Lab5 Report

0856091 蘇邱弘

August 2020

1 Introduction

GANs (Generative Adversarial Network) are a type of generative models: they create new data instances that resemble our training data. Auxiliary classifier GANs (ACGANs) are one kind of conditional GANs that can generate data based on the given condition. In this lab, I implement an ACGAN to do the image generation task on i-CLEVR datasets.

In this report, I will first introduce the implementation details of my ACGAN model and the corresponding loss function I used. Next, I will show my testing results of image generation. Finally, I will do some discussions.

2 Implementation Details

2.1 GAN Architecture

Figure 1 shows the general architecture for ACGAN. In this lab, I implement the ACGAN using Pytorch, and the implementation details are as follows.

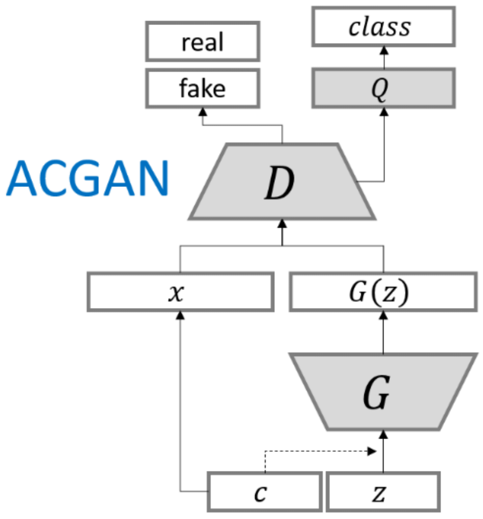
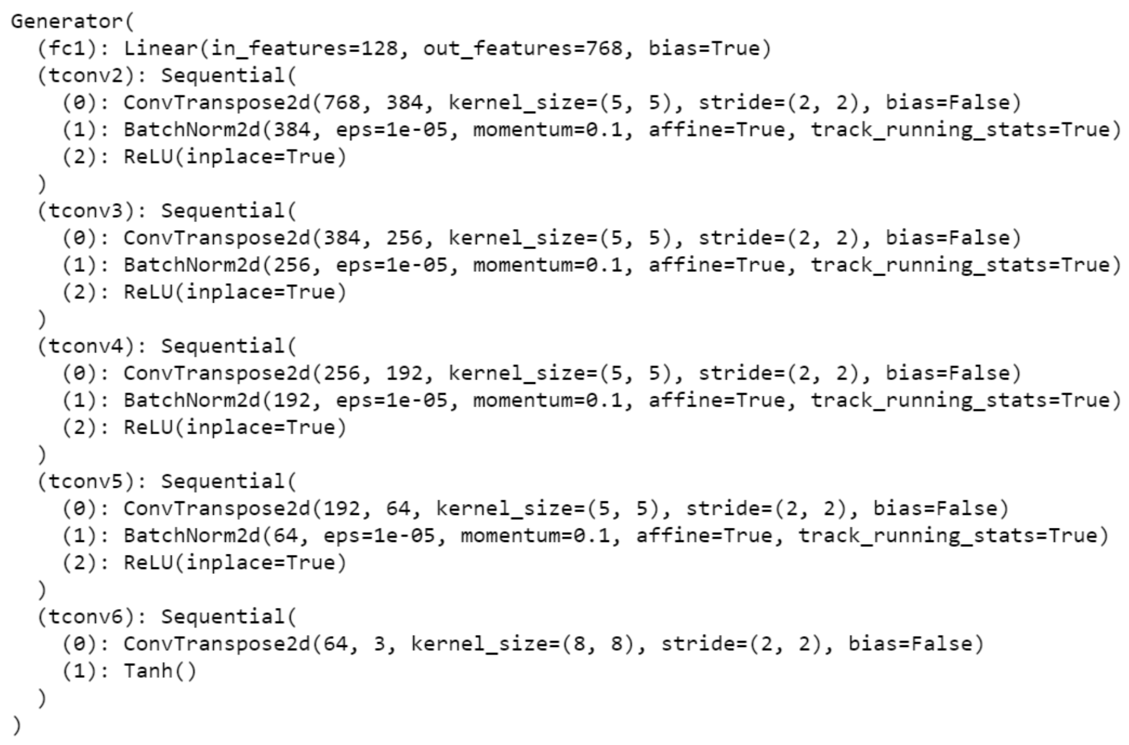


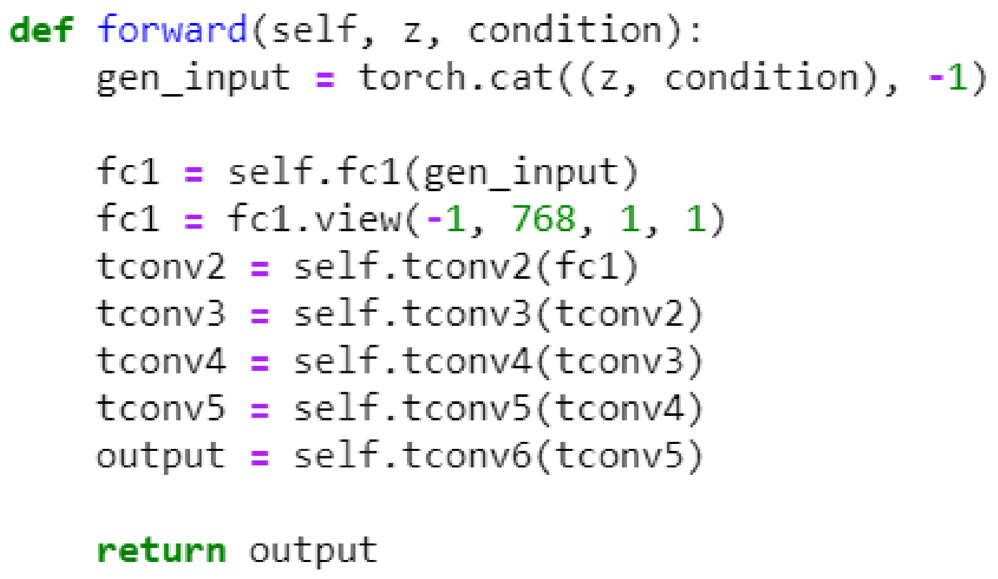
Figure 1: Architecture for ACGAN.

For the generator, the architecture is shown in Figure 2 and it's forward function is shown in Figure 3. As shown in these figures, the generator's inputs are a noise sampled from normal distribution and a 24-dimension one-hot-encoded label, and the output is a fake image belonging to the input class label.



```
Generator(
  (fc1): Linear(in_features=128, out_features=768, bias=True)
  (tconv2): Sequential(
    (0): ConvTranspose2d(768, 384, kernel_size=(5, 5), stride=(2, 2), bias=False)
    (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (tconv3): Sequential(
    (0): ConvTranspose2d(384, 256, kernel_size=(5, 5), stride=(2, 2), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (tconv4): Sequential(
    (0): ConvTranspose2d(256, 192, kernel_size=(5, 5), stride=(2, 2), bias=False)
    (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (tconv5): Sequential(
    (0): ConvTranspose2d(192, 64, kernel_size=(5, 5), stride=(2, 2), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (tconv6): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(8, 8), stride=(2, 2), bias=False)
    (1): Tanh()
  )
)
```

Figure 2: Architecture for the generator.



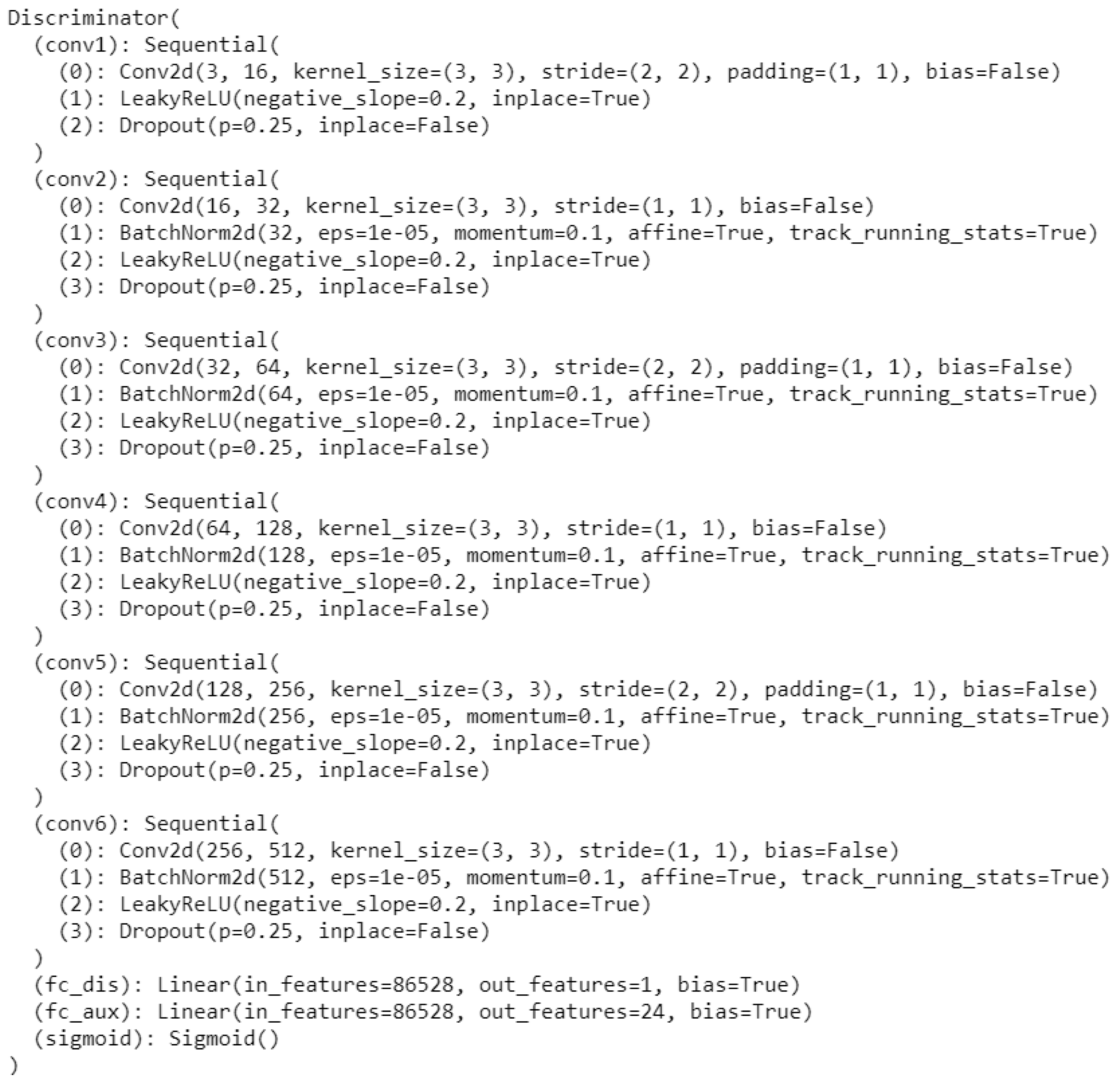
```python
def forward(self, z, condition):
    gen_input = torch.cat((z, condition), -1)

    fc1 = self.fc1(gen_input)
    fc1 = fc1.view(-1, 768, 1, 1)
    tconv2 = self.tconv2(fc1)
    tconv3 = self.tconv3(tconv2)
    tconv4 = self.tconv4(tconv3)
    tconv5 = self.tconv5(tconv4)
    output = self.tconv6(tconv5)

    return output
```

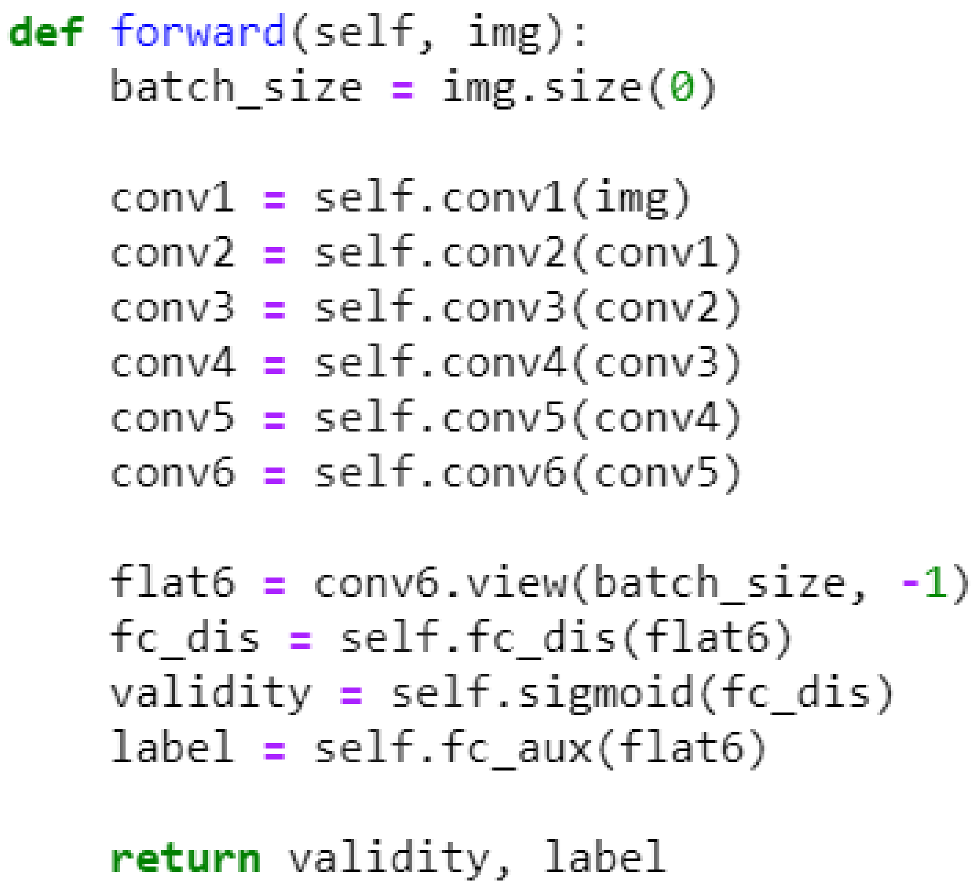
Figure 3: Generator's forward function.

For the discriminator, the architecture is shown in Figure 4 and it's forward function is shown in Figure 5. As shown in these figures, the discriminator took an image (whose size is 128x128) as input, then go through 6 convolutional layers, and use one fully connected layer (fc_dis) to get the probability that the image is real, at the same time use another fully connected layer (fc_aux) to get the class label.



```
Discriminator(
  (conv1): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Dropout(p=0.25, inplace=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Dropout(p=0.25, inplace=False)
  )
  (conv3): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Dropout(p=0.25, inplace=False)
  )
  (conv4): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Dropout(p=0.25, inplace=False)
  )
  (conv5): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Dropout(p=0.25, inplace=False)
  )
  (conv6): Sequential(
    (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Dropout(p=0.25, inplace=False)
  )
  (fc_dis): Linear(in_features=86528, out_features=1, bias=True)
  (fc_aux): Linear(in_features=86528, out_features=24, bias=True)
  (sigmoid): Sigmoid()
)
```

Figure 4: Architecture for the discriminator.



```python
def forward(self, img):
    batch_size = img.size(0)

    conv1 = self.conv1(img)
    conv2 = self.conv2(conv1)
    conv3 = self.conv3(conv2)
    conv4 = self.conv4(conv3)
    conv5 = self.conv5(conv4)
    conv6 = self.conv6(conv5)

    flat6 = conv6.view(batch_size, -1)
    fc_dis = self.fc_dis(flat6)
    validity = self.sigmoid(fc_dis)
    label = self.fc_aux(flat6)

    return validity, label
```

Figure 5: Discriminator's forward function.

2.2 Loss Function

For the discriminator, because it will generate two outputs, so I use two losses to evaluate them respectively. For the probability that the image is real, I use binary cross entropy loss between the output and the ground truth, and define it as $adversarial\_loss$. For the class label, I also use binary cross entropy loss between the output and the ground truth, and define it as $classification\_loss$. In the following sections, I will describe how I use these two losses to update the generator and the discriminator respectively.

2.2.1 Discriminator Loss

In the training process, I first fix the generator, and evaluate the discriminator using the real images and the fake images generated by the generator. In order to update it, I calculate total loss for the discriminator D_loss as follows:

I first compute the loss for the real images:

$$d\_loss\_real = adversarial\_loss + (classification\_loss * 48)$$

Then compute the loss for the fake images:

$$d\_loss\_fake = adversarial\_loss$$

Finaly obtain total loss for the discriminator:

$$D\_loss = d\_loss\_real + d\_loss\_fake$$

The idea is, when we evaluate the discriminator with fake images, we don't have to consider whether they have been classified correctly or not. But for the real images, we have to consider it, and we have to give much large weight to the $classification\_loss$, because I found that it is harder for a vector (one-hot-encoded label) to get the same loss as a scalar when using binary cross entropy loss.

2.2.2 Generator Loss

After updating the discriminator, I fix the discriminator, and use generator to generate images, then input them into discriminator to calculate the loss for the generator G_loss as follows:

$$G\_loss = adversarial\_loss + (classification\_loss * 24)$$

The idea is the same as above, and notice that in order to keep the proportion of total $adversarial\_loss$ to total $classification\_loss$ about 1 to 24 in both D_loss and G_loss, I use 24 (half of $classification\_loss$ weight in d_loss_real) as the $classification\_loss$ weight.

In my implementation, in fact I use a linear schedule function to gradually increase the weight of $classification\_loss$ to 24 in the first 1000 iterations. It is because I want to let generator focus on learning to generate real image at first, then consider the classification part after it can generate images that is real enough.

2.3 Hyperparameters

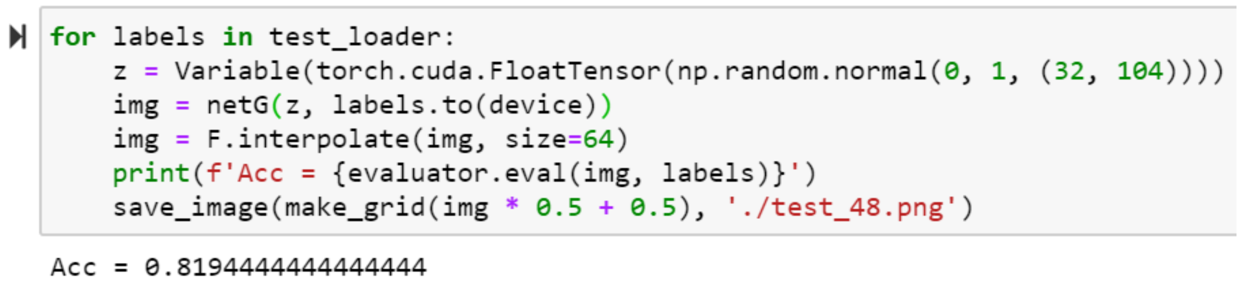
The followings are the hyper-parameters I used:

- batch size: 128
- epochs: 400
- total iterations: 56400
- learning rate: 0.0002 for both generator and discriminator
- optimizer: Adam
- noise size: 104

3 Results

3.1 Results of Image Generation

Figure 6 shows the highest testing accuracy evaluated by the evaluator provided by TA, and Figure 7 shows the corresponding image. The generator is load from the model saved in the 30127-th iteration during training, and the image generated from it can achieve accuracy of 0.819.



```
for labels in test_loader:
    z = Variable(torch.cuda.FloatTensor(np.random.normal(0, 1, (32, 104))))
    img = netG(z, labels.to(device))
    img = F.interpolate(img, size=64)
    print(f'Acc = {evaluator.eval(img, labels)}')
    save_image(make_grid(img * 0.5 + 0.5), './test_48.png')
```

```
Acc = 0.8194444444444444
```

Figure 6: Highest accuracy.

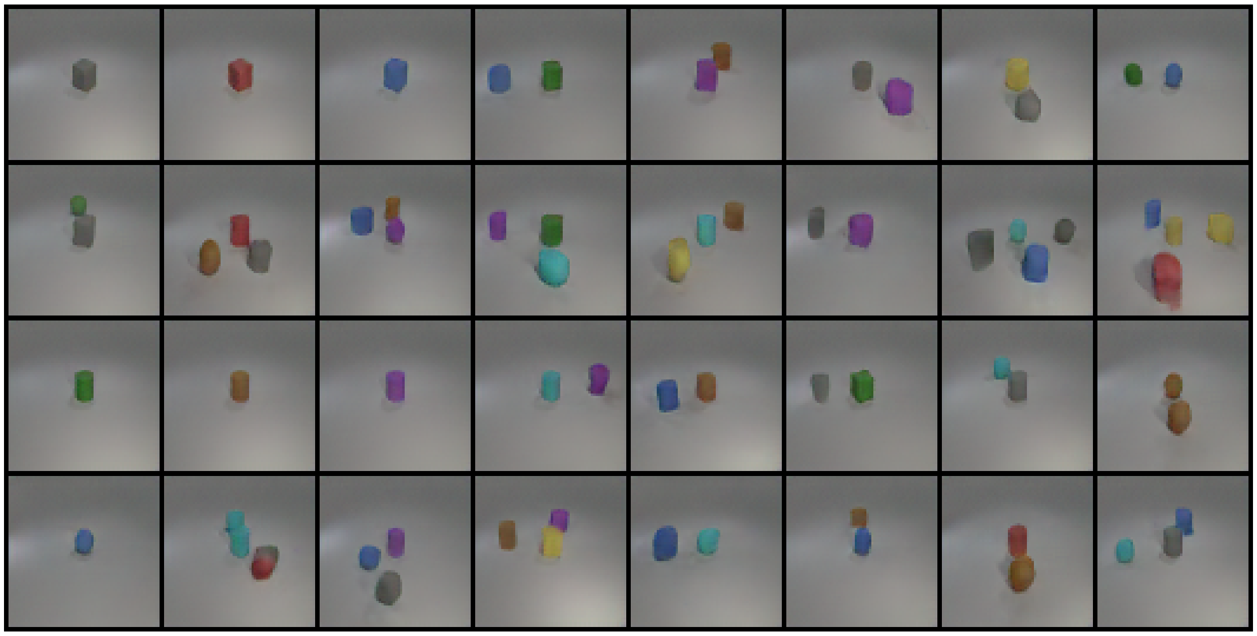


Figure 7: Generated image.

3.2 Training Trends

Figure 8 and 9 show the testing accuracy trend and the loss trend during training (with fixed noise). The highest testing accuracy (0.819) appeared in the 30127-th iteration.

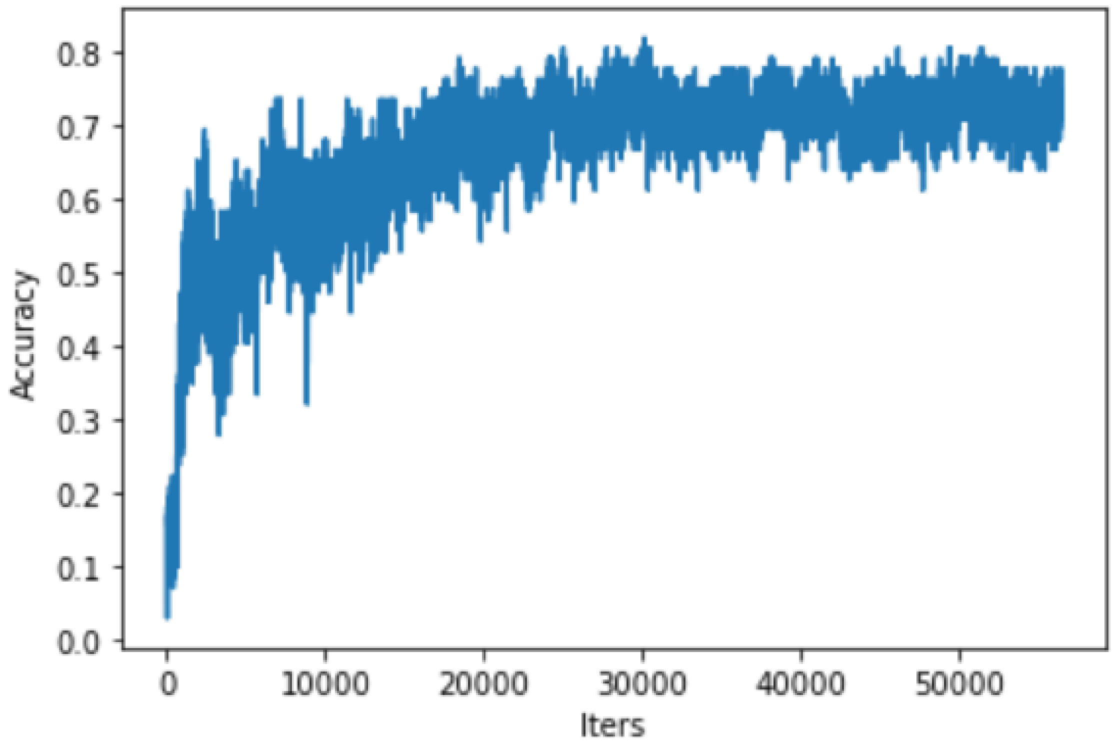


Figure 8: Testing accuracy trend during training.

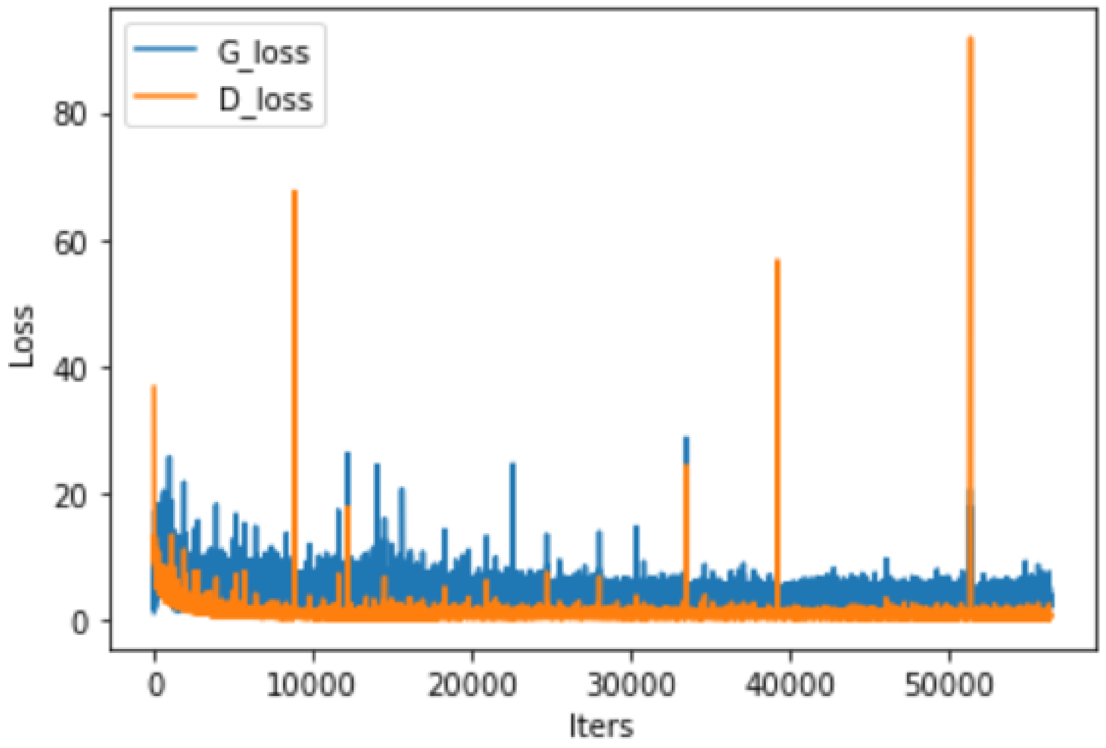


Figure 9: Loss trend during training.

4 Discussion

4.1 Weight of Classification Loss

During the implementation of the training process, I found that if the weights between $adversarial\_loss$ and $classification\_loss$ are the same, it can only generate images that look like real images, but these images can hardly match the labels which were input to the generator. As describe in section 2.2, it is because that for the discriminator, it is harder to reduce $classification\_loss$ with binary cross entropy loss. In order to fix this issue, I apply a larger weight to $classification\_loss$ to insure that the discriminator actually learn how to do the classification.

And in this lab, I tried different $classification\_loss$ weights in d_loss_real (defined in section 2.2.1) during the implementation (and the corresponding weight in G_loss would be half of it). As shown in the figure, with larger weight, the testing accuracy converge faster, and get a better performance.

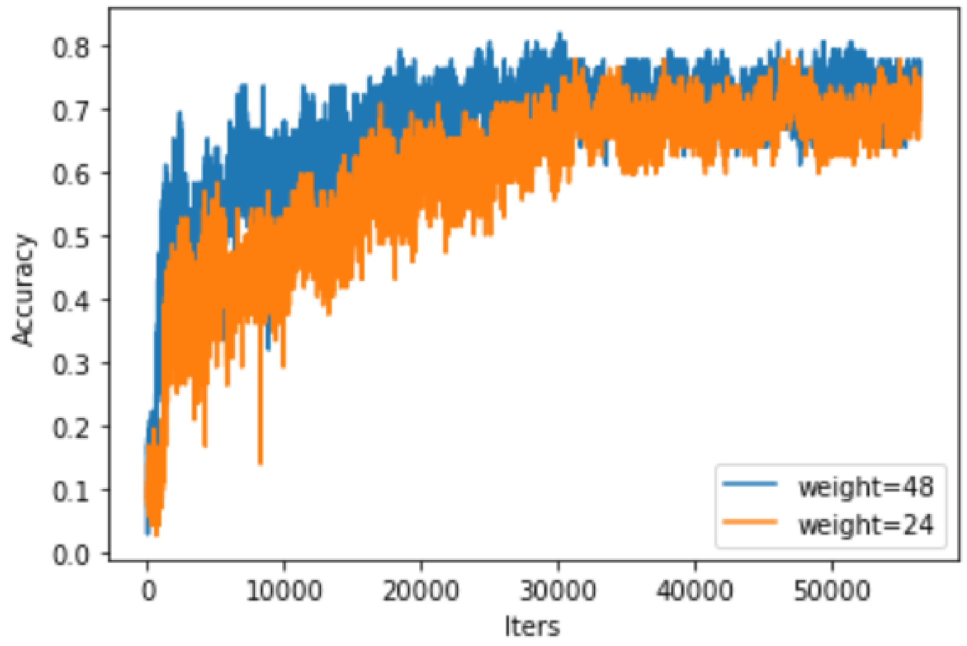


Figure 10: Accuracy comparison between different $classification\_loss$ weights in d_loss_real.