DLP-Lab7

ILE 310553043 You-Pin, Chen

June 7, 2022

1 Introduction

In this lab, we need to implement a conditional GAN [1] to generate synthetic images according to multi-label conditions. Recall that we have seen the generation capability of conditional VAE in the Lab 5. To achieve higher generation capacity, especially in computer vision, generative adversarial network is proposed and has been widely applied on style transfer and image synthesis. In this lab, given a specific condition, your model should generate the corresponding synthetic images. For example, given *red cube* and *blue cylinder*, your model should generate the synthetic images with red cube and blue cylinder and meanwhile, input your generated images to a pre-trained classifier for evaluation. For the implementation, we choose deep convolutional generative adversarial networks [4] as out main architecture and used projection discriminator [2] to tackle the conditions. The code is available in github[link].

2 Implementation

2.1 conditional DCGAN + ACGAN

2.1.1 Generator

About the Generator, we followed the DCGAN structure. We first embedded 24 conditions into 300 dimensions vector with a fully connected layer to expand the information. Then we concated the new embedded condition vector with z which is sampled from normal distribution as input of generator. Main structure of generator is referenced by [4]. It contains four convolutional layers while the proportion of channel is follow by $8:4:2:1$. There are batch normalization and ReLU behind in each convolutional layer. For the last layer, it is another convolutional layer while the output channel is RGB and we used tanh as activation function. Fig.1 illustrates the structure of generator while gray vector represents sample z and red vector represents condition vector.

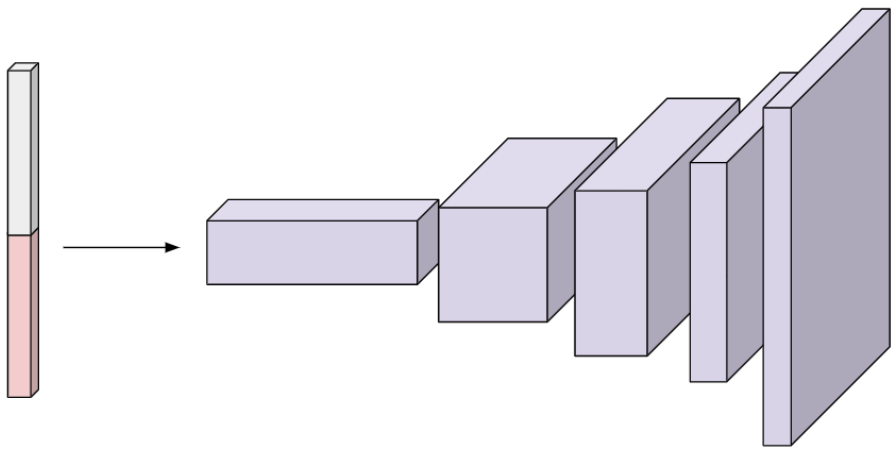


Figure 1: Illustration of Generator in our model.

```
class Generator(nn.Module):
    def __init__(self, args):
        super(Generator, self).__init__()
        self.rgb = 3
        self.nz = args.nz
        self.nc = args.nc

        self.map= nn.Sequential(
            nn.Linear(args.num_conditions, args.nc),
            nn.ReLU(True))

        self.main = nn.Sequential(
            nn.ConvTranspose2d(args.nz + args.nc, args.ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(args.ngf * 8),
            nn.ReLU(True),

            nn.ConvTranspose2d(args.ngf * 8, args.ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(args.ngf * 4),
            nn.ReLU(True),

            nn.ConvTranspose2d(args.ngf * 4, args.ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(args.ngf * 2),
            nn.ReLU(True),

            nn.ConvTranspose2d(args.ngf * 2, args.ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(args.ngf),
            nn.ReLU(True),

            nn.ConvTranspose2d(args.ngf, self.rgb, 4, 2, 1, bias=False),
            nn.Tanh())

    def forward(self, z, c):
        z = z.reshape(-1, self.nz, 1, 1)
        c = self.map(c).reshape(-1, self.nc, 1, 1)
        x = torch.cat((z, c), dim=1)
        out = self.main(x)
        return out
```

2.1.2 Discriminator

Then, about the discriminator, we had tried many different structure but they all still based on [4]. Basically, the structure is similar with generator but with inverse version. The proportion of each layer's channel is follow by $1:2:4:8:16$ while the activation function replace to Leaky ReLU and the last layer convert the channel to one with sigmoid function behind it. There are two ways to deal with the condition, first is that we transferred the condition vector to the shape of image size and concated with the input image as a new input. The figure illustrates in Fig.3. Second one is tackling the condition vector after the input image pass through four convolutional layers in discriminator. The method we called *Projection Discriminator*. The equation exhibits in Eq.2.1.1 while x is the input image, matrix V is the embedding layer which to transform the condition y in to embedded vector. Then embedded y will multiply with $\phi(x)$ and added back as a residual to the output. $V \cdot \phi(x)$ is equivalent to a special classification network. The output number is not mapped into a probability distribution by softmax, but it can still represent the degree to which the input data belongs to a certain class. The larger it is, the more it belongs to a certain class, and the smaller it is.

$$f^*(x, y) = y^T V \phi(x) + \psi(\phi(x)) \tag{2.1.1}$$

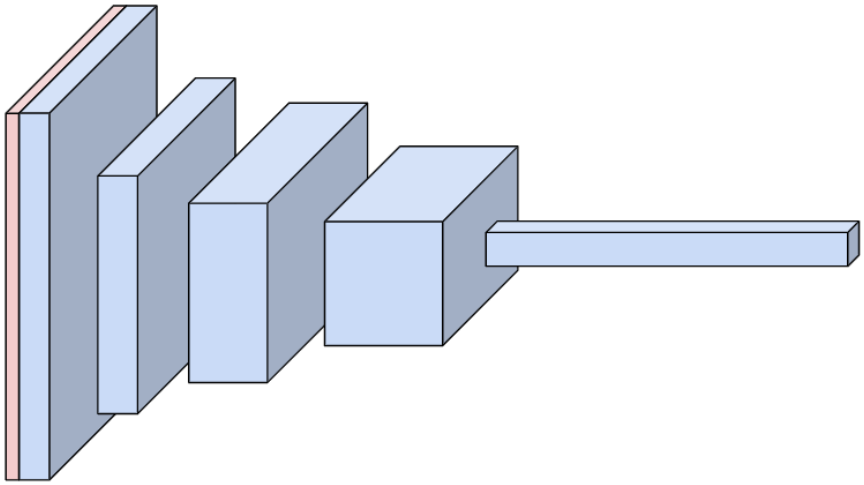


Figure 2: Illustration of the Concatenation of Discriminator.

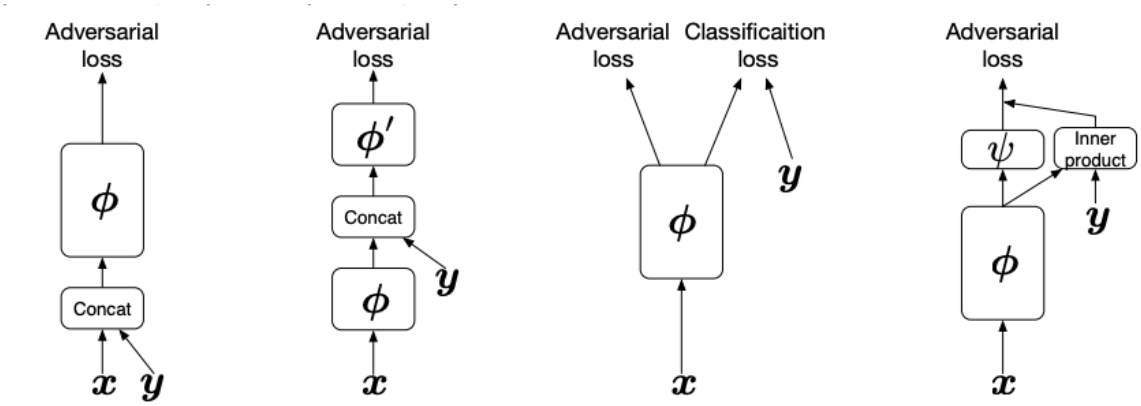


Figure 3: Rightmost is the Projection Discriminator.

```
class Discriminator(nn.Module):
    def __init__(self, args):
        super(Discriminator, self).__init__()
        self.img_size = args.img_size
        self.input_size = 3 if args.dis_mode == 'proj' else 4

        self.map = nn.Sequential(
            nn.Linear(args.num_conditions, args.img_size * args.img_size),
            nn.ReLU(inplace=True))

        self.main = nn.Sequential(
            nn.Conv2d(self.input_size, args.ndf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(args.ndf),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(args.ndf, args.ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(args.ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(args.ndf * 2, args.ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(args.ndf * 4),
```

```
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(args.ndf * 4, args.ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(args.ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(args.ndf * 8, args.ndf * 16, 4, 2, 1, bias=False),
            nn.BatchNorm2d(args.ndf * 16),
            nn.LeakyReLU(0.2, inplace=True))

        self.adv_layer = nn.Sequential(
            nn.Conv2d(args.ndf * 16, 1, 4, 2, 1, bias=False),
            nn.Sigmoid())

        self.aux_layer = nn.Sequential(
            nn.Conv2d(args.ndf * 16, args.num_conditions, 4, 2, 1, bias=False),
            nn.Sigmoid())

        self.embedded_layer = nn.Linear(args.num_conditions, args.ndf * 16)

    def forward(self, image, c):
        if self.input_size == 3:
            phi_x = self.main(image)
            out = torch.sum(self.adv_layer(phi_x), dim=(2, 3))
            labels = torch.sum(self.aux_layer(phi_x), dim=(2, 3))
            c = self.embedded_layer(c)
            out = out + torch.sum(c * torch.sum(phi_x, dim=(2, 3)), dim=1, keepdim=True)
            return out.reshape(-1), labels
        else:
            c = self.map(c).reshape(-1, 1, self.img_size, self.img_size)
            x = torch.cat((image, c), dim=1)
            phi_x = self.main(x)
            validity = self.adv_layer(phi_x).reshape(-1)
            labels = self.aux_layer(phi_x).squeeze()
            return validity, labels
```

2.2 Setup

The hyperparameters setup for baseline are as follows:

- *epochs* 300
- *batch size* 128
- *learning rate* $2e-4$
- *beta*1 0.5
- *beta*2 0.999
- *nz* 100
- *num conditions* 24
- *nc* 300
- *ngf* 128
- *ndf* 128
- *image size* 64
- *proportion* 4

3 Result

For our best result, the hyperparameters setup shows in previous section. We used conditional DCGAN structure with projection discriminator plus ACGAN's classification loss. The spectial part is that we added another convolutional layer in discriminator which means the proportion of each layer in discriminator become $1 : 2 : 4 : 8 : 16$. This movement pushed the result a little bit forward. And we had also tried learning rate scheduler during training since we observed that the loss of generator stop going down after the half of epochs. However, it seems not really work. We achieved F1-score 0.854 in test file and 0.794 in new test file while the baseline is to got over 0.8 in test file.

3.1 Generated Images

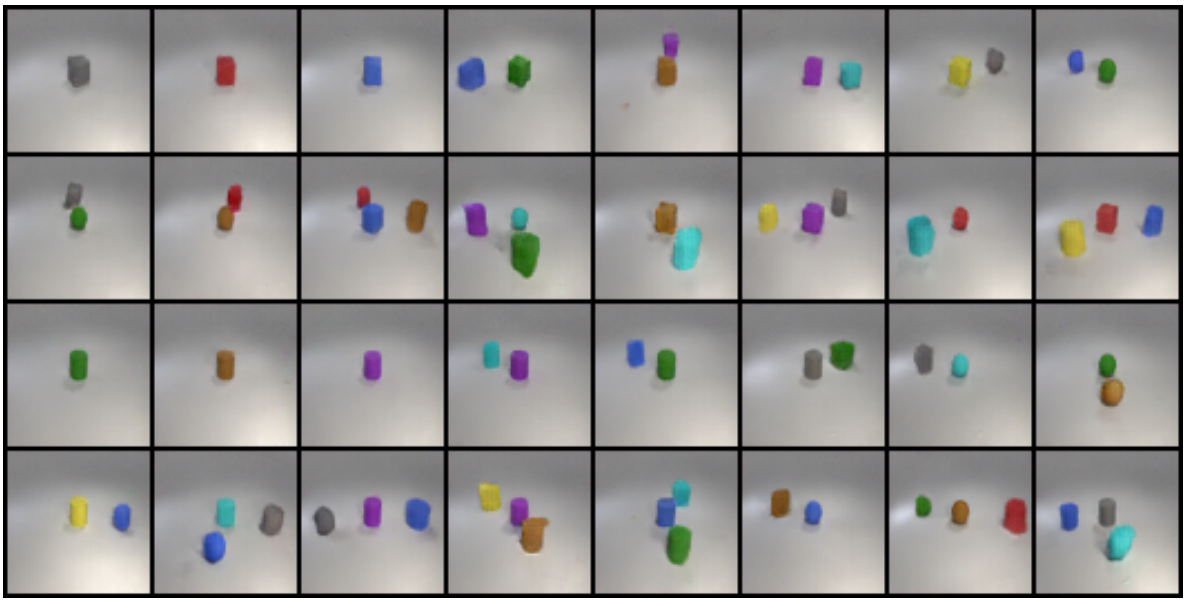
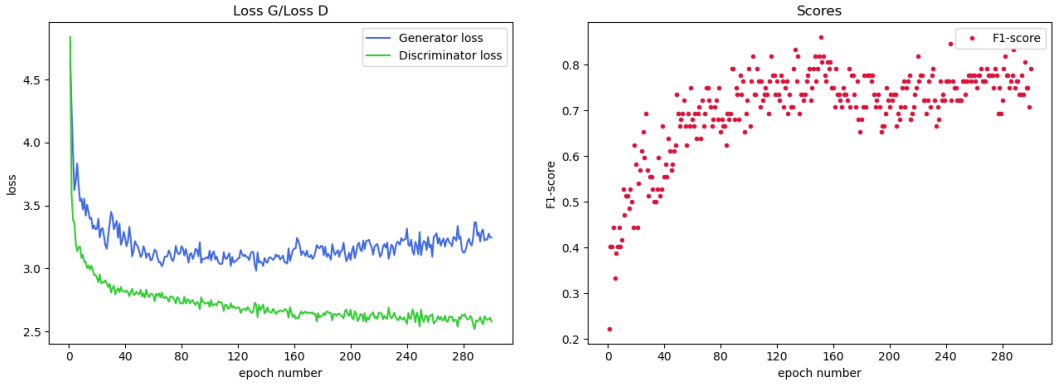


Figure 4: The synthetic images of F1-score of 0.854 in *test.json*.

Figure 5: The synthetic images of F1-score of 0.794 in *new_test.json*.

3.2 Curves



(a) Loss of Generator and Discriminator.

(b) F1-score curve.

Figure 6: Our best result's training curves.

4 Discussion

4.1 cGAN vs cDCGAN

In the first of our experiment, we used the simple cGAN which just contains few hidden layers in both generator and discriminator and all layers are linear. The score improved slowly and only got highest score 0.324 in the end. Then we turned the model to conditional DCGAN which remove fully connected hidden layers for deeper architectures. Used ReLU activation in generator for all layers except for the output, which uses Tanh and used LeakyReLU activation in the discriminator for all layers. Since the success of CNNs in image, DCGAN is really suitable for this task.

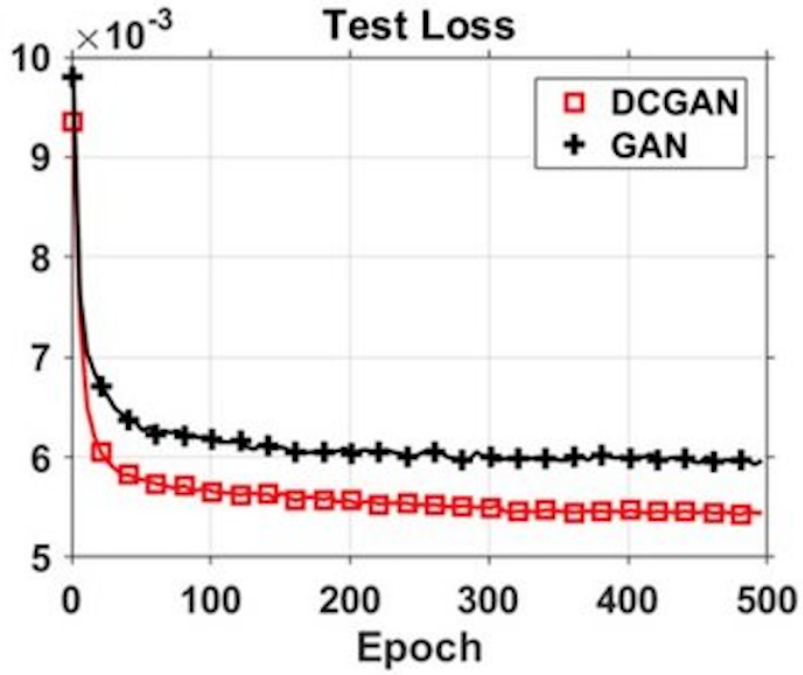
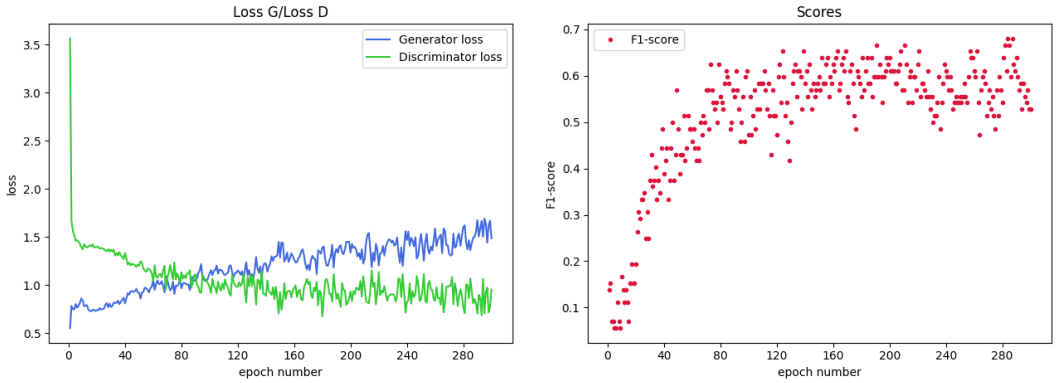


Figure 7: Illustration of testing loss between cGAN and cDCGAN.

4.2 Concatenation vs Projection

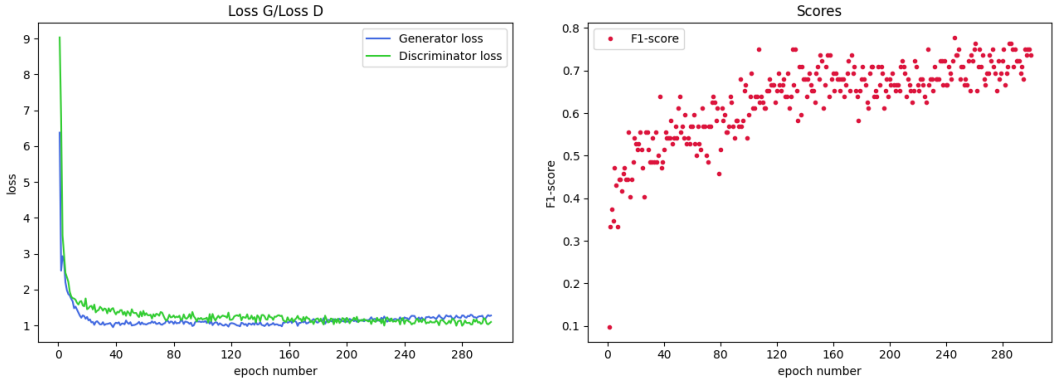
Since we had mentioned in Sec.2.1.2, there are two type of discriminator. One is to concat sample z and conditions directly as input to discriminator. Another is calculating the inner product of embedded conditions and $\phi(x)$. In our experiment, the second one that is called *projection discriminator* improved the learning performance a lot. As Fig.8 and 9, with projection discriminator, the score can easily achieved over 0.7 just using less then half of training time, but concatenation discriminator can not reach the same goal even the training is finished and the loss curves are shaking heavily.



(a) Loss of Generator and Discriminator.

(b) F1-score curve.

Figure 8: Concatenation discriminator.



(a) Loss of Generator and Discriminator.

(b) F1-score curve.

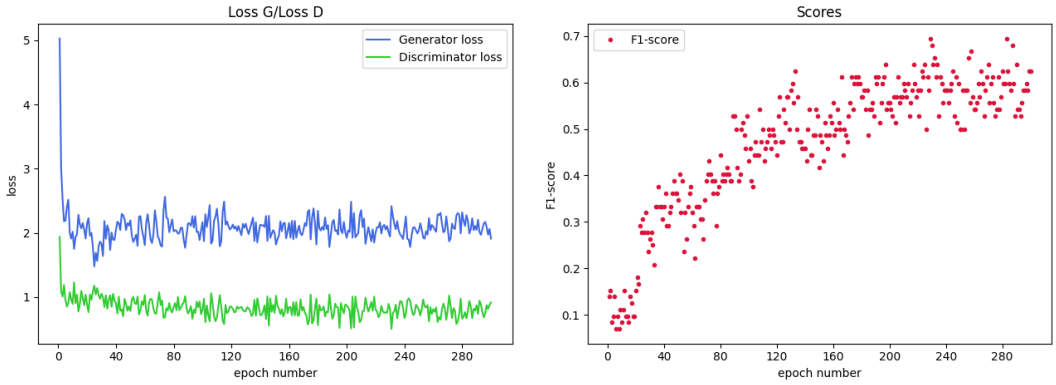
Figure 9: Projection discriminator.

4.3 Dimension of Condition Vector

The hyperparameter nc is the dimension of conditions which we used to concat with sample z as input to generator and discriminator. The reason to do this is that our condition is only 24, and by doing so, we hope that the information of condition will expand and model will learn more. In the first of our experiment, we set the value to 100 which is referenced by DCGAN implementation on Pytorch website. However, the score stucked around 0.65, sometimes will reach 0.7. Then we kept increasing the value to 200 and 300 to see if the factor will affect the model, in our surprise, the score can achieved around 0.7 to 0.8 stably. After successful observation, we set the value to 500 but it seems not that useful. So for our experience in condition vector, the best setting will be 200 to 300.

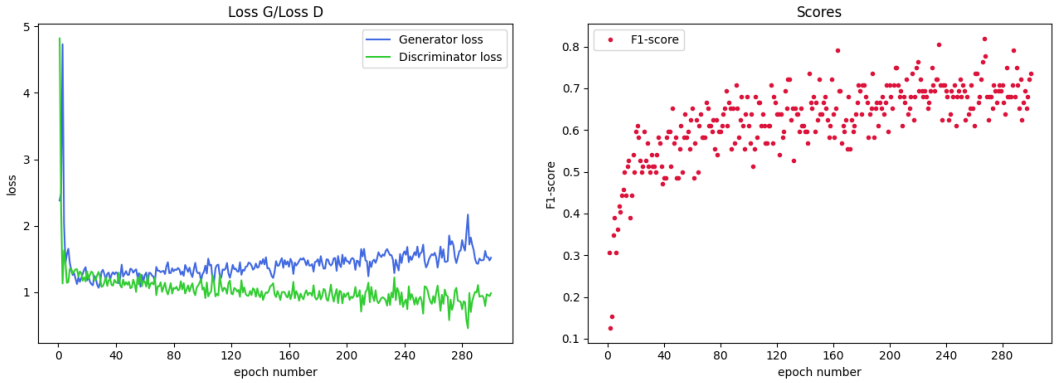
4.4 Proportion of G/D's Iterations

This is another important trick for training GAN since our goal is to minimize the loss of both generator and discriminator. And also, we hope that the loss between them can more closer. So we set the proportion of iteration in generator and discriminator to $4:1$ because we found out the generator's loss are always bigger than the one in discriminator. By doing this trick, our loss decreasing extremely at the first of training and can reach over 0.5 just using 5 epoch. And also thanks to the trick, we can achieved our best score which is over 0.8 in final.



(a) Loss of Generator and Discriminator. (b) F1-score curve.

Figure 10: Same number of iteration in G/D.



(a) Loss of Generator and Discriminator. (b) F1-score curve.

Figure 11: Proportion of $4:1$ iteration in G/D.

4.5 Single-loss vs Multi-loss

In the beginning of our training, we used DCGAN with projection discriminator as our main model and had already beat the baseline. Since we found out that ACGAN [3] used another classifier to robust the training, we rethought that if we could combined this idea into our work. Then, we rebuilt our discriminator as Fig.12. Compare with the original one, our new discriminator had two outputs which are validity and labels. We added auxiliary layer to classify the image into 24 classes. And because it is a multi label problem, so we used sigmoid as activation function in auxiliary layer and calculated the loss with cross entropy loss. Therefore, generator loss and discriminator loss became not just adversarial loss but also include classification loss. In our experiment, by adding classification loss, the performance improved a lots and pushed our best result to the next level. The comparison between single and multi-loss exhibits in Table.1.

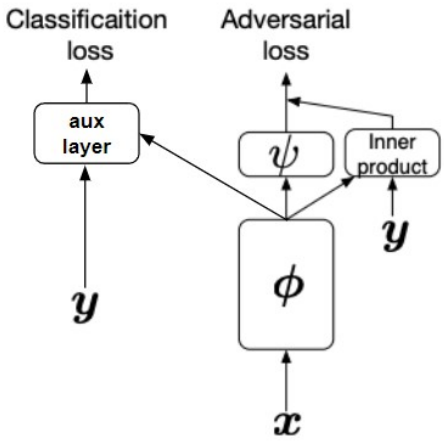


Figure 12: Illustration of our novel discriminator.

	Loss	Score (*test*)	Score (*new test*)
single-loss	adv	0.802	0.644
multi-loss	adv+aux	**0.854**	**0.794**

Table 1: Performance Comparison in single-loss and multi-loss.

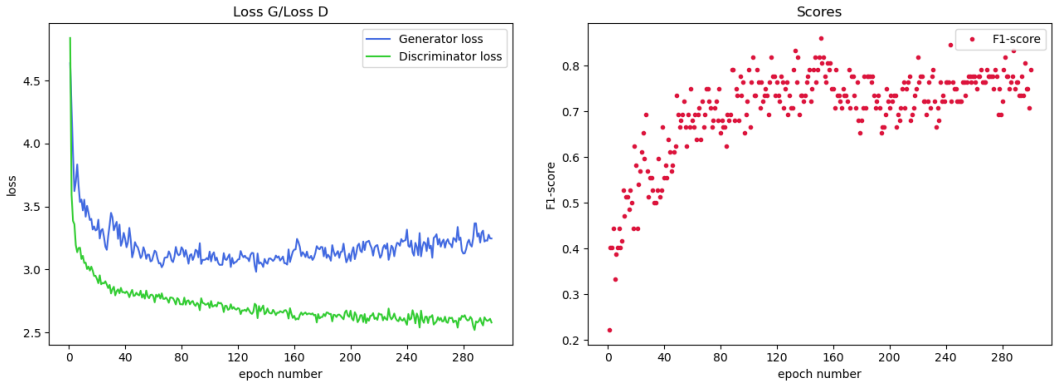
References

[1] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[2] Takeru Miyato and Masanori Koyama. cgans with projection discriminator. *arXiv preprint arXiv:1802.05637*, 2018.

[3] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans. In *International conference on machine learning*, pages 2642–2651. PMLR, 2017.

[4] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.



(a) Loss of Generator and Discriminator.

(b) F1-score curve.

Figure 13: After adding classification loss in original DCGAN.