

Deep Learning — Assignment 9

Assignment for week 9 of the 2023 Deep Learning course (NWI-IMC070) of the Radboud University.

Names: Luka Mucko, Luca Poli

Group: 46

Instructions:

- Fill in your names and the name of your group.
- Answer the questions and complete the code where necessary.
- Keep your answers brief, one or two sentences is usually enough.
- Re-run the whole notebook before you submit your work.
- Save the notebook as a PDF and submit that in Brightspace together with the `.ipynb` notebook file.
- The easiest way to make a PDF of your notebook is via File > Print Preview and then use your browser's print option to print to PDF.

Objectives

In this assignment you will

1. Implement and train a generative adversarial network.
2. Experiment with reverse gradient training.
3. Implement a CycleGAN.
4. Experiment with CycleGAN optimization.

Required software

If you haven't done so already, you will need to install the following additional libraries:

- `torch` and `torchvision` for PyTorch,
- `d2l` , the library that comes with the [Dive into deep learning](#) book.
- `PIL` , the python image library

All libraries can be installed with `pip install` .

```
In [11]: # render plots as png, not as svg
# (svg is very slow with large scatterplots)
%config InlineBackend.figure_formats = ['png']
%matplotlib inline
import csv
import glob
import re
from collections import defaultdict
import numpy as np
import scipy
import sklearn.datasets
import matplotlib.pyplot as plt
import PIL
import torch
import torch.autograd
import torchvision
import torchvision.transforms
from d2l import torch as d2l

from IPython import display

device = d2l.try_gpu()

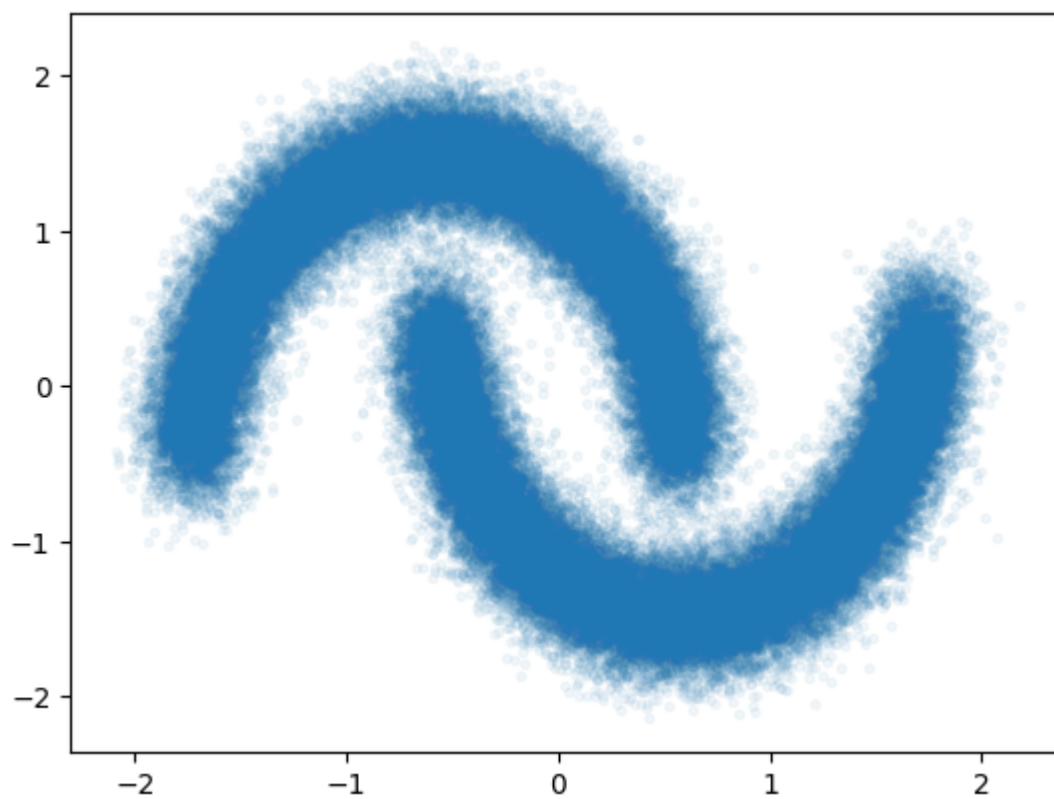
# Fix the seed to make the solutions more reproducible
torch.manual_seed(12345);
```

9.1 Moon dataset

The noisy moon dataset is a synthetic dataset with the following distribution:

```
In [3]: n_samples = 100000
noisy_moons = sklearn.datasets.make_moons(n_samples=n_samples, noise=.1)
noisy_moons[0][:, 0] -= np.mean(noisy_moons[0][:, 0])
noisy_moons[0][:, 0] /= np.std(noisy_moons[0][:, 0])
noisy_moons[0][:, 1] -= np.mean(noisy_moons[0][:, 1])
```

```
noisy_moons[0][:, 1] /= np.std(noisy_moons[0][:, 1])
plt.plot(noisy_moons[0][:, 0], noisy_moons[0][:, 1], '.', alpha=0.05);
```



(a) Run the following code to convert the data to a PyTorch dataset:

```
In [4]: moon_dataset = torch.utils.data.TensorDataset(torch.tensor(noisy_moons[0], dtype=torch.float32),
                                                         torch.tensor(noisy_moons[1], dtype=torch.float32))
```

9.2 Generator

We define a generator that generates samples from a learned distribution, based on a random noise input.

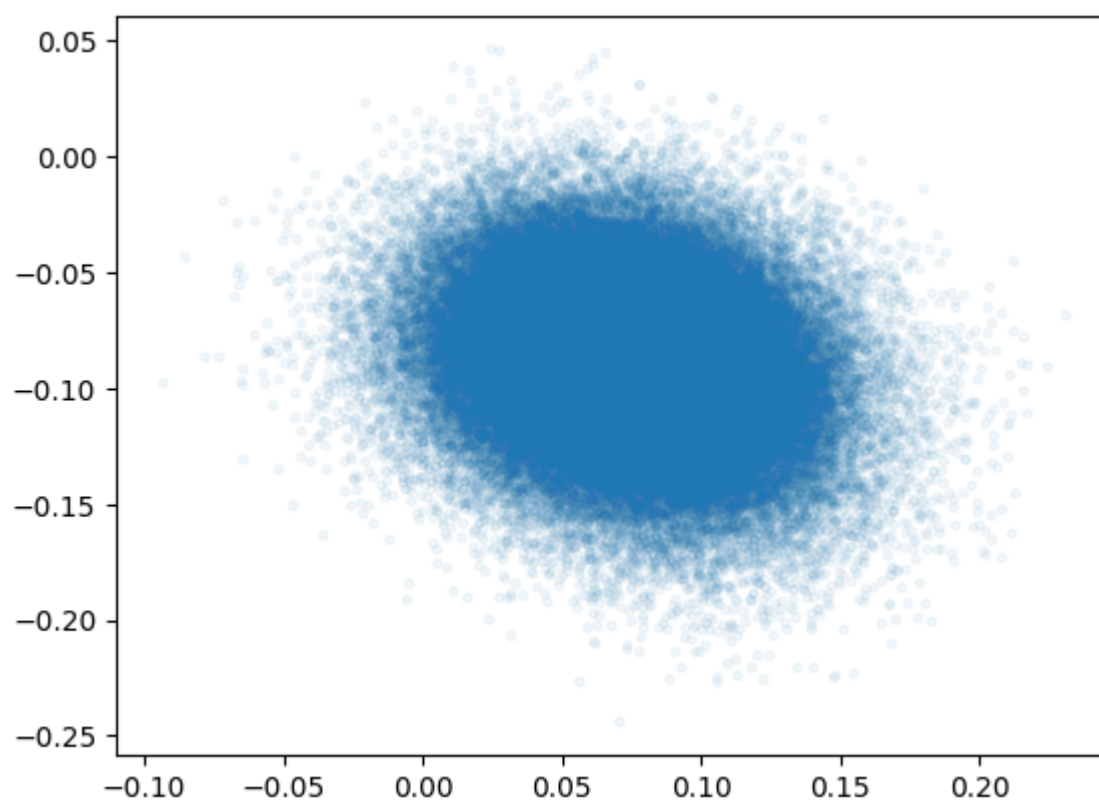
The generator accepts 1D input vector with 100 elements and has to output a 1D vector with 2 elements.

```
In [5]: class MoonGenerator(torch.nn.Module):
        def __init__(self, input_size=100, output_size=2):
            super().__init__()
            self.input_size = input_size
            self.output_size = output_size
            self.net = torch.nn.Sequential(
                torch.nn.Linear(input_size, 100),
                torch.nn.ReLU(),
                torch.nn.Linear(100, 100),
                torch.nn.ReLU(),
                torch.nn.Linear(100, output_size)
            )

        def forward(self, x):
            return self.net(x)
```

(a) Generate some samples from this generator before training and plot the resulting distribution.

```
In [6]: gen = MoonGenerator()
x = torch.rand((n_samples, 100)) * 2 - 1
y = gen(x).detach().cpu().numpy()
plt.plot(y[:, 0], y[:, 1], '.', alpha=0.05);
```



9.3 Untrainable dummy generator network

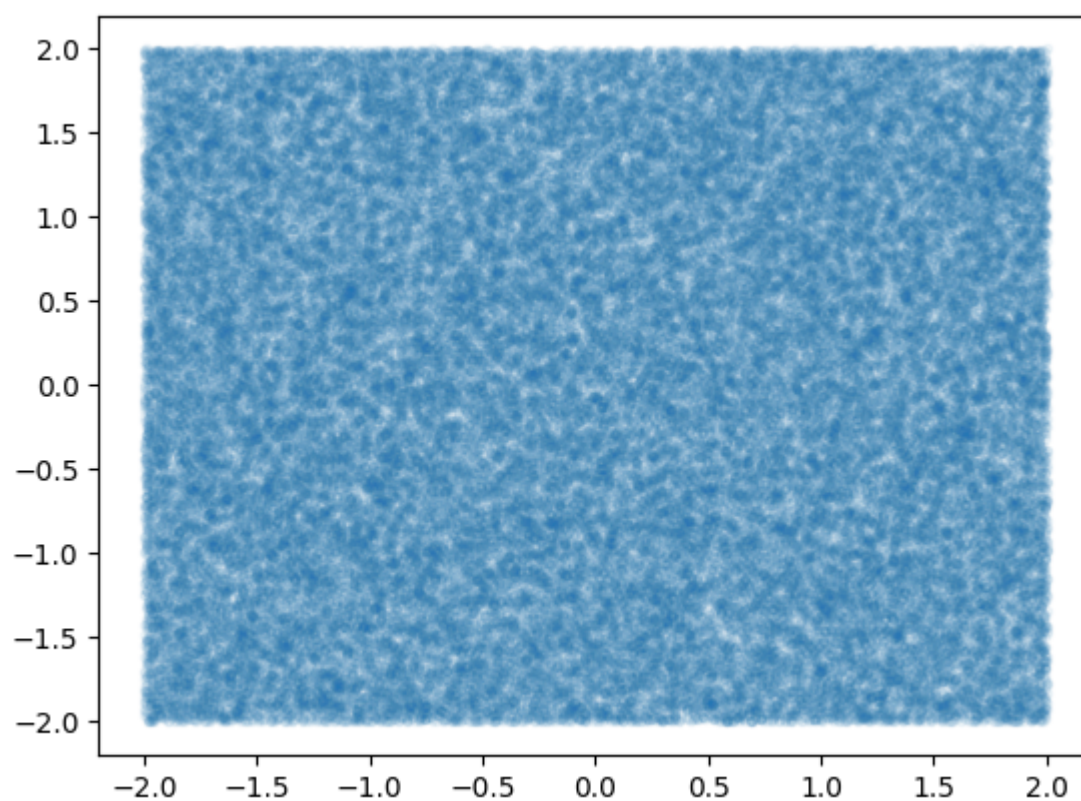
For our experiments, we also define an untrainable dummy generator network that produces samples from a uniform distribution. We'll use this later to investigate what our discriminator learns.

```
In [7]: class UniformMoonGenerator(torch.nn.Module):
        def __init__(self, input_size=100, output_size=2):
            super().__init__()
            self.input_size = input_size
            self.output_size = output_size
            self.dummy_param = torch.nn.Parameter(torch.tensor([0.]))

        def forward(self, x):
            return torch.rand((x.shape[0], self.output_size),
                              device=x.device, dtype=x.dtype) * 4 - 2
```

(a) Run the code to generate some samples from this generator and plot the resulting distribution.

```
In [8]: gen = UniformMoonGenerator()
x = torch.randn((n_samples, 100))
y = gen(x).detach().cpu().numpy()
plt.plot(y[:, 0], y[:, 1], '.', alpha=0.05);
```



9.4 Discriminator (1 point)

To train the generator, we need a discriminator that takes the samples from the generator and samples from the real distribution. For real samples, the discriminator should predict 1, for fake samples it should predict 0.

For stability, we will exclude the final sigmoid activation from the discriminator and use the [BCEWithLogitsLoss](#) function during training.

(a) Inspect the code for the discriminator below:

```
In [9]: class MoonDiscriminator(torch.nn.Module):
    def __init__(self, inputs=2, hidden=1024):
        super().__init__()
        self.net = torch.nn.Sequential(
            torch.nn.Linear(inputs, hidden),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden, hidden),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden, 1)
        )
        # Note: Although this is a binary classifier, we do not yet apply
        # a sigmoid activation here. Instead, we'll use the
        # BCEWithLogitsLoss later to compute sigmoid + BCE loss in
        # a numerically stable way.

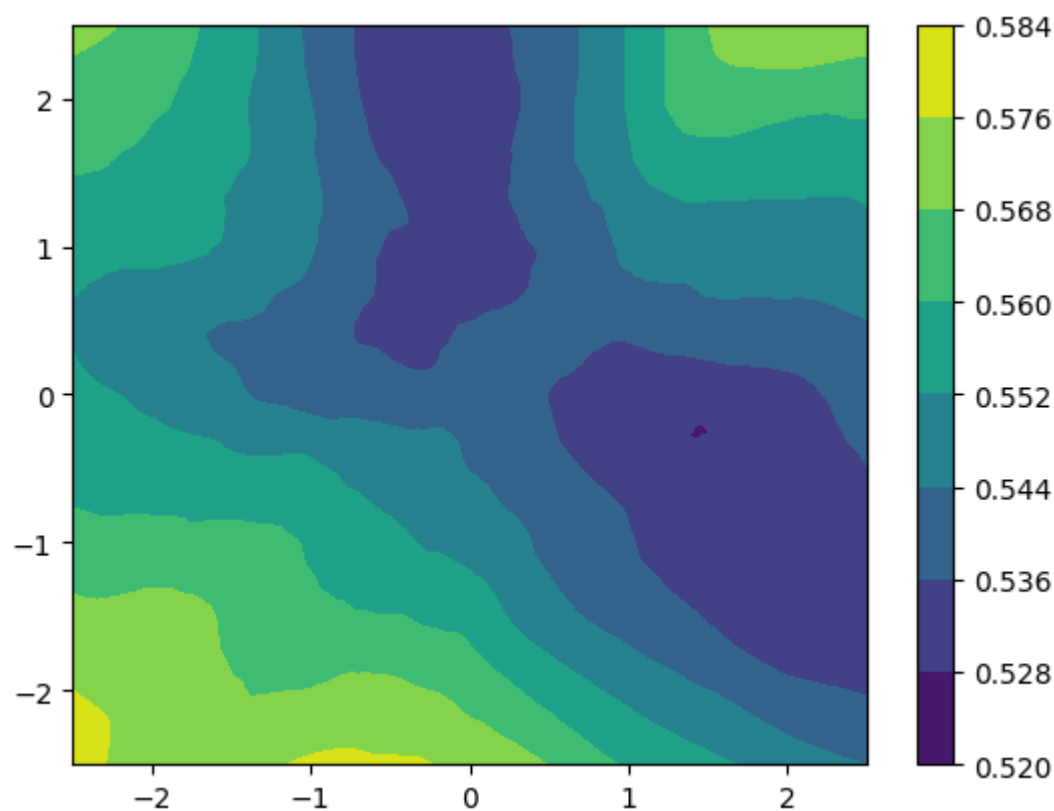
    def forward(self, x):
        return self.net(x)
```

We can plot the value of the discriminator in our sample space to see what it is doing.

(b) Run the code to plot the output of an untrained discriminator:

```
In [10]: def plot_discriminator(discriminator, xmin=-2.5, xmax=2.5, steps=500, device=device):
    x0, x1 = torch.meshgrid(torch.linspace(xmin, xmax, steps=steps),
                             torch.linspace(xmin, xmax, steps=steps), indexing='ij')
    x = torch.stack([x0.flatten(), x1.flatten()], axis=1).to(device)
    y = discriminator(x)
    y = torch.sigmoid(y).detach().cpu().numpy()
    y = y.reshape(x0.shape)
    plt.contourf(x0, x1, y)
    plt.colorbar()

disc2 = MoonDiscriminator().to(device)
plot_discriminator(disc2)
```



(c) How should we expect this plot to look after training the discriminator for the moon dataset?

(1 point)

We expect this plot to look like the moons, with the discriminator predicting 1 inside the moons and 0 outside.

9.5 Adversarial training loop (1 point)

Now we have a generator and a discriminator, we can attempt to train the model. We will define a training function that implements the adversarial training procedure:

For each minibatch of real samples:

1. Generate a batch of fake samples;
2. Compute the discriminator loss on the real and fake samples;
3. Optimize the discriminator;
4. Generate another batch of fake samples;
5. Compute the generator loss on the fake samples;
6. Update the generator.

To monitor training, we'll print the discriminator and generator loss. We'll also monitor the accuracy of the discriminator (the percentage of correctly labeled real and fake samples) and the 'accuracy' of the generator (the percentage of fake samples incorrectly labeled as real by the

discriminator).

(a) Complete the training loop below:

(1 point)

```
In [11]: def train_adversarial(generator, discriminator, data_loader, epochs=10,
                                lr_gen=0.001, lr_disc=0.001, device=device):
    gen_optimizer = torch.optim.Adam(generator.parameters(), lr=lr_gen)
    disc_optimizer = torch.optim.Adam(discriminator.parameters(), lr=lr_disc)

    bce_logits_loss = torch.nn.BCEWithLogitsLoss()

    for epoch in range(epochs):
        epoch_disc_loss = 0
        epoch_gen_loss = 0
        epoch_disc_acc = 0
        epoch_gen_acc = 0
        mb_count = 0

        for x_real, _ in data_loader:
            x_real = x_real.to(device)

            ## 1. Discriminator
            # generate noise for the generator
            rand_for_gen = torch.rand((x_real.shape[0], generator.input_size),
                                       device=x_real.device, dtype=x_real.dtype) * 2 - 1

            # generate fake samples
            x_fake = generator(rand_for_gen)

            # run discriminator on real and fake samples
            d_real = discriminator(x_real)
            d_fake = discriminator(x_fake)

            # compute discriminator loss
            # - for real samples, the discriminator should predict 1
            # - for fake samples, the discriminator should predict 0
            disc_loss = (bce_logits_loss(d_real, torch.ones_like(d_real)) +
                        bce_logits_loss(d_fake, torch.zeros_like(d_fake)))
            disc_acc = (torch.mean((d_real > 0).to(torch.float)) +
                        torch.mean((d_fake < 0).to(torch.float))) / 2

            # update discriminator
            disc_optimizer.zero_grad()
            disc_loss.backward()
            disc_optimizer.step()

            ## 2. Generator
            # generate another batch of fake samples
            rand_for_gen = torch.rand((x_real.shape[0], generator.input_size),
                                       device=x_real.device, dtype=x_real.dtype) * 2 - 1
            x_fake = generator(rand_for_gen)

            # compute generator loss
            d_fake = discriminator(x_fake)
            # TODO: compute the generator loss using d_fake and bce_logits_loss
            # and the appropriate target value (see the implementation for
            # the discriminator loss)
            gen_loss = bce_logits_loss(d_fake, torch.ones_like(d_fake))
            # for the generator, we compute how many generated samples were given
            # the label 'real' by the discriminator
            gen_acc = torch.mean((d_fake > 0).to(torch.float))

            # update generator
            gen_optimizer.zero_grad()
            gen_loss.backward()
            gen_optimizer.step()

            ## 3. Statistics
            epoch_disc_loss += disc_loss.item()
            epoch_gen_loss += gen_loss.item()
            epoch_disc_acc += disc_acc.item()
            epoch_gen_acc += gen_acc.item()
            mb_count += 1

        print('Epoch %d: disc_loss=%f gen_loss=%f disc_acc=%f gen_acc=%f' %
              (epoch, epoch_disc_loss / mb_count, epoch_gen_loss / mb_count,
               epoch_disc_acc / mb_count, epoch_gen_acc / mb_count))
```

9.6 Experiment: Train the discriminator only (1 point)

First, we'll train the discriminator only, using the dummy generator to generate samples from a uniform distribution.

(a) Run the code to train the discriminator:

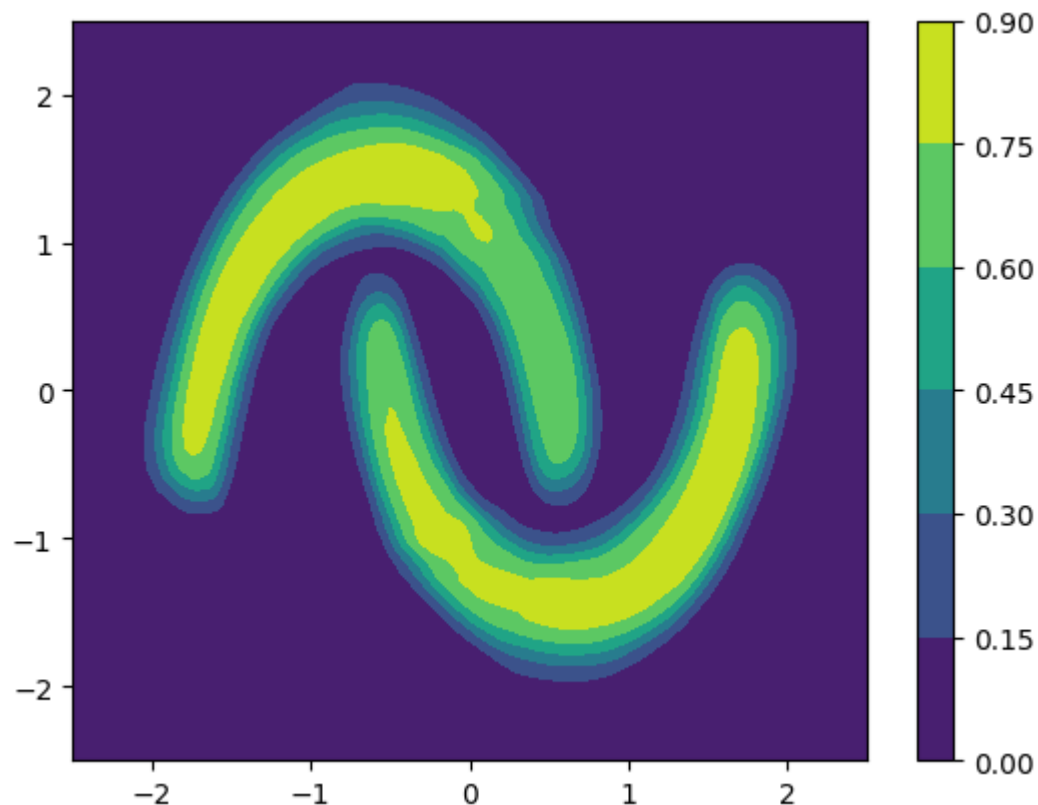

```
In [12]: gen1 = UniformMoonGenerator().to(device)
disc1 = MoonDiscriminator().to(device)

loader = torch.utils.data.DataLoader(moon_dataset, batch_size=128)
train_adversarial(gen1, disc1, loader, epochs=10, lr_gen=0.001, lr_disc=0.001, device=device)
```

```
Epoch 0: disc_loss=0.984504 gen_loss=2.848409 disc_acc=0.755500 gen_acc=0.347786
Epoch 1: disc_loss=0.907978 gen_loss=3.399391 disc_acc=0.785801 gen_acc=0.306176
Epoch 2: disc_loss=0.901225 gen_loss=3.502365 disc_acc=0.787724 gen_acc=0.312880
Epoch 3: disc_loss=0.899779 gen_loss=3.588417 disc_acc=0.787804 gen_acc=0.307755
Epoch 4: disc_loss=0.892155 gen_loss=3.665708 disc_acc=0.790756 gen_acc=0.309143
Epoch 5: disc_loss=0.893504 gen_loss=3.689910 disc_acc=0.790002 gen_acc=0.313549
Epoch 6: disc_loss=0.892215 gen_loss=3.840442 disc_acc=0.789877 gen_acc=0.308594
Epoch 7: disc_loss=0.890746 gen_loss=3.850468 disc_acc=0.789922 gen_acc=0.309763
Epoch 8: disc_loss=0.886762 gen_loss=3.953291 disc_acc=0.791815 gen_acc=0.310822
Epoch 9: disc_loss=0.883503 gen_loss=3.966144 disc_acc=0.793318 gen_acc=0.312220
```

(b) Plot the discriminator output and inspect the result.

```
In [13]: plot_discriminator(disc1)
```



(c) Why does the discriminator not predict 1.00 inside the moons?

(1 point)

1.00 would imply that the discriminator knows exactly all the points of the moons. In other words, the loss reached its minimum, the generator represents a distribution made up of delta spikes at the points of the moons set. This is not doable in practice since the discriminator uses a sigmoid which achieves 1.00 asymptotically at infinity.

9.7 Experiment: Train the generator and discriminator (8 points)

We'll now train the model with the trainable generator.

(a) Train the generator and discriminator together:

```
In [14]: gen2 = MoonGenerator().to(device)
disc2 = MoonDiscriminator().to(device)

loader = torch.utils.data.DataLoader(moon_dataset, batch_size=128)
train_adversarial(gen2, disc2, loader, epochs=10, lr_gen=0.001, lr_disc=0.001, device=device)
```

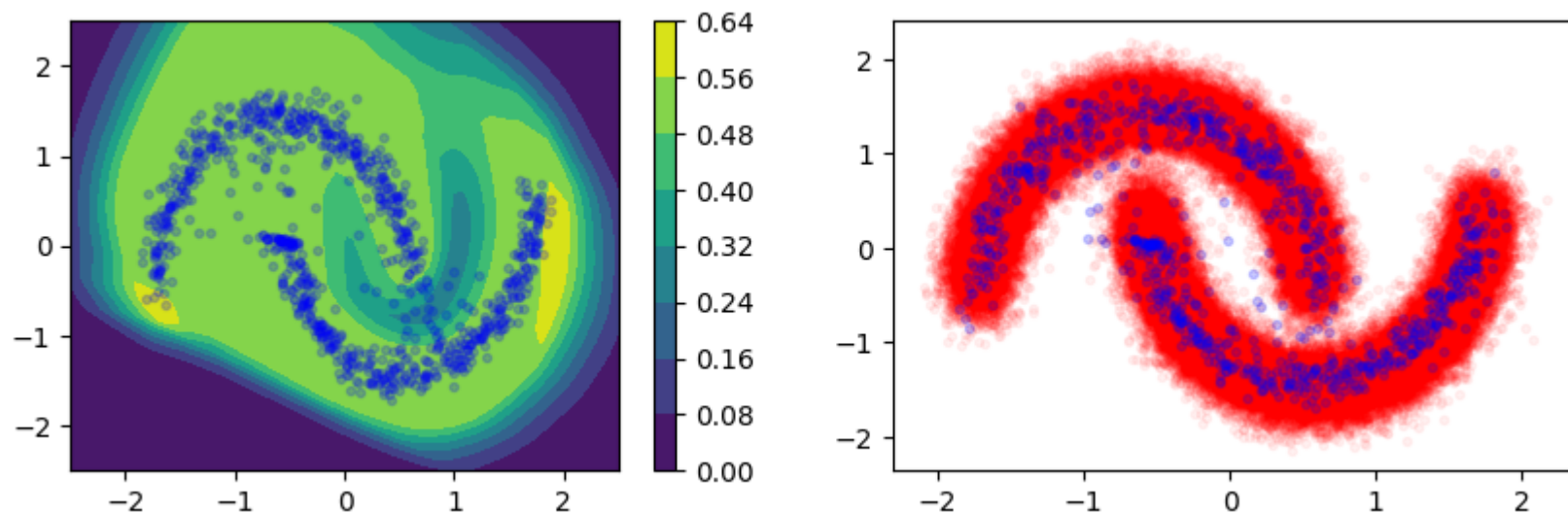
```
Epoch 0: disc_loss=1.063887 gen_loss=1.478187 disc_acc=0.718440 gen_acc=0.261998
Epoch 1: disc_loss=1.248615 gen_loss=0.926934 disc_acc=0.631594 gen_acc=0.297095
Epoch 2: disc_loss=1.309411 gen_loss=0.826754 disc_acc=0.592371 gen_acc=0.328515
Epoch 3: disc_loss=1.366315 gen_loss=0.737275 disc_acc=0.537969 gen_acc=0.395580
Epoch 4: disc_loss=1.327233 gen_loss=0.814000 disc_acc=0.573195 gen_acc=0.342961
Epoch 5: disc_loss=1.349248 gen_loss=0.762383 disc_acc=0.554877 gen_acc=0.348386
Epoch 6: disc_loss=1.375089 gen_loss=0.715617 disc_acc=0.526530 gen_acc=0.365709
Epoch 7: disc_loss=1.367725 gen_loss=0.752895 disc_acc=0.535751 gen_acc=0.338025
Epoch 8: disc_loss=1.367127 gen_loss=0.744468 disc_acc=0.533613 gen_acc=0.432385
Epoch 9: disc_loss=1.383085 gen_loss=0.699348 disc_acc=0.504591 gen_acc=0.468011
```

(b) Run the code below to plot the generated samples, the discriminator output, and the real samples.

Note: If you don't get a good results, try to run the model again. This model is quite sensitive to the random initialisation.

```
In [15]: def plot_generator(generator, n_samples=1000, device=device):
x = torch.rand((n_samples, 100)).to(device) * 2 - 1
y = generator(x).detach().cpu().numpy()
plt.plot(y[:, 0], y[:, 1], 'b.', alpha=0.2)
```

```
plt.figure(figsize=(10, 3))
plt.subplot(1, 2, 1)
plot_discriminator(disc2)
plot_generator(gen2)
plt.subplot(1, 2, 2)
plt.plot(noisy_moons[0][:, 0], noisy_moons[0][:, 1], 'r.', alpha=0.05)
plot_generator(gen2)
```



(c) Briefly discuss this result.

(1 point)

As we can see, the generator's loss and the discriminator's loss are converged to a similar value. So the generator now is able to create more realistic instances. In the figures we can see that the generated instances are similar to the real one, so the discriminator is less able to distinguish them

Compare the output of the new discriminator with the output of the discriminator trained without a generator.

(d) Are the discriminator outputs the same? Explain why this happens.

(1 point)

The output of the two discriminators are different, the first one was able to detect with more precision the areas of the moons, while the second one has worse precision. This happen because the generator is able to mimic better the real distribution, so the task of the discriminator is harder.

(e) Does the discriminator still reach a high accuracy? Why (not?)

(1 point)

No, the discriminator reach a poor accuracy (~0.5). Because the fake instances are more similar to the real ones, so the task is harder.

(f) How can we see if the model is working well based on the discriminator accuracy?

(2 points)

Based on the accuracy of the discriminator we can understand the abilities of the generator and discriminator: if it's close to 1, the generator is not able to generate realistic instances; if is close to 0 the discriminator is missclassifying most of the sample, so there are some problems with the discriminator or the generator is too good; if is close to 0.5 the discriminator is not better than random chance, so generator is able to generate realistic instances.

(g) Compare the distribution learned by the generator with the real distribution. What are the main differences?

(1 point)

The two distributions are similar but some differences along the edges. The fake moons are thinner and shortier (the didn't reach properly the ends)

(h) Explain the differences you observed above: Why are the generated and real distributions different?

(1 point)

The cause of those differences could be that there are less real sample with point in that positions (because that depends on the noise of the real data); so the generator is less able to mimic those particular instances.

(i) How can you make the distributions more similar?

(1 point)

We could make the distributions more similar by increasing the epochs or the model complexity.

9.8 Gradient reversal (5 point)

As an alternative to training the discriminator and generator separately, we can also train the model with a gradient reversal layer that reverses the gradient coming from the discriminator:

```
Forward: generator -> discriminator.
Backward: generator gradient <- gradient reversal <- discriminator gradient.
```

In PyTorch, we'll implement this as a function `revgrad(x)` that will reverse the gradient that passes through it. You can use it like this:

```
y = discriminator(revgrad(generator(x)))
loss = loss_fn(y, target)
```

```
loss.backward()
```

(a) Complete the code to define the `revgrad` gradient reversal function:

(1 point)

```
In [12]: class RevGrad(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        output = input
        return output

    @staticmethod
    def backward(ctx, grad_output):
        # TODO: Compute the reverse of the gradient
        grad_input = -grad_output
        return grad_input

revgrad = RevGrad.apply
```

The training loop is now a bit simpler than before, because we do not have to compute the generator loss separately.

(b) Complete the new training function:

(1 point)

```
In [17]: def train_adversarial_revgrad(generator, discriminator, data_loader, epochs=10, lr=0.001, device=device):
    parameters = list(generator.parameters()) + list(discriminator.parameters())
    optimizer = torch.optim.Adam(parameters, lr=lr)

    bce_logits_loss = torch.nn.BCEWithLogitsLoss()

    for epoch in range(epochs):
        epoch_loss = 0
        epoch_acc_real = 0
        epoch_acc_fake = 0
        mb_count = 0

        for x_real, _ in data_loader:
            x_real = x_real.to(device)

            # generate fake samples
            rand_for_gen = torch.rand((x_real.shape[0], generator.input_size),
                                      device=x_real.device, dtype=x_real.dtype) * 2 - 1
            x_fake = generator(rand_for_gen)

            # run discriminator on real and random samples,
            # reverse the gradient for the generator
            d_real = discriminator(x_real)
            # TODO: compute the discriminator output like before,
            # but include the gradient reversal layer
            d_fake = discriminator(revgrad(x_fake))

            # compute loss
            loss_real = bce_logits_loss(d_real, torch.ones_like(d_real))
            loss_fake = bce_logits_loss(d_fake, torch.zeros_like(d_fake))
            loss = loss_real + loss_fake

            # compute discriminator accuracy
            acc_real = torch.mean((d_real > 0).to(torch.float))
            acc_fake = torch.mean((d_fake < 0).to(torch.float))

            # update generator and discriminator
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # update statistics
            epoch_loss += loss.item()
            epoch_acc_real += acc_real.item()
            epoch_acc_fake += acc_fake.item()
            mb_count += 1

        print('Epoch %d: loss=%f acc_real=%f acc_fake=%f' %
              (epoch, epoch_loss / mb_count,
               epoch_acc_real / mb_count, epoch_acc_fake / mb_count))
```

(c) Train a generator and discriminator with the new training function:

```
In [18]: gen3 = MoonGenerator().to(device)
disc3 = MoonDiscriminator().to(device)

loader = torch.utils.data.DataLoader(moon_dataset, batch_size=128)
train_adversarial_revgrad(gen3, disc3, loader, epochs=10, lr=0.001, device=device)
```

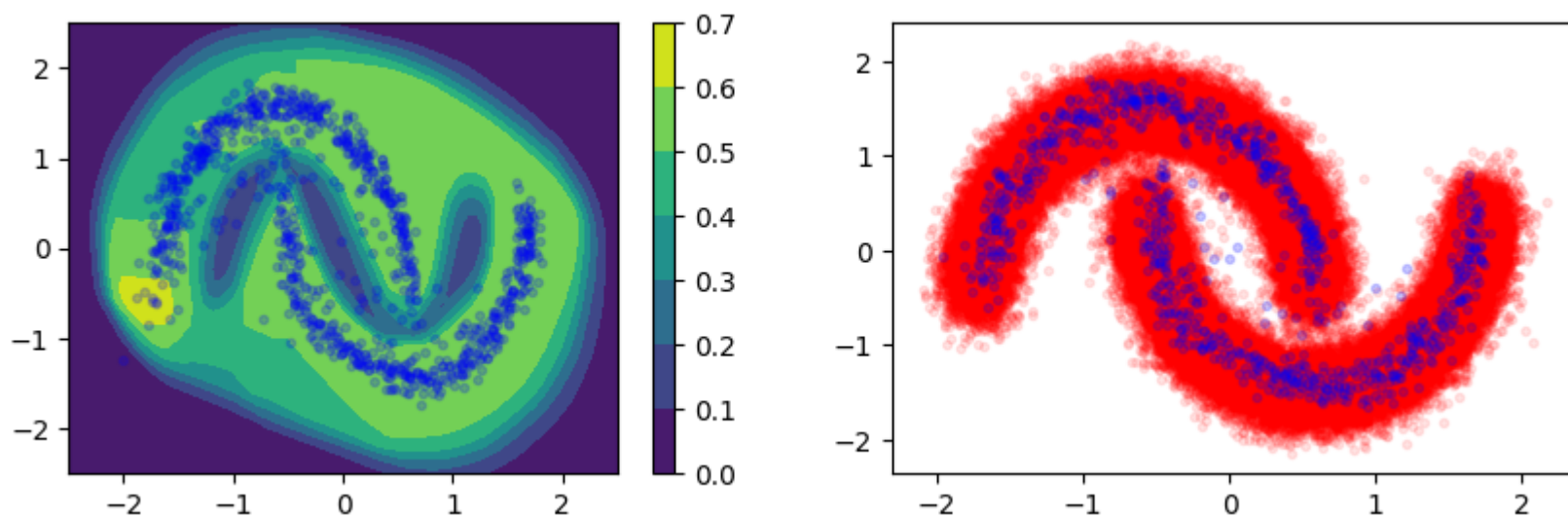


```
Epoch 0: loss=1.067242 acc_real=0.720998 acc_fake=0.713935
Epoch 1: loss=1.228348 acc_real=0.667299 acc_fake=0.627258
Epoch 2: loss=1.286362 acc_real=0.614090 acc_fake=0.602282
Epoch 3: loss=1.341042 acc_real=0.578445 acc_fake=0.542279
Epoch 4: loss=1.369710 acc_real=0.616708 acc_fake=0.445802
Epoch 5: loss=1.376202 acc_real=0.698030 acc_fake=0.337216
Epoch 6: loss=1.189355 acc_real=0.695073 acc_fake=0.565547
Epoch 7: loss=1.207557 acc_real=0.695542 acc_fake=0.578145
Epoch 8: loss=1.338995 acc_real=0.588815 acc_fake=0.517633
Epoch 9: loss=1.378203 acc_real=0.582251 acc_fake=0.448040
```

(d) Plot and inspect the results:

```
In [19]: plt.figure(figsize=(10, 3))
plt.subplot(1, 2, 1)
plot_discriminator(disc3)
plot_generator(gen3)

plt.subplot(1, 2, 2)
plt.plot(noisy_moons[0][:, 0], noisy_moons[0][:, 1], 'r.', alpha=0.1)
plot_generator(gen3)
```



(e) Briefly discuss the result.

(1 point)

As we can see from the figures, the results are similar. But in this case the generator is a bit less precise at generating fake samples.

(f) What are some advantages and disadvantages of the gradient reversal layer, compared with the previous two-step approach?

(2 points)

The main advantage of this method is that we don't need to train the generator and the discriminator separately, so we can save some time. The main disadvantage is that the generator is less precise at generating fake samples.

9.9 Emoji dataset

For the second part of this assignment we will borrow an emoji dataset (and some ideas) from a course at the [University of Toronto](https://www.cs.toronto.edu/~jba/emojis.tar.gz).

The dataset contains images of Apple-style and Windows-style emojis. You can [download the files](https://www.cs.toronto.edu/~jba/emojis.tar.gz) yourself or use the code below.

(a) Download the dataset and extract the files:

```
In [2]: !wget -c http://www.cs.toronto.edu/~jba/emojis.tar.gz
!tar xzf emojis.tar.gz

--2023-11-18 17:28:10-- http://www.cs.toronto.edu/~jba/emojis.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5933583 (5.7M) [application/x-gzip]
Saving to: 'emojis.tar.gz'

emojis.tar.gz      100%[=====] 5.66M  1.59MB/s   in 4.0s

2023-11-18 17:28:15 (1.41 MB/s) - 'emojis.tar.gz' saved [5933583/5933583]
```

We'll resize the images to 32 by 32 pixels and normalize the RGB intensities to values between -1 and 1.

(b) Run the code to construct the datasets:

```
In [13]: def image_loader(path):
with open(path, 'rb') as f:
img = PIL.Image.open(f)
return img.convert('RGBA').convert('RGB')

transform = torchvision.transforms.Compose([
torchvision.transforms.Resize((32, 32)),
```

```
torchvision.transforms.ToTensor(),
torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

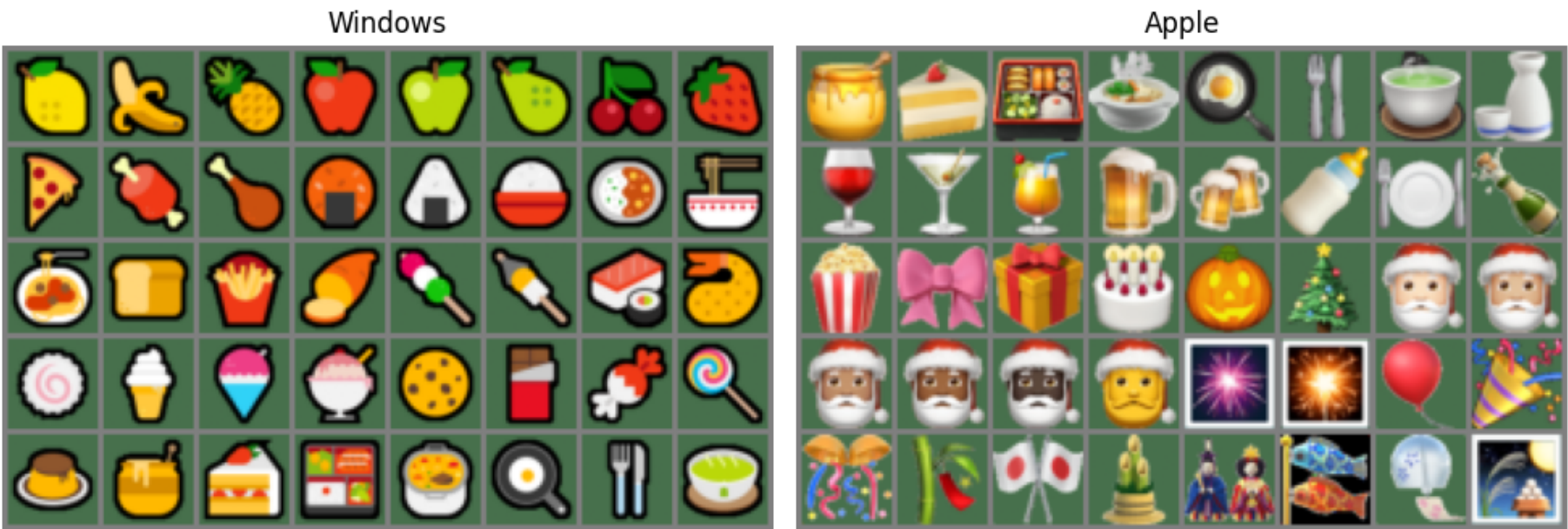
d_windows = torchvision.datasets.ImageFolder('emojis/Windows/', transform, loader=image_loader)
d_apple = torchvision.datasets.ImageFolder('emojis/Apple/', transform, loader=image_loader)
```

(c) Plot a few images to see the different styles:

```
In [14]: def image_grid(d, idxs):
         images = [d[idx][0] for idx in idxs]
         grid = torchvision.utils.make_grid(images)
         return grid.numpy().transpose(1, 2, 0) / 2 + 0.5

# Depending on the PyTorch version, this code might print
# a warning about transparency. This is not a problem.

plt.figure(figsize=(10, 10))
plt.subplot(1, 2, 1)
plt.imshow(image_grid(d_windows, range(100, 140)))
plt.title('Windows')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(image_grid(d_apple, range(400, 440)))
plt.title('Apple')
plt.axis('off')
plt.tight_layout()
```



9.10 CycleGAN (2 points)

We'll try to train a CycleGAN that can translate emojis between the Windows and Apple styles.

This CycleGAN has the following components:

- A generator that translates from Windows to Apple;
- A generator that translates from Apple to Windows;
- A discriminator that discriminates between real and fake emojis from the Windows distribution;
- A discriminator that discriminates between real and fake emojis from the Apple distribution.

Generator

First, we define the generator. We'll use the same generator architecture for both directions. Unlike before, the generator does not take random noise as input, but expects a 32 by 32 RGB image as input and returns a 32 by 32 RGB image as output.

The generator has the following structure:

- Input: 32x32 pixels, 3 channels.
- Two downsampling + convolution blocks:
kernel size = (5, 5), stride = 2, padding = ?,
from 3 -> 32 -> 64 channels.
- One convolution block in the middle:
kernel size = (5, 5), stride = 1, padding = ?,
from 64 -> 64 channels.
- Two upsampling + convolution blocks:
upsampling (scale factor 2) followed by convolution,
kernel size = (5, 5), stride = 1, padding = ?,
from 64 -> 32 -> 3 channels.
- Output: 32x32 pixels, 3 channels.

Add batch normalization and ReLU activations after each convolution, except after the very last layer.

The images have a [-1, +1] range, so the last output should use a tanh activation without BN.

(a) Complete the code below:

(2 points)

```
In [15]: class CycleGenerator(torch.nn.Module):
    def __init__(self, input_size=100):
        super().__init__()
        self.input_size = input_size
        self.net = torch.nn.Sequential(
            # downsampling 32 -> 16 -> 8 pixels
            # TODO: implement the downsampling part as described above
            torch.nn.Conv2d(3, 32, kernel_size=5, stride=2, padding=2),
            torch.nn.BatchNorm2d(32), torch.nn.ReLU(),
            torch.nn.Conv2d(32, 64, kernel_size=5, stride=2, padding=2),
            torch.nn.BatchNorm2d(64), torch.nn.ReLU(),
            # no downsampling, no upsampling
            torch.nn.Conv2d(64, 64, (5, 5), padding=2, bias=False),
            torch.nn.BatchNorm2d(64),
            torch.nn.ReLU(),

            # upsampling 8 -> 16 -> 32 pixels
            # TODO: complete the upsampling part as described above
            torch.nn.Upsample(scale_factor=2),
            torch.nn.Conv2d(64, 32, (5, 5), stride=1, padding=2),
            torch.nn.BatchNorm2d(32), torch.nn.ReLU(),
            torch.nn.Upsample(scale_factor=2),
            torch.nn.Conv2d(32, 3, (5, 5), stride=1, padding=2),
            torch.nn.Tanh(),
        )

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

# the output should have the same shape as the input
assert CycleGenerator()(torch.zeros((30, 3, 32, 32))).shape == torch.Size([30, 3, 32, 32]), "the output should have the same shape as the input"
assert torch.min(CycleGenerator()(torch.zeros((30, 3, 32, 32)))) > -1, "outputs should be in the range [-1,1]"
assert torch.max(CycleGenerator()(torch.zeros((30, 3, 32, 32)))) < 1, "outputs should be in the range [-1,1]"
```

Discriminator

The discriminator is similar in concept to what we had in the GAN model: it takes an image and predicts 1 for a real image and 0 for a fake.

- Input: 32x32 pixels, 3 channels.
- Three downsampling + convolution blocks:
kernel size = (5, 5), stride = 2, padding = ?,
from 3 -> 64 -> 64 -> 64 channels.
- One fully connected layer from (64*4*4) to 1.
- Output: 1 output element.

Add batch normalization and ReLU after each convolution, except at the end of the network.

(b) Read through the code below:

```
In [16]: class Discriminator(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.net = torch.nn.Sequential(
            # downsampling 32 -> 16 -> 8 -> 4
            torch.nn.Conv2d(3, 64, (5, 5), stride=2, padding=2, bias=False),
            torch.nn.BatchNorm2d(64),
            torch.nn.ReLU(),

            torch.nn.Conv2d(64, 64, (5, 5), stride=2, padding=2, bias=False),
            torch.nn.BatchNorm2d(64),
            torch.nn.ReLU(),

            torch.nn.Conv2d(64, 64, (5, 5), stride=2, padding=2, bias=False),
            torch.nn.BatchNorm2d(64),
            torch.nn.ReLU(),

            torch.nn.Flatten(),

            torch.nn.Linear(64 * 4 * 4, 1)
        )
        # Note: Although this is a binary classifier, we do not apply
        # a sigmoid activation here. We'll optimize a mean-squared
        # error to make the discriminator's task a bit harder and
        # get a slightly better gradient.

    def forward(self, x):
        return self.net(x)
```

```
# the output shape should be (30, 1)
assert Discriminator()(torch.zeros((30, 3, 32, 32))).shape == torch.Size([30, 1]), "the output should have shape [30,1]"
```

9.11 CycleGAN training loop (2 points)

The training loop for the GAN with cycle-consistency loss follows the following procedure:

For each batch of samples from domain A and B:

- Use the generators to predict the fake B given A, and fake A given B.
- Use the generators to reconstruct A given fake B, and B given fake A.

The discriminator loss is composed of:

- The discriminator losses for real samples from A and B.
- The discriminator losses for fake samples from A and B.

The cycle-consistency loss is composed of:

- The reconstruction loss comparing the real A with the cycled A->B->A.
- The reconstruction loss comparing the real B with the cycled B->A->B.

Finally, the two groups losses are combined with a weight `lambda_cycle` for the cycle-consistency loss:

```
loss = discriminator_loss + lambda_cycle * cycle-consistency loss
```

(a) Complete the code below to implement this procedure:

(2 points)

```
In [17]: def train_cycle(generator_ab, generator_ba, discriminator_a, discriminator_b,
                        data_loader_a, data_loader_b,
                        epochs=10, lr=0.001, lambda_cycle=0.1, device=device):
    mse_loss = torch.nn.MSELoss()
    models = torch.nn.ModuleList([generator_ab, generator_ba, discriminator_a, discriminator_b])
    optimizer = torch.optim.Adam(models.parameters(), lr=lr, betas=(0.5, 0.999))

    plt.figure(figsize=(10, 15))

    for epoch in range(epochs):
        epoch_stats = defaultdict(lambda: 0)
        mb_count = 0

        disc_a.train()
        disc_b.train()
        gen_ab.train()
        gen_ba.train()

        for (real_a, _), (real_b, _) in zip(loader_a, loader_b):
            real_a = real_a.to(device)
            real_b = real_b.to(device)

            # compute fake images A->B->A
            fake_ab = generator_ab(real_a)
            cycle_aba = generator_ba(fake_ab)

            # compute fake images B->A->B
            fake_ba = generator_ba(real_b)
            cycle_bab = generator_ab(fake_ba)

            # run discriminator on real and fake images
            d_real_a = discriminator_a(real_a)
            # TODO: compute other discriminator output, use gradient reversal where necessary
            d_real_b = discriminator_b(real_b) # TODO
            d_fake_ba = discriminator_a(revgrad(cycle_aba)) # TODO
            d_fake_ab = discriminator_b(revgrad(cycle_bab)) # TODO

            # compute discriminator loss
            # we optimize the MSE loss function to make the gradients of
            # the discriminator a bit easier to use
            loss_real_a = mse_loss(d_real_a, torch.ones_like(d_real_a))
            loss_real_b = mse_loss(d_real_b, torch.ones_like(d_real_b)) # TODO
            loss_fake_a = mse_loss(d_fake_ba, torch.zeros_like(d_fake_ba)) # TODO
            loss_fake_b = mse_loss(d_fake_ab, torch.zeros_like(d_fake_ab)) # TODO

            # compute cycle-consistency loss
            loss_cycle_a = mse_loss(cycle_aba, real_a)
            loss_cycle_b = mse_loss(cycle_bab, real_b)

            # compute loss
            loss = loss_real_a + loss_real_b + \
                    loss_fake_a + loss_fake_b + \
                    lambda_cycle * (loss_cycle_a + loss_cycle_b)

            # optimize
```

```

optimizer.zero_grad()
loss.backward()
optimizer.step()

# update statistics
epoch_stats['loss'] += loss.item()
epoch_stats['loss_real_a'] += loss_real_a.item()
epoch_stats['loss_real_b'] += loss_real_b.item()
epoch_stats['loss_fake_a'] += loss_fake_a.item()
epoch_stats['loss_fake_b'] += loss_fake_b.item()
epoch_stats['loss_cycle_a'] += loss_cycle_a.item()
epoch_stats['loss_cycle_b'] += loss_cycle_b.item()
mb_count += 1

if epoch % 5 == 0:
    print('Epoch %d: ' % epoch, end='')
    for k, v in epoch_stats.items():
        print(' %s=%6.4f' % (k, v / mb_count), end='')

    images_for_plot = {
        'real_a': real_a, 'fake_ab': fake_ab, 'cycle_aba': cycle_aba,
        'real_b': real_b, 'fake_ba': fake_ba, 'cycle_bab': cycle_bab,
    }

    for k in range(10):
        for i, (im_title, im) in enumerate(images_for_plot.items()):
            plt.subplot(10, 6, k * 6 + i + 1)
            plt.imshow(im[k].detach().cpu().numpy().transpose(1, 2, 0) / 2 + 0.5)
            if k == 0:
                plt.title(im_title)
            plt.axis('off')
    plt.tight_layout()
    display.display(plt.gcf())
    display.clear_output(wait=True)

```

9.12 Experiment: CycleGAN training (6 points)

We can now train our CycleGAN model.

(a) Run the code below and play with the hyperparameters if necessary to learn a reasonable output.

Note that GANs can be notoriously difficult to train, so don't worry if your results are not perfect. Hopefully, you will be able to get somewhat recognizable results, but it's more important that you can interpret and discuss what happens.


```

In [10]: gen_ab = CycleGenerator().to(device)
gen_ba = CycleGenerator().to(device)
disc_a = Discriminator().to(device)
disc_b = Discriminator().to(device)

loader_a = torch.utils.data.DataLoader(d_windows, batch_size=32, shuffle=True, num_workers=4)
loader_b = torch.utils.data.DataLoader(d_apple, batch_size=32, shuffle=True, num_workers=4)

train_cycle(gen_ab, gen_ba, disc_a, disc_b, loader_a, loader_b,
            epochs=100, lr=0.0002, lambda_cycle=10, device=device)

```

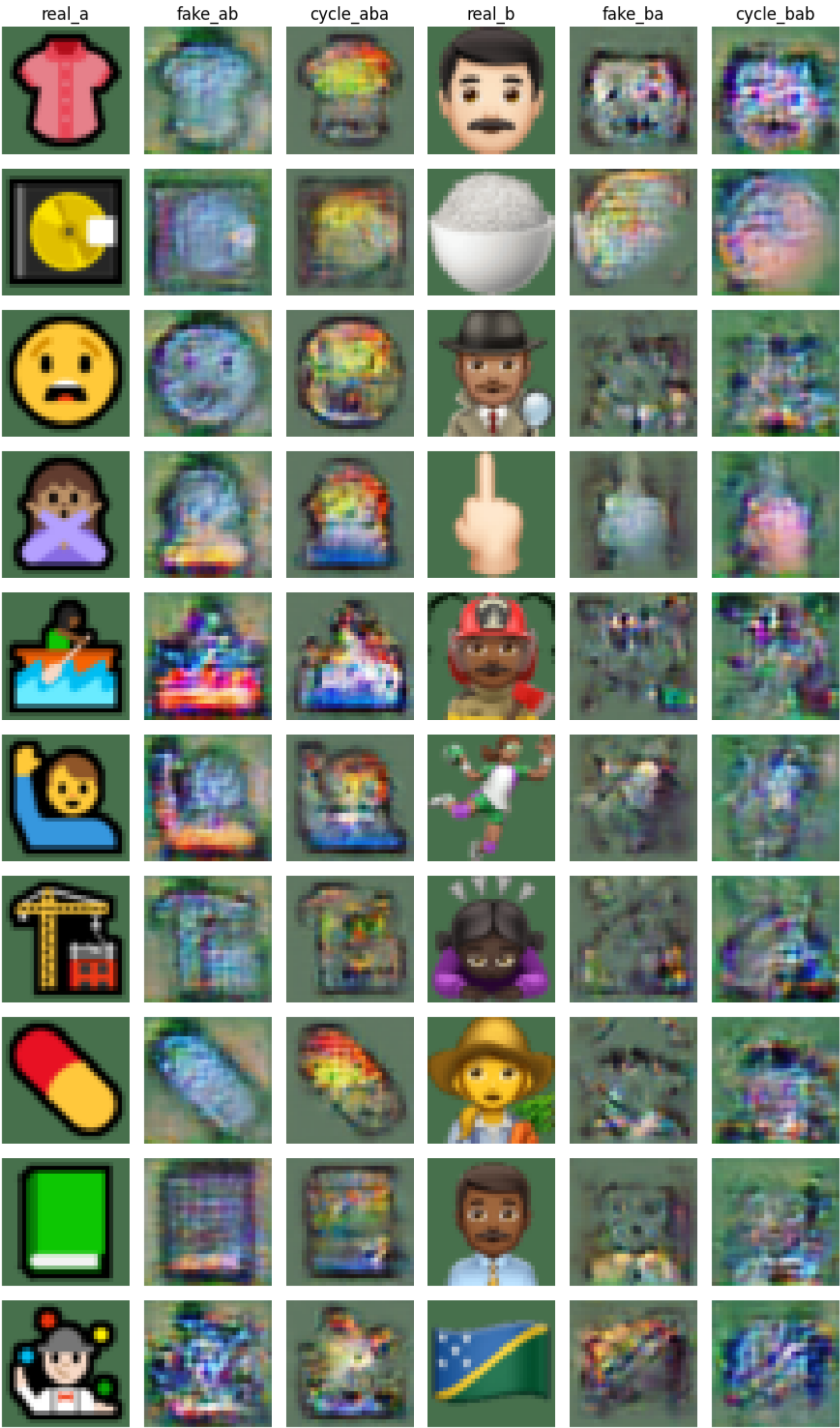

real_a	fake_ab	cycle_aba	real_b	fake_ba	cycle_bab
					
					
					
					
					
					
					
					
					
					

(b) Discuss your results and training experience. Was the model easy to train? What do you think of the results? Does it learn a good translation between Windows and Apple emojis? (2 points)









































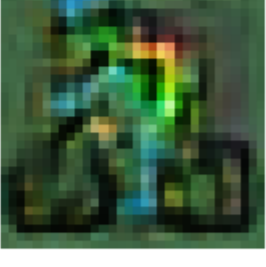
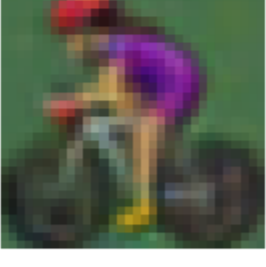

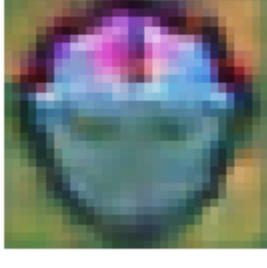

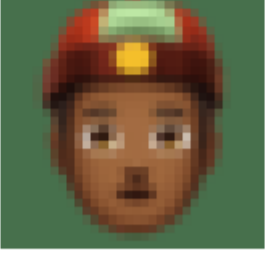

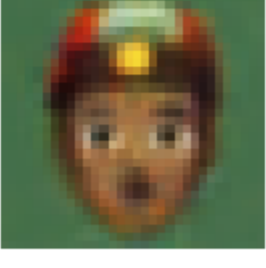








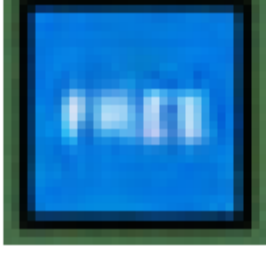


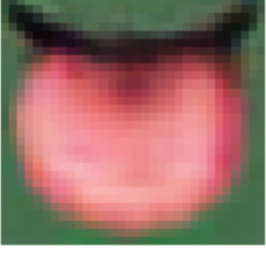
The translation between Windows and Apple emojis is not as good as we would wish, but the translation from fake images back to real ones is done really well. The model was hard to train given that with the same parameters from the paper and less complex images we weren't able to achieve the results of same quality as the paper did.

(c) Run some more experiments to study the effect of the `lambda_cycle` weight. (1 point)

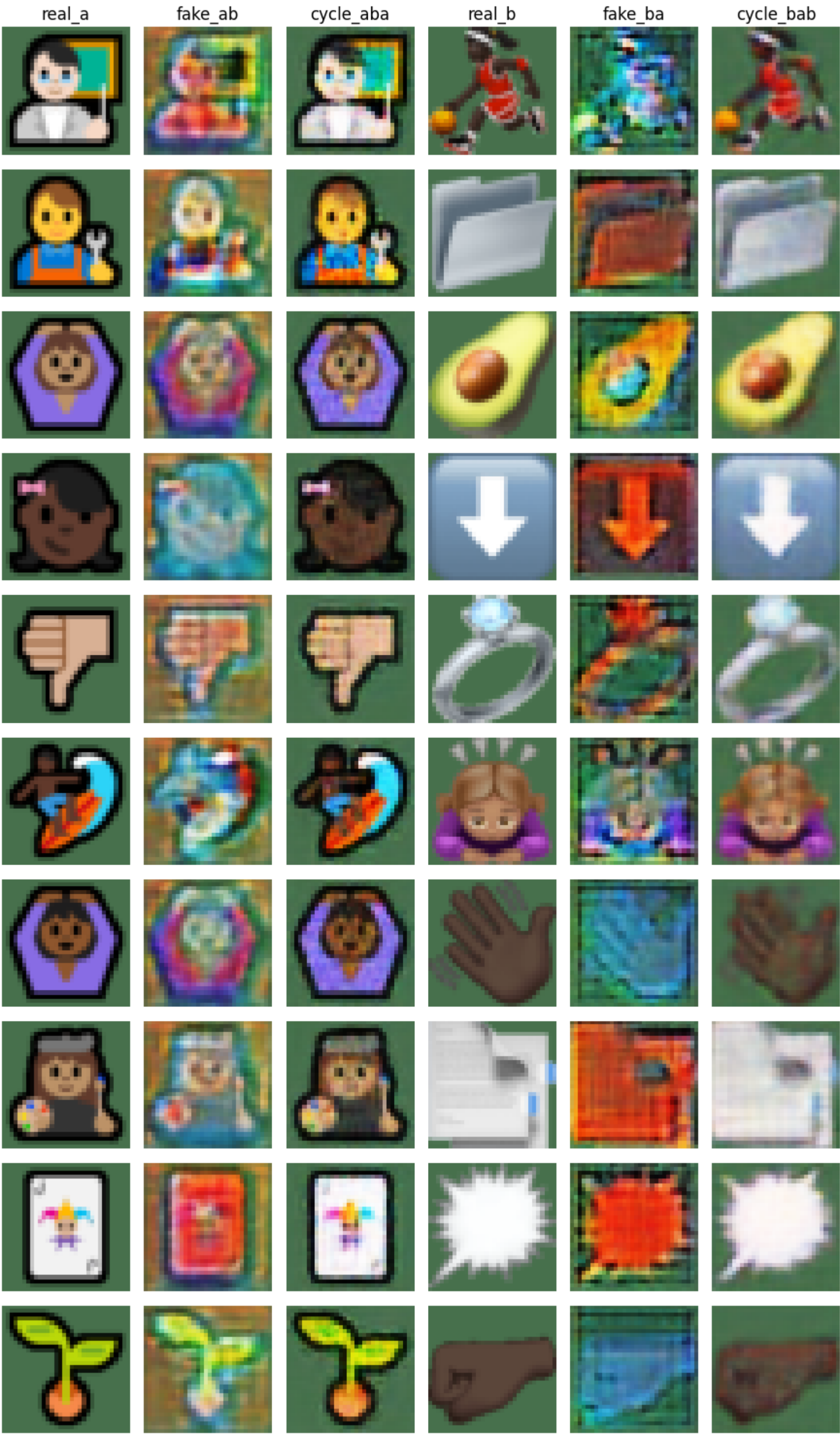
```
In [14]: train_cycle(gen_ab, gen_ba, disc_a, disc_b, loader_a, loader_b,
                  epochs=100, lr=0.0002, lambda_cycle=0, device=device)
```



```
In [13]: train_cycle(gen_ab, gen_ba, disc_a, disc_b, loader_a, loader_b,  
                    epochs=100, lr=0.0002, lambda_cycle=100, device=device)
```

real_a	fake_ab	cycle_aba	real_b	fake_ba	cycle_bab
					
					
					
					
					
					
					
					
					
					


```
In [17]: train_cycle(gen_ab, gen_ba, disc_a, disc_b, loader_a, loader_b,  
                    epochs=1000, lr=0.0002, lambda_cycle=10, device=device)
```



(d) What is the effect of the `lambda_cycle` weight? What happens if you set it to a much larger value? What happens if you set it to 0? Can you explain this? (2 points)

The value of `lambda_cycle` gives importance to cycle consistency i.e. how good of a inverse functions should G and F be. ($F(G(x)) \approx x$ and $G(F(y)) \approx y$). If we set it to a very large value the mapping A-B-A will be really good which we can see in upper plots. If we set it to 0 the generating functions will contradict each other and produce any permutation of input images if they're stochastic, therefore the A-B-A will be really bad and in turn the A-B, B-A will be bad aswell. It is crucial to pick a right value for `lambda_cycle` because if it's extremely large the model will mostly optimize for translations A-B-A and might not focus on the main task of translating A-B.

(e) Why is the reconstructed output (A->B->A or B->A->B) usually better than the translated output (A->B or B->A)? (1 point)

The reconstructed output is usually better than the translated output because of the cycle consistency constraint during training enforces a more meaningful and realistic mapping between the two domains, preventing unrealistic or distorted translations.

9.13 Final questions (4 points)

(a) Discuss how the balance between the generator and discriminator affects GAN training. What can go wrong if one part is better or learns more quickly than the other? (2 points)

If the discriminator outperforms the generator the discriminator will not be able to learn how to generate good fakes and the learning will be really slow since its hard for the generator to learn how to make good fakes.

If the generator outperforms the discriminator the generator might learn to produce only certain samples that are known to perform badly on the discriminator.

(b) CycleGAN and similar methods are unsupervised models that learn to map inputs from one domain to another. Does this mapping necessarily preserve the semantics of the images? Why, or why not? (1 point)

(For example, think about how our emoji model would translate flags.)

No, the adversarial loss has no component which accounts for semantics of the image. Primary focus of Cycle-GANs and similar models is on capturing low-level features and style transfer not semantic preservation.

(c) Have a brief look at [CycleGAN, a Master of Steganography](#), a paper published at NIPS 2017. The authors show that a CycleGAN network sometimes 'hides' information in the generated images, to help with the reconstruction. Can you see something like this in your results as well? (1 point)

We can observe that our model encodes the color of the original in the image, given that the colors of the translation is not equal in any way to the original so there must be some kind of color encoding.

The end

Well done! Please double check the instructions at the top before you submit your results.

This assignment has 30 points.

Version d7aee7b / 2023-11-09