Weekly Work Report

Yufeng Jiang

VISION@OUC

September 2, 2018

1 Research problem

The main task is adding some codes to modify the network and develop extended applications for [2]. And comparing to other similar papers published recently, we want to get a better results in order to prove advantages of ours.

2 Research approach

Modifing the generaotr network in dilated convolution is the first job I need to do.

3 Research progress

Now, I have implemented two kinds of networks with dilated convolution layers to enlarge the receptive field of feature points without reducing the resolution of the feature maps. Results of the first modified networks has been compared to the basic codes [1], but results are unsatisfactory.

4 Progress in this week

In this week, I have got results of second network with dilated convolution layers. Reading the self-attention paper [3] to get a basic understanding of self-attention based on GAN. Besides, I also modify the first dilated convolution network and the basic network to get better results.

4.1 D-LinkNet vs DRPAN

After interrupting several times, I got the final result testing at cityscape dataset with the codes using the D-LinkNet [4] structure. From the Figure 1, Figure 2 and Figure 3, it is clearly see that the D-LinkNet network result looks worse than the DRPAN network result in terms of detail and color. Although the D-LinkNet generated image generates more details in some part, their color information has been lost so that all generated images are using white as the keynote. And the D-LinkNet generated image looks even more chaotic.

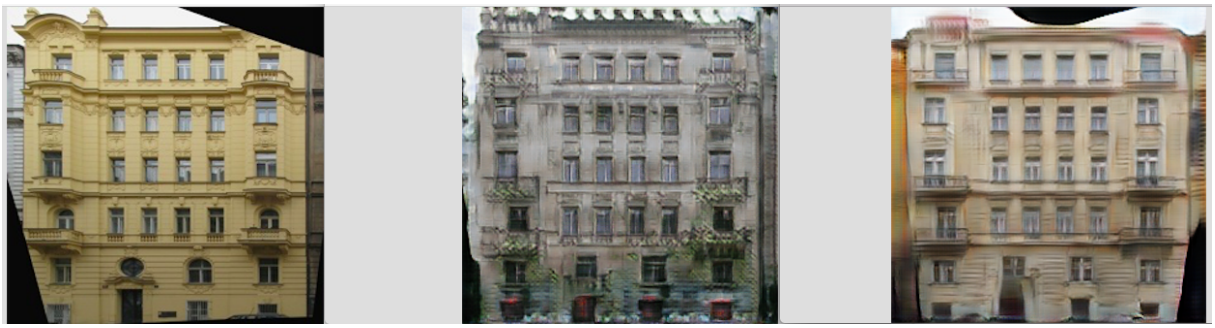

Figure 1: **Left:** The real image. **Center:** The D-LinkNet generated image. **Right:** The DRPAN generated image.

Possible reasons for this situation may be the encoder part of D-LinkNet using resnet34 is too deep, and the encoder part and the decoder part is not strictly symmetric. The encoder part is too deep while the decoder part only 5 layers. After talking to Chao Wang, we decide to drop out this D-LinkNet network and change to modify the first dilated convolution network.

4.2 The modified dilated convolution network

Considering I did not use normalization which may result to the complete collapse at all network, I add $InstanceNorm2d$ to some layers and use tanh function as activation function instead of clip the value

Figure 2: **Left:** The real image. **Center:** The D-LinkNet generated image. **Right:** The DRPAN generated image.

Figure 3: **Left:** The real image. **Center:** The D-LinkNet generated image. **Right:** The DRPAN generated image.

with *torch.clamp*.

```
class Dialted_Generators(nn.Module):
    def __init__(self, input_nc, output_nc, ngf=32):
        super(Dialted_Generators, self).__init__()
        self.conv1 = nn.Conv2d(input_nc, ngf, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(ngf, ngf * 2, kernel_size=3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(ngf * 2, ngf * 2, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(ngf * 2, ngf * 4, kernel_size=3, stride=2, padding=1)
        self.conv5 = nn.Conv2d(ngf * 4, ngf * 4, kernel_size=3, stride=1, padding=1)
        self.conv6 = nn.Conv2d(ngf * 4, ngf * 4, kernel_size=3, stride=1, padding=1)
        self.conv7 = nn.Conv2d(ngf * 4, ngf * 4, kernel_size=3, stride=1, padding=2,
                                                  dilation=2)
        self.conv8 = nn.Conv2d(ngf * 4, ngf * 4, kernel_size=3, stride=1, padding=4,
                                                  dilation=4)
        self.conv9 = nn.Conv2d(ngf * 4, ngf * 4, kernel_size=3, stride=1, padding=8,
                                                  dilation=8)
        self.conv10 = nn.Conv2d(ngf * 4, ngf * 4, kernel_size=3, stride=1, padding=16,
                                                  dilation=16)
        self.conv11 = nn.Conv2d(ngf * 4, ngf * 4, kernel_size=3, stride=1, padding=1)
        self.conv12 = nn.Conv2d(ngf * 4, ngf * 4, kernel_size=3, stride=1, padding=1)
        self.resize1 = nn.UpsamplingNearest2d(scale_factor=2)
        self.conv13 = nn.Conv2d(ngf * 4, ngf * 2, kernel_size=3, stride=1, padding=1)
        self.conv14 = nn.Conv2d(ngf * 2, ngf * 2, kernel_size=3, stride=1, padding=1)
        self.resize2 = nn.UpsamplingNearest2d(scale_factor=2)
        self.conv15 = nn.Conv2d(ngf * 2, ngf, kernel_size=3, stride=1, padding=1)
        self.conv16 = nn.Conv2d(ngf, ngf / 2, kernel_size=3, stride=1, padding=1)
        self.conv17 = nn.Conv2d(ngf / 2, output_nc, kernel_size=3, stride=1, padding=1)

        self.instance_norm = nn.InstanceNorm2d(ngf)
        self.instance_norm2 = nn.InstanceNorm2d(ngf * 2)
        self.instance_norm4 = nn.InstanceNorm2d(ngf * 4)
        self.instance_norm1 = nn.InstanceNorm2d(ngf / 2)

        self.elu = nn.ELU(True)

        self.tanh = nn.Tanh()

    def forward(self, input):
        c1 = self.instance_norm(self.conv1(self.elu(input)))
        c2 = self.instance_norm2(self.conv2(self.elu(c1)))
        c3 = self.instance_norm2(self.conv3(self.elu(c2)))
        c4 = self.instance_norm4(self.conv4(self.elu(c3)))
        c5 = self.instance_norm4(self.conv5(self.elu(c4)))
        c6 = self.instance_norm4(self.conv6(self.elu(c5)))
        c7 = self.instance_norm4(self.conv7(self.elu(c6)))
        c8 = self.instance_norm4(self.conv8(self.elu(c7)))
        c9 = self.instance_norm4(self.conv9(self.elu(c8)))
        c10 = self.instance_norm4(self.conv10(self.elu(c9)))
        c11 = self.instance_norm4(self.conv11(self.elu(c10)))
        c12 = self.instance_norm4(self.conv12(self.elu(c11)))
        r1 = self.resize1(c12)
        c13 = self.conv13(self.elu(r1))
        c14 = self.instance_norm2(self.conv14(self.elu(c13)))
        r2 = self.resize2(c14)
        c15 = self.conv15(self.elu(r2))
        c16 = self.instance_norm1(self.conv16(self.elu(c15)))
        c17 = self.conv17(self.elu(c16))
        output = self.tanh(c17)
        # output = torch.clamp(c17, -1, 1)
        return output
```

Because my code has been interruotted for several times by some reasons, So, I learn two methods to ensure the safety of running codes. The one is that if it interrupts, I need to run it from the original epoch 0 that it wastes a lot of time to train. Thus, adding codes for saving models is necessary. Firstly, define a new parameter *resume_epoch* to indicate which epoch I choose to start.

```
parser.add_option('--resume_epoch', type=int, default=0, help='resume model')
```

```
    def resume(self):
        self.netG.load_state_dict(torch.load('%s/netG_epoch_%03d.pth' % (self.config['
                                                outf'], self.resume_epoch)))
        self.netD.load_state_dict(torch.load('%s/netD_epoch_%03d.pth' % (self.config['
                                                outf'], self.resume_epoch)))
        self.netD_r.load_state_dict(torch.load('%s/netD_r_epoch_%03d.pth' % (self.config
                                                ['outf'], self.resume_epoch)))
```

Then, set up model to use the *resume* part, and save once time every 5 epoch. When I want to restart training, the one thing I need to do is adding $--$ *resume_epochnumber* where number can be instead by the latest epoch.

```
# setup model
    trainer = trainer_gan(config, train_loader, resume_epoch=opt.resume_epoch)

    if opt.cuda:
        trainer.cuda()
    if opt.resume_epoch:
        trainer.resume()
    # training
    for epoch in range(opt.resume_epoch, config['nepoch']):
        trainer.train(epoch)
        if epoch % 5 == 0:
            trainer.save(epoch)
```

The other method is using *tmux* to keep the program running in the background. I use
tmux new $-$ *s session name* $-$ *n window name*
to start a new session. This modify dilated convolution code is still running and I have not tested it.

4.3 The modified resnet network

The basic code uses 9 resnet blocks at the center. The main task is adding dilated convolution layers to generator, why I do not attempt to add dilated convolutions to DRPnet directly. Taking all factors into consideration, I decide to change the four layers with dilated convolution layers in these resnet blocks. For the symmetric of the whole network, so I use 8 resnet blocks where the center four layers using dilated convolution with 2, 4, 8, 16. This code is also running at the server now.

```
class ResnetGenerator(nn.Module):
    def __init__(self, input_nc, output_nc, ngf=64, norm_layer=nn.InstanceNorm2d,
                                                use_dropout=False, n_blocks=6):
        assert(n_blocks >= 0)
        super(ResnetGenerator, self).__init__()
        self.input_nc = input_nc
        self.output_nc = output_nc
        self.ngf = ngf

        model = [nn.Conv2d(input_nc, ngf, kernel_size=7, padding=3),
                 norm_layer(ngf, affine=True),
                 nn.ReLU(True)]
        # downsampling size for different image size
        n_downsampling = 2
        for i in range(n_downsampling):
            mult = 2**i
            model += [nn.Conv2d(ngf * mult, ngf * mult * 2, kernel_size=3,
                                stride=2, padding=1),
                      norm_layer(ngf * mult * 2, affine=True),
                      nn.ReLU(True)]

        mult = 2**n_downsampling
        model += [ResnetBlock(ngf * mult, 'zero', norm_layer=norm_layer, use_dropout=
                                                use_dropout)]

        for i in range(n_downsampling):
            mult = 2**(n_downsampling - i)
            model += [nn.ConvTranspose2d(ngf * mult, int(ngf * mult / 2),
                                         kernel_size=3, stride=2,
                                         padding=1, output_padding=1),
```

```
                      norm_layer(int(ngf * mult / 2), affine=True),
                      nn.ReLU(True)]

        model += [nn.Conv2d(ngf, output_nc, kernel_size=7, padding=3)]
        model += [nn.Tanh()]

        self.model = nn.Sequential(*model)

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


class ResnetBlock(nn.Module):
    def __init__(self, dim, padding_type, norm_layer, use_dropout):
        super(ResnetBlock, self).__init__()
        self.conv_block = self.build_conv_block(dim, padding_type, norm_layer,
                                                use_dropout)

    def build_conv_block(self, dim, padding_type, norm_layer, use_dropout):
        conv_block = []

        p = 0
        # TODO: support padding types
        assert (padding_type == 'zero')

        p = 1
        for i in range (2):
            conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=p),
                           norm_layer(dim, affine=True),
                           nn.ReLU(True)]
            if use_dropout:
                conv_block += [nn.Dropout(0.5)]
            conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=p),
                           norm_layer(dim, affine=True)]

        n_dilation = 4
        for i in range (n_dilation):
            p = 2 ** (i+1)
            conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=p, stride=1,
                                                    dilation=p),
                           norm_layer(dim, affine=True),
                           nn.ReLU(True)]
            if use_dropout:
                conv_block += [nn.Dropout(0.5)]
            conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=p, stride=1,
                                                    dilation=p),
                           norm_layer(dim, affine=True)]

        p = 1
        for j in range(2):
            conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=p),
                           norm_layer(dim, affine=True),
                           nn.ReLU(True)]
            if use_dropout:
                conv_block += [nn.Dropout(0.5)]
            conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=p),
                           norm_layer(dim, affine=True)]

        return nn.Sequential(*conv_block)

    def forward(self, x):
        out = x + self.conv_block(x)
        return out
```

References

[1] GitHub_godisboy. https://github.com/godisboy/DRPAN. 1

[2] C. Wang, H. Zheng, Z. Yu, Z. Zheng, Z. Gu, and B. Zheng. Discriminative region proposal adversarial networks for high-quality image-to-image translation. In *ECCV*, 2018. 1

[3] H. Zhang, L. Goodfellow, D. Metaxas, and A. Odena. Self-attention generative adversarial networks. *arXiv:1805.08318*, 2018. 1

[4] L. Zhou, C. Zhang, and M. Wu. D-LinkNet: LinkNet with pretrained encoder and dilated convolution for high resolution satellite imagery road extraction. In *CVPR*, 2018. 1