch_{}.pth".format(dataset, epoch)
        net_d_model_out_path = "checkpoint/{}/netD_model_epoch_{}.pth".format(dataset, epoch)
        torch.save(net_g, net_g_model_out_path)
        torch.save(net_d, net_d_model_out_path)
        print("Checkpoint saved to {}".format("checkpoint" + dataset))
```

Test

In [0]:

```
nepochs = 150
model_path = "checkpoint/{}/netG_model_epoch_{}.pth".format(dataset, nepochs)

net_g = torch.load(model_path).to(device)

if direction == "a2b":
    image_dir = "datasets/{}/test/a/".format(dataset)
else:
    image_dir = "datasets/{}/test/b/".format(dataset)

image_filenames = [x for x in os.listdir(image_dir) if is_image_file(x)]

transform_list = [transforms.ToTensor(),
                  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]

transform = transforms.Compose(transform_list)

for image_name in image_filenames:
    img = load_img(image_dir + image_name)
    img = transform(img)
    input = img.unsqueeze(0).to(device)
    out = net_g(input)
    out_img = out.detach().squeeze(0).cpu()

    if not os.path.exists(os.path.join("result", dataset)):
        os.makedirs(os.path.join("result", dataset))
    save_img(out_img, "result/{}/{}".format(dataset, image_name))
```

```
---------------------------------------------------------------------------
---
NameError                                 Traceback (most recent call last)
<ipython-input-12-be723b2f1915> in <module>
     14
     15 nepochs = 150
---> 16 model_path = "checkpoint/{}/netG_model_epoch_{}.pth".format(dataset
, nepochs)
     17
     18 net_g = torch.load(model_path).to(device)

NameError: name 'dataset' is not defined
```