Technical University of Denmark

# Generative Adversial Networks (GAN)

Deep learning in Computer Vision

Maiken Errebo Lindberg, s114012
Jules Belveze, s182291

June 26, 2019

# Contents

# 1 MNIST

## 1.1 Introduction

This project aims to generate MNIST digits using generative adversarial networks (GAN). To do so, we first implement a vanilla GAN and then a deep convolutional generative adversarial networks (DCGAN [1]) with different loss functions.

## 1.2 Methods

### 1.2.1 Architecture networks

Generative adversarial networks are composed of two neural networks. The generator, that aims to generate new data instances and the discriminator that tries to guess whether or not an instance belongs to the training set. We first implemented a fully connected version of both before converting it to a DCGAN.
Our generator is composed of four hidden layers of 4096 neurons. Each of them is followed by a *LeakyReLU* activation function, in order to avoid dead neurons, and a batch normalization layer to speed up training. The output is then generated using a hyperbolic tangent, so that all the pixels lay in $[-1, 1]$. The discriminator is a stacking of three fully connected layers, with 1024, 512 and 256 neurons respectively. Here we again use a *LeakyReLU* activation function after a dropout layer. The output is then converted to a probability using the sigmoid function for activating the last layer.
We then convert our fully connected networks to convolutional ones to obtain a DCGAN. To do so, we use deconvolutional layers for our generator in order to output a $3 * 28 * 28$ image. In opposition, the discriminator uses convolutional layers to compute features to be able to output the probability for the input to belong to the training set.

### 1.2.2 Training process

In order to train our GAN and DCGAN models we have to train both generator and discriminator simultaneously. The underlying idea is that the generator takes noise as input and returns an image. This generated image is fed into the discriminator alongside a stream of images from the training dataset. The discriminator returns a probability for each image reflecting its authenticity. The generator is trying its best to fool the discriminator, whereas the last one aims to be as accurate as possible in its prediction.

## 1.3 Results

Different generative adversarial networks and loss function were tested on the handwritten MNIST data. First a GAN with a vanilla loss function were implemented. The generator of GANs ability to generate fake digits based on training on the handwritten MNIST data set and

its discriminator's ability to discriminate between the fake and real images of digits are shown in Figure 1. By visual inspection of the generated digits, it is seen that the fake digits somewhat resemble the real digits, but the edges of the digits are not very clear. The last plot shows the two distributions for fake and real digits. Hereof, it is clear that these two distributions overlap each other around a probability of 0.5. This means that the discriminator is really bad at discriminating between fake and real digits, since it predicts either of them with a probability of 0.5.

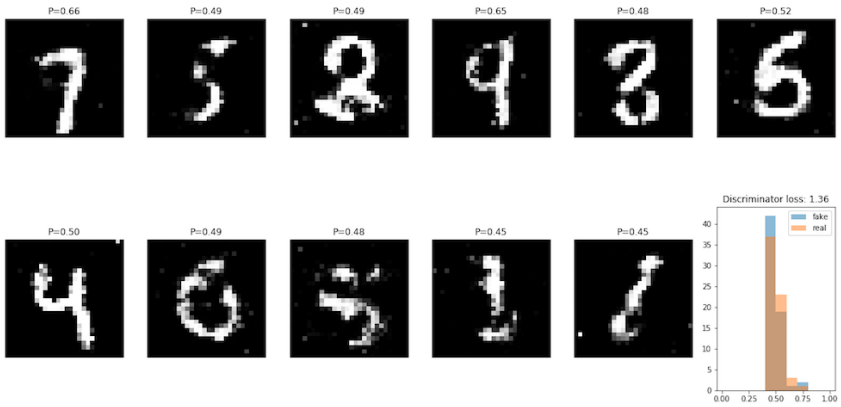


**Figure 1:** Fake generated digits using a generative adversarial network (GAN) with a vanilla loss function trained on the handwritten MNIST data.

Next, a DCGAN with a vanilla loss function is implemented to check whether it improves performance. The results of the DCGAN with a vanilla loss function are shown in Figure 2. By visual inspection of the fake digits, it is clear that the generator is really good at generating fake digits with fine edges such that the fake digits resemble real digits. For the sake of the discriminator it is evident from the two distributions for fake and real digits that these two are well-separated, where the discriminator mostly discriminates real digits with a probability of 1 or close hereto as well as discriminate fake digits with a probability close to 0. Comparing the GAN and DCGAN reveals that the DCGAN is thus performing better at both generating fake digits and discriminating between real and fake digits than the simple GAN model.

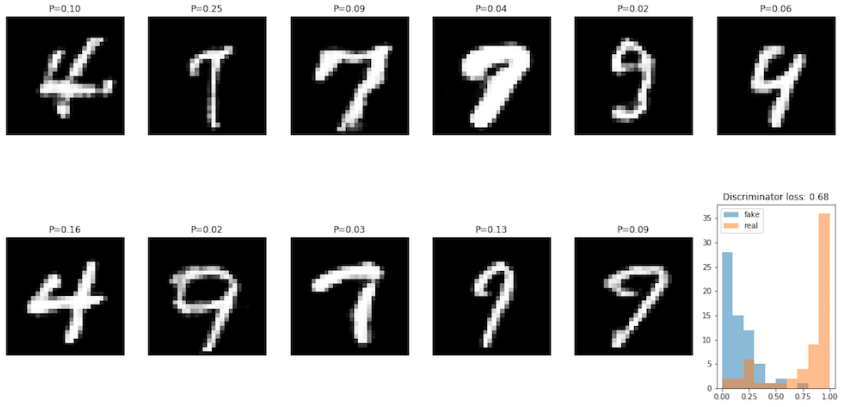


**Figure 2:** Fake generated digits using a deep convolutional generative adversarial network (DCGAN) with a vanilla loss function trained on the handwritten MNIST data.

Then the DCGAN is used together with a least squares lost function instead of a vanilla loss to be able to compare the performance of different loss functions. In Figure 3 the result of the model with a DCGAN and a least squares loss function are shown. By investigation of the fake generated digits, it is clear that the generator is able to generate digits that resemble real digits with quite the same performance as the DCGAN using a vanilla loss function. As for the discriminator, it is evident from the two distributions that they overlap a bit more than the DCGAN combined with a vanilla loss function, which means that the discriminator sometimes have a harder time discriminating between whether a digit is real or generated when using a least squares loss function rather than a vanilla loss function.

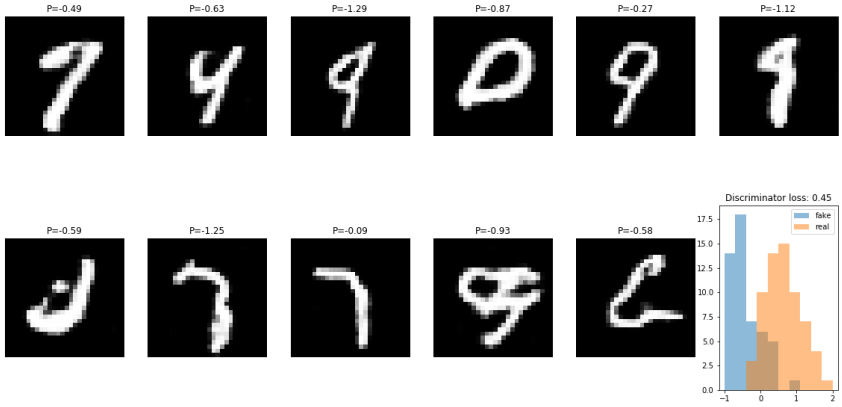


**Figure 3:** Fake generated digits using a deep convolutional generative adversarial network (DCGAN) with a least squares loss function trained on the handwritten MNIST data.

# 2 CycleGAN

## 2.1 Introduction

The aim of this part of the project is to convert images of horses into images of zebras and vice versa by using a CycleGAN.

## 2.2 Methods

In order to implement this CycleGAN we used the suggested networks architecture and hyper-parameters value. To train our model we had to train four different networks: a discriminator and a generator for each image category. Then our goal is to learn mappings $G_{A\to B} : A \to B$ and $G_{B\to A} : B \to A$ such that the mapped image will be indistinguishable from the output distribution. During the training we are trying to minimize a combined loss for our generators and the loss of each discriminator. The generators' loss is the following:

$$\text{Loss} = (\text{LossIdentityA} + \text{LossIdentityB}) * 0.5 + (\text{LossA2B} + \text{LossB2A}) + (\text{LossCycleA} + \text{LossCycleB}) * 10 \quad (1)$$

The first parenthesis is evaluating how well the generators perform when the input lies on the output space. The second one depicts how well the generators fool the discriminators. Whereas the last one assess the performance of a cycle. That is mapping an image to the other space and mapping this last one back to the original space.

## 2.3 Results

### 2.3.1 Res-Net as generator

First the model described in Section 2.2 was implemented and trained for 30 epochs and the results from the generators, when converting horse to zebra and vice versa are shown in Figure 4 and 5. As seen on Figure 4 the generated zebra from horse has obtained stripes.

**Figure 4:** Left: Original image of horse. Right: Image of horse converted to zebra, when using the Res-net as generator in the CycleGAN.

By visual inspection of Figure 5, the image to the right depicting a zebra converted to a horse has obtained a brown color and the stripes are also much less visible.

**Figure 5:** Left: Original image of zebra. Right: Image of zebra converted to horse, when using the Res-net as generator in the CycleGAN.

Figure 6 is representing the evolution of the contribution to the total of each term in Eq. 1 through epochs. We cannot identify any real pattern. However, as expected the cycle losses contribute more to the final loss than the others, because of a bigger penalty.

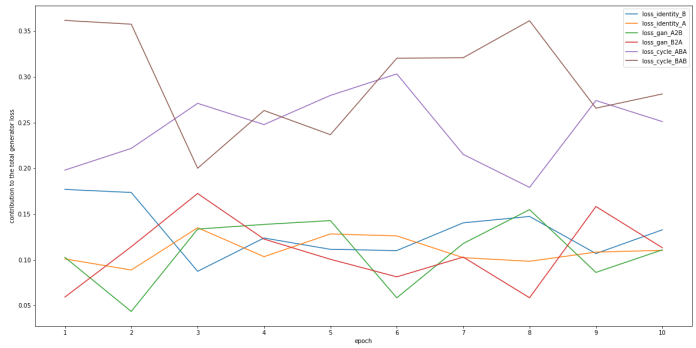


**Figure 6:** Evolution of the each loss contribution to the total loss in Eq. 1, when using the Res-net as generator in the CycleGAN.

Figure 7 reflects the evolution of the generators and discriminators losses throughout epochs. It looks like the generators losses are not really decreasing. Whereas the generators losses are bigger and fluctuating. This can be explained by the fact that the discriminator is not performing well. Thus, the generators are producing trash without really trying to get better. One way to solve this would be training the discriminator more per generator iteration. However, by looking at the generated images, it seems that the generators are performing quite well. To observe a real tendency, we might have to observe losses for a higher number of epochs.

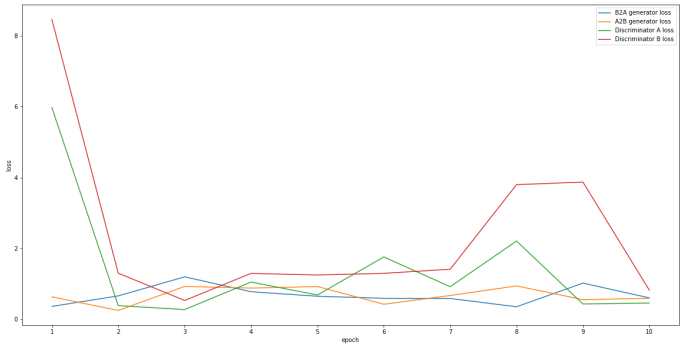


**Figure 7:** Evolution of the generator losses versus discriminator losses, when using the Res-net as generator in the CycleGAN.

### 2.3.2 Res-Net as generator with data augmentation

Next, the CycleGAN using the Res-Net as generator was trained with data augmentation applied on the zebra and horse training data. The following augmentation was applied using Pytorch on resized images of $128 * 128$ instead of $256 * 256$:

```
transforms.RandomAffine(rotation=(-30, 30), translate=(0.1, 0.3), scale=(0.8, 1.2))
```

The results for conversion of horse to zebra and zebra to horse after training for 30 epochs are shown in Figure 8 and 9, respectively.

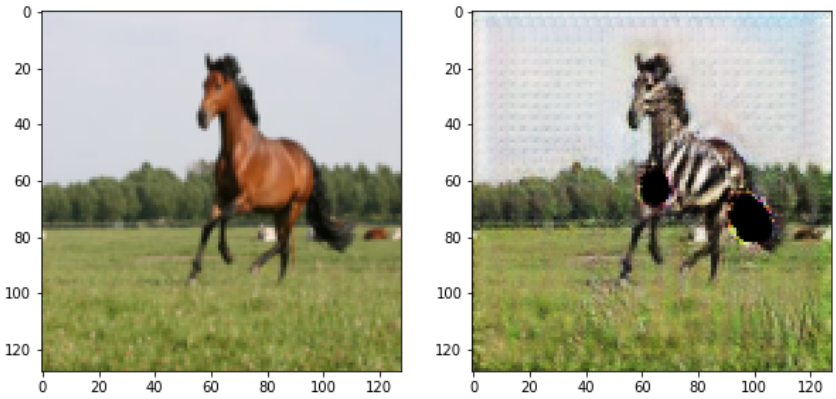

**Figure 8:** Left: Original image of horse. Right: Image of horse converted to zebra, when using the Res-net as generator and data augmentation in the CycleGAN.

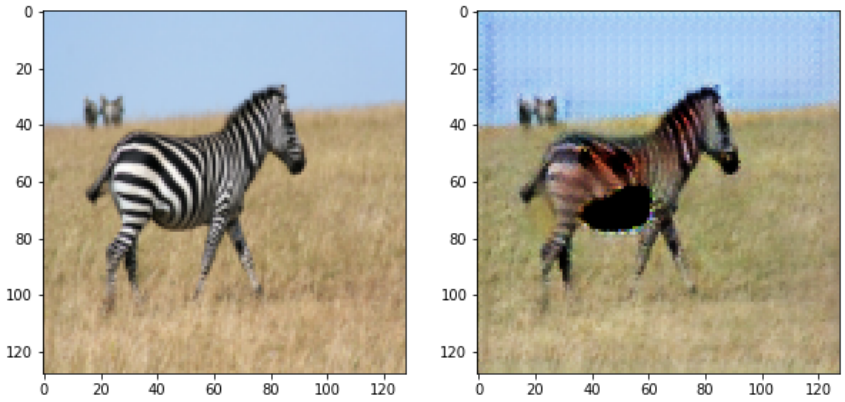

**Figure 9:** Left: Original image of zebra. Right: Image of zebra converted to horse, when using the Res-net as generator and data augmentation in the CycleGAN.

By visual inspection of Figure 8 and 9 it is clear that the generated zebra from real horse has gained very clear zebra stripes, while the generated horse from real zebra has obtained a brown coloring, respectively. Comparing Figure 8 and 4, it is clear that the generated zebra from horse is even more convincing for the CycleGAN with data augmentation than without. Moreover, comparison of these results with data augmentation with those without data augmentation in Figure 4 and 5 reveal that the background gets a bit distorted, when applying data augmentation during training of the CycleGAN.

### 2.3.3 U-Net as generator

Next the same model as described above in Section 2.2 is used to convert zebras to horses and vice versa, but with a modified generator, to investigate how this effect the performance of the CycleGAN. We then try using U-Net as generator rather than the Res-Net. Figure 10 and 11 depict the performance of our generators trained on 10 epochs.

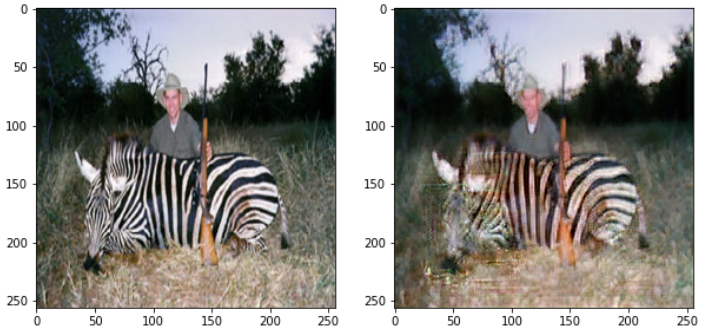

**Figure 10:** Left: Original image of zebra. Right: Conversion of zebra to horse, when using the U-net as generator in the CycleGAN.

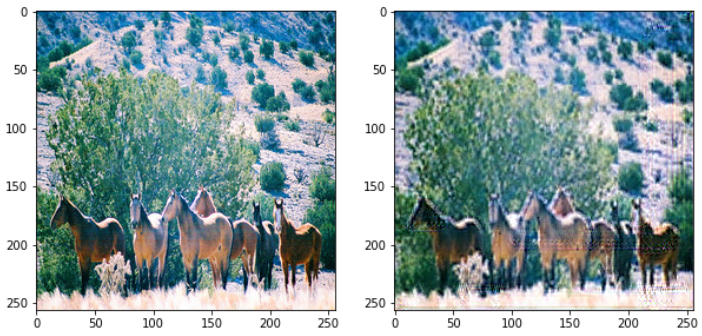


**Figure 11:** Left: Original image of horse. Right: Conversion of horse to zebra, when using the U-net as generator in the CycleGAN.

As we can observe and according to [2], the U-net as a generator is not good at extracting the underlying image features, which explains that the output images are almost the same as the input image. We can see that the generated horse from zebra in Figure 10 is slightly brown. However, in Figure 11 we cannot identify any modification. This might also comes from our short training, only 10 epochs, i.e. the network needs a lot more epochs for training to get descent results.

### 2.4 Conclusion

As seen in this report it is really difficult to train a CycleGAN and it needs a lot of time to be properly trained. Even if the evolution of the different losses appear messy and it is difficult to come up with an analysis we manage to get some descent results. Figure 12 shows one of the best result we were able to obtain. This was achieved after training the model for 50 epochs.

**Figure 12:** Lft: Original image of horse. Right: Best conversion of horse to zebra, when using the ResNet as generator in the CycleGAN.

However, with more time we will be able to train it for more epochs and try to tune the different hyperparameters of the model.

## References

[1] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," 2016.

[2] X. Jin, Y. Qi, and S. Wu, "Cyclegan face-off," Assessed: June 26, 2019. [Online]. Available: https://arxiv.org/pdf/1712.03451.pdf

# A Python code

## A.1 ResNet

```
class ResNetBlock(nn.Module):
    def __init__(self, n_features):
        super(ResNetBlock, self).__init__()
        self.resblock = nn.Sequential(
            nn.Conv2d(n_features,n_features, kernel_size=3, stride=1, padding=1),
            nn.InstanceNorm2d(n_features),
            nn.ReLU(inplace=True),
            nn.Conv2d(n_features,n_features, kernel_size=3, stride=1, padding=1),
            nn.InstanceNorm2d(n_features)
        )

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

## A.2 Generator

```
class CGAN_G(nn.Module):
    def __init__(self):
        super(CGAN_G, self).__init__()

        # encoder
        self.hidden0 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=1, padding=3),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.hidden1 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.hidden2 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True)
        )

        # transformer
        num_res_blocks = 9
        conv_layers = [nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)]
        for i in range(num_res_blocks):
            conv_layers.append(ResNetBlock(256))
        self.res_blocks = nn.Sequential(*conv_layers)

        # decoder
        self.hidden3 = nn.Sequential(
            nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.hidden4 = nn.Sequential(
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.hidden5 = nn.Sequential(
            nn.ConvTranspose2d(64, 3, kernel_size=7, stride=1, padding=3), #padding=1?
            nn.Tanh()
        )

    def forward(self, x):
        x = self.hidden0(x)
        x = self.hidden1(x)
        x = self.hidden2(x)
        x = self.res_blocks(x)
        x = self.hidden3(x)
        x = self.hidden4(x)
        x = self.hidden5(x)
        return x
```

## A.3 Discriminator

```
class CGAN_D(nn.Module):
    def __init__(self):
        super(CGAN_D, self).__init__()

        self.hidden0 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(64),
            nn.LeakyReLU(.2, inplace=True)
        )
        self.hidden1 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(128),
            nn.LeakyReLU(.2, inplace=True),
        )
        self.hidden2 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(256),
            nn.LeakyReLU(.2, inplace=True)
        )
        self.hidden3 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=4, stride=1, padding=1),
            nn.InstanceNorm2d(512),
            nn.LeakyReLU(.2, inplace=True),
        )
        self.out = nn.Sequential(
            nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=1)
        )

    def forward(self, x):
        x = self.hidden0(x)
        x = self.hidden1(x)
        x = self.hidden2(x)
        x = self.hidden3(x)
        x = self.out(x)
        return x
```

## A.4 Training

```
#Initialize networks
D_A   = CGAN_D().to(device)
D_B   = CGAN_D().to(device)
G_A2B = CGAN_G().to(device)
G_B2A = CGAN_G().to(device)

pretrained = False
save_state = False
if pretrained:
    D_A.load_state_dict(torch.load('netD_A.pth'))
    D_B.load_state_dict(torch.load('netD_B.pth'))
    G_A2B.load_state_dict(torch.load('netG_A2B.pth'))
    G_B2A.load_state_dict(torch.load('netG_B2A.pth'))

# optimizers
d_opt_A   = torch.optim.Adam(D_A.parameters(), 0.0002, (0.5, 0.999))
g_opt_A2B = torch.optim.Adam(G_A2B.parameters(), 0.0001, (0.5, 0.999))
d_opt_B   = torch.optim.Adam(D_B.parameters(), 0.0002, (0.5, 0.999))
g_opt_B2A = torch.optim.Adam(G_B2A.parameters(), 0.0001, (0.5, 0.999))

# losses
criterion_gan   = torch.nn.MSELoss()
criterion_cycle = torch.nn.L1Loss()
criterion_id    = torch.nn.L1Loss()

target_real = Variable(torch.cuda.FloatTensor(batch_size).fill_(1.0), requires_grad=False)
target_fake = Variable(torch.cuda.FloatTensor(batch_size).fill_(0.0), requires_grad=False)

# lambdas
lambda_identity = 5
lambda_cycle    = 10

#Clean up all old variables on the GPU no longer in use
torch.cuda.empty_cache()

num_epochs = 10
visualization = ipywidgets.Output()
display.display(visualization)

with visualization:
    plt.figure(figsize=(20,10))
subplots = [plt.subplot(2, 4, k+1) for k in range(8)]

losses = {
    "loss_identity_B": [],
    "loss_identity_A": [],
    "loss_gan_A2B":    [],
    "loss_gan_B2A":    [],
    "loss_cycle_ABA":  [],
    "loss_cycle_BAB":  [],
    "total_loss":      [],
    "loss_discrim_A":  [],
    "loss_discrim_B":  []
}

for epoch in tqdm(range(num_epochs), unit='epoch'):
    for minibatch_nb, (real_A, real_B) in tqdm(enumerate(zip(train_loader_horses, train_loader_zebras))
        , total=len(train_loader_horses)):
        real_A = real_A.to(device)
        real_B = real_B.to(device)

        g_opt_A2B.zero_grad()
        g_opt_B2A.zero_grad()

        ### identity losses
        replicate_B       = G_A2B(real_B)
        loss_identity_B   = criterion_id(replicate_B, real_B) * lambda_identity

        replicate_A       = G_B2A(real_A)
        loss_identity_A   = criterion_id(replicate_A, real_A) * lambda_identity

        ### transform losses
        fake_B            = G_A2B(real_A)
        prediction_fake_B = D_B(fake_B)
        loss_gan_A2B      = criterion_gan(prediction_fake_B, target_real)

        fake_A            = G_B2A(real_B)
        prediction_fake_A = D_A(fake_A)
        loss_gan_B2A      = criterion_gan(prediction_fake_A, target_real)

```

```
        ### cycle losses
        cycle_A           = G_B2A(fake_B)
        loss_cycle_ABA    = criterion_cycle(cycle_A, real_A) * lambda_cycle

        cycle_B           = G_A2B(fake_A)
        loss_cycle_BAB    = criterion_cycle(cycle_B, real_B) * lambda_cycle

        ### total loss
        loss_G = loss_identity_A + loss_identity_B + loss_gan_A2B + loss_gan_B2A + loss_cycle_ABA +
            loss_cycle_BAB
        loss_G.backward()
        g_opt_A2B.step()
        g_opt_B2A.step()

        ### discriminator A
        d_opt_A.zero_grad()

        prediction_real_A = D_A(real_A)
        loss_real_A       = criterion_gan(prediction_real_A, target_real)

        fake_A            = G_B2A(real_B)
        prediction_fake_A = D_A(fake_A)
        loss_fake_A       = criterion_gan(prediction_fake_A, target_fake)

        loss_D_A = (loss_real_A + loss_fake_A) * lambda_identity
        losses["loss_discrim_A"].append(loss_D_A)
        loss_D_A.backward()
        d_opt_A.step()

        ### discriminator B
        d_opt_B.zero_grad()

        prediction_real_B = D_B(real_B)
        loss_real_B       = criterion_gan(prediction_real_B, target_real)

        fake_B            = G_A2B(real_A)
        prediction_fake_B = D_B(fake_B)
        loss_fake_B       = criterion_gan(prediction_fake_B, target_fake)

        loss_D_B = (loss_real_B + loss_fake_B) * lambda_identity
        losses["loss_discrim_B"].append(loss_D_B)
        loss_D_B.backward()
        d_opt_B.step()

        if minibatch_nb % 100 == 0:
            with torch.no_grad(), visualization:
                G_A2B.eval()
                G_B2A.eval()
                D_A.eval()
                D_B.eval()

                real_A = next(iter(test_loader_horses)).to(device)
                real_B = next(iter(test_loader_zebras)).to(device)

                fake_A = G_B2A(real_B)
                replicate_A = G_B2A(real_A)

                fake_B = G_A2B(real_A)
                replicate_B = G_A2B(real_B)

                cycle_A = G_B2A(fake_B)
                cycle_B = G_A2B(fake_A)

                subplots[0].imshow(real_A.cpu().squeeze(0).permute(1,2,0)/2 + .5)
                subplots[0].set_title("real A")
                subplots[0].axis("off")

                subplots[1].imshow(replicate_A.cpu().detach().squeeze(0).permute(1,2,0)/2 + .5)
                subplots[1].set_title("replicate A")
                subplots[1].axis("off")

                subplots[2].imshow(fake_A.cpu().detach().squeeze(0).permute(1,2,0)/2 + .5)
                subplots[2].set_title("fake A")
                subplots[2].axis("off")

                subplots[3].imshow(cycle_A.cpu().detach().squeeze(0).permute(1,2,0)/2 + .5)
                subplots[3].set_title("cycle A")
                subplots[3].axis("off")

                subplots[4].imshow(real_B.cpu().squeeze(0).permute(1,2,0)/2 + .5)
                subplots[4].set_title("real B")
                subplots[4].axis("off")

```

```
                    subplots[5].imshow(replicate_B.cpu().detach().squeeze(0).permute(1,2,0)/2 + .5)
                    subplots[5].set_title("replicate B")
                    subplots[5].axis("off")

                    subplots[6].imshow(fake_B.cpu().detach().squeeze(0).permute(1,2,0)/2 + .5)
                    subplots[6].set_title("fake B")
                    subplots[6].axis("off")

                    subplots[7].imshow(cycle_B.cpu().detach().squeeze(0).permute(1,2,0)/2 + .5)
                    subplots[7].set_title("cycle B")
                    subplots[7].axis("off")

                    display.display(plt.gcf())
                    display.clear_output(wait=True)

        losses["loss_identity_B"].append(loss_identity_B)
        losses["loss_identity_A"].append(loss_identity_A)
        losses["loss_gan_A2B"].append(loss_gan_A2B)
        losses["loss_gan_B2A"].append(loss_gan_B2A)
        losses["loss_cycle_ABA"].append(loss_cycle_ABA)
        losses["loss_cycle_BAB"].append(loss_cycle_BAB)
        losses["total_loss"].append(loss_G)

        if save_state:
            torch.save(G_A2B.state_dict(), 'netG_A2B.pth')
            torch.save(G_B2A.state_dict(), 'netG_B2A.pth')
            torch.save(D_A.state_dict(), 'netD_A.pth')
            torch.save(D_B.state_dict(), 'netD_B.pth')
```