Generative Adversarial Neural Networks

April 19, 2025

This document is the requested PDF conversion of the Jupyter Notebook, including all output cells.

Authors:

Fabio Cozzuto Student ID: 002214965

Johan Mogollon Student ID: 002359844

Contributions:

- Fabio Cozzuto: All code, experiments, and analysis
- Johan Mogollon: All code, experiments, and analysis

Course: CS551 - Deep Learning

Professor: Dr. Mohammed Ayoub Alaoui Mhamdi

1 PART 1: Deep Convolutional GAN

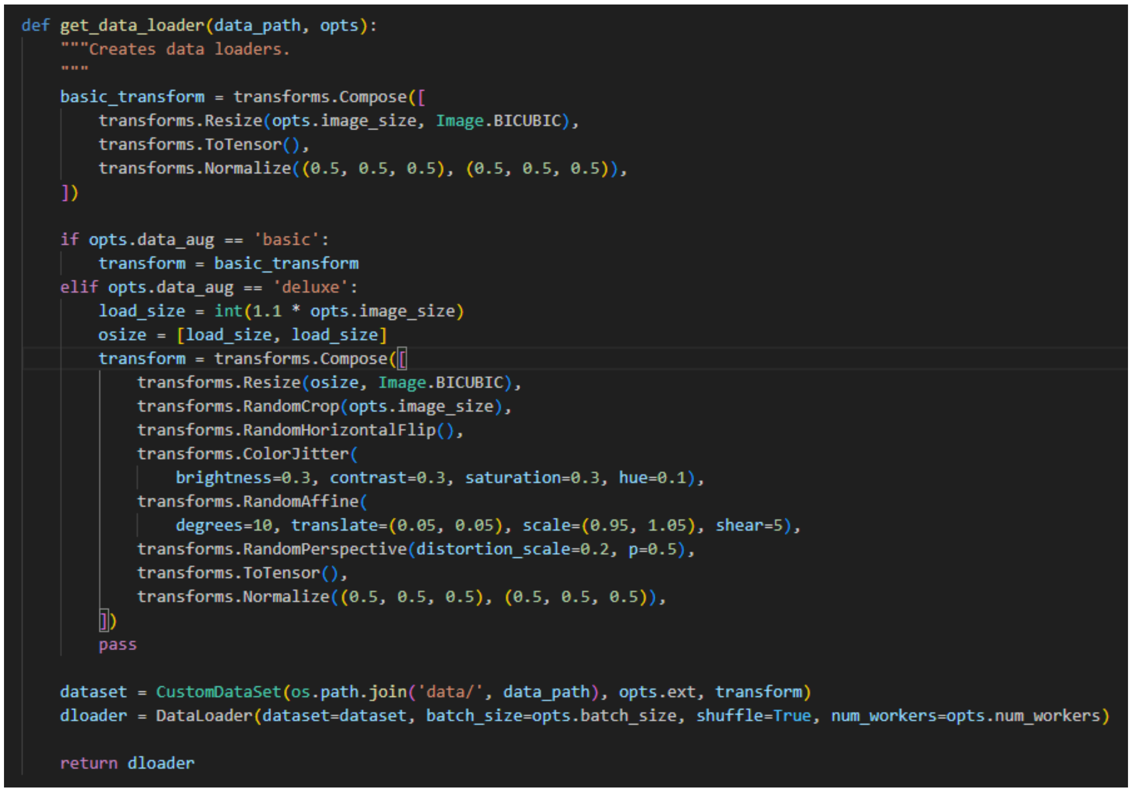
1.1 Environment Setup & Imports

[1]:
```python
# Ensure Jupyter can import our GAN modules
import os, sys
sys.path.insert(0, os.path.abspath('.'))
sys.path.append('.')

# --- Standard Libraries ---
import warnings
warnings.filterwarnings("ignore")

# --- Data Handling ---
import numpy as np
from PIL import Image
import math
np.math = math

# --- PyTorch ---
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import transforms
from torch.utils.tensorboard import SummaryWriter

# --- Local Modules ---
from data_loader import get_data_loader, CustomDataSet
from models import DCGenerator, DCDiscriminator, CycleGenerator, conv, deconv,␣
↪ResnetBlock
from utils import to_var, to_data, create_dir

# --- Visualization ---
import matplotlib.pyplot as plt
import imageio # For saving images
import matplotlib.image as mpimg

# --- Argument Parsing ---
import argparse

# --- Other ---
import glob

# Set random seed
SEED = 11
np.random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
  torch.cuda.manual_seed(SEED)

os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'

# Check for GPU availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
Using device: cuda
```

1.2 Implement Data Augmentation [10 points]

We implemented the augmentations in the following code:



```python
def get_data_loader(data_path, opts):
    """Creates data loaders.
    """
    basic_transform = transforms.Compose([
        transforms.Resize(opts.image_size, Image.BICUBIC),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])

    if opts.data_aug == 'basic':
        transform = basic_transform
    elif opts.data_aug == 'deluxe':
        load_size = int(1.1 * opts.image_size)
        osize = [load_size, load_size]
        transform = transforms.Compose([
            transforms.Resize(osize, Image.BICUBIC),
            transforms.RandomCrop(opts.image_size),
            transforms.RandomHorizontalFlip(),
            transforms.ColorJitter(
                brightness=0.3, contrast=0.3, saturation=0.3, hue=0.1),
            transforms.RandomAffine(
                degrees=10, translate=(0.05, 0.05), scale=(0.95, 1.05), shear=5),
            transforms.RandomPerspective(distortion_scale=0.2, p=0.5),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ])
        pass

    dataset = CustomDataSet(os.path.join('data/', data_path), opts.ext, transform)
    dloader = DataLoader(dataset=dataset, batch_size=opts.batch_size, shuffle=True, num_workers=opts.num_workers)

    return dloader
```

1.3 Implement the Discriminator of the DCGAN [10 points]

1.3.1 Padding Calculation for DCGAN Discriminator

Question: With kernel size (K=4) and stride (S=2), what padding (P) halves the spatial dimensions?

Answer: We want each layer to reduce the spatial dimensions by a factor of 2, without clipping important features. That means that we want to control the padding. So, we have the convolution output formula:

$$O = \left\lfloor \frac{I + 2P - K}{S} \right\rfloor + 1$$

Where: - (I) = input size - (O) = output size - (K = 4) (kernel size) - (S = 2) (stride) - (P) = padding

We want to obtain this:

$$output\_size = \frac{input\_size}{2}$$

So we solve as follows:

$$\left\lfloor \frac{I + 2P - 4}{2} \right\rfloor + 1 = \frac{I}{2} \Rightarrow 2P = 2 \Rightarrow P = 1$$

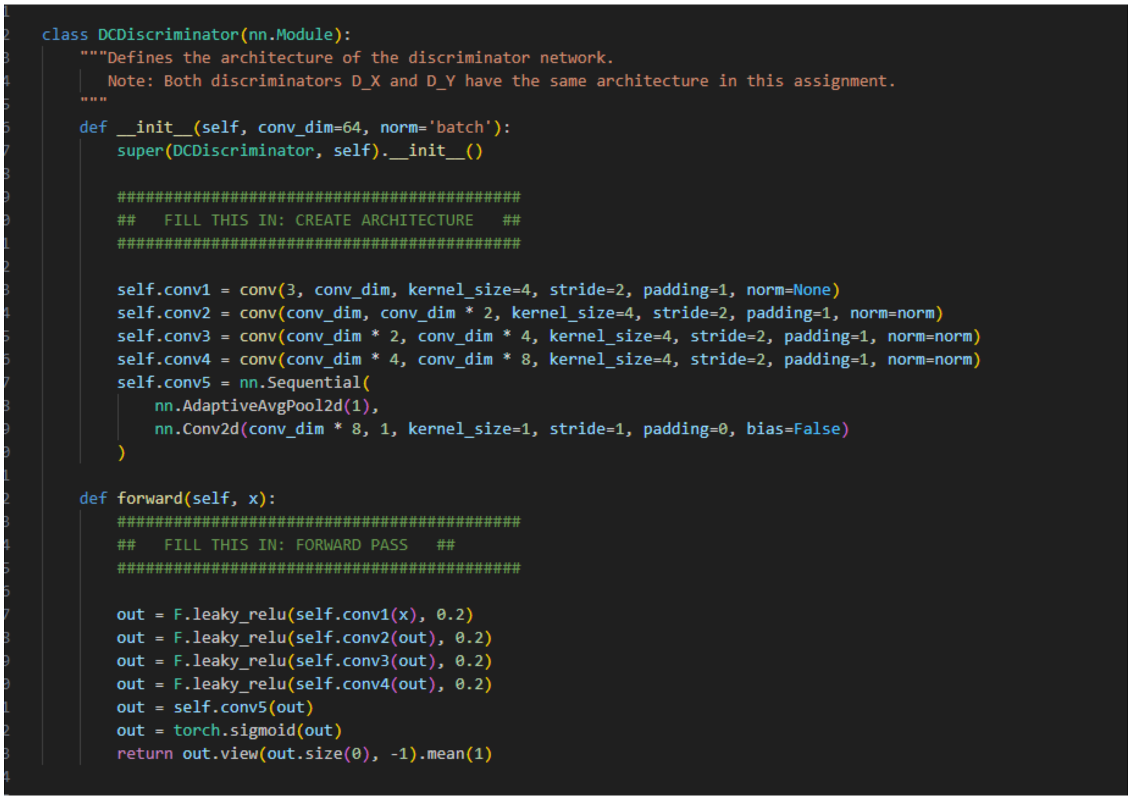
```
[ ]: # We can do the same calculations with the following code:
```

```
input_size = 64  # Example input size, this will vary per layer
kernel_size = 4
stride = 2
padding = 1
output_size = (input_size - kernel_size + 2 * padding) / stride + 1


print(f"Given kernel_size={kernel_size}, stride={stride}, the required padding␣
  ↪is: {padding}")
print(f"Example: Input size = {input_size}, Output size = {output_size}")
```

```
Given kernel_size=4, stride=2, the required padding is: 1
Example: Input size = 64, Output size = 32.0
```

1.3.2 DCDiscriminator class in the models.py file

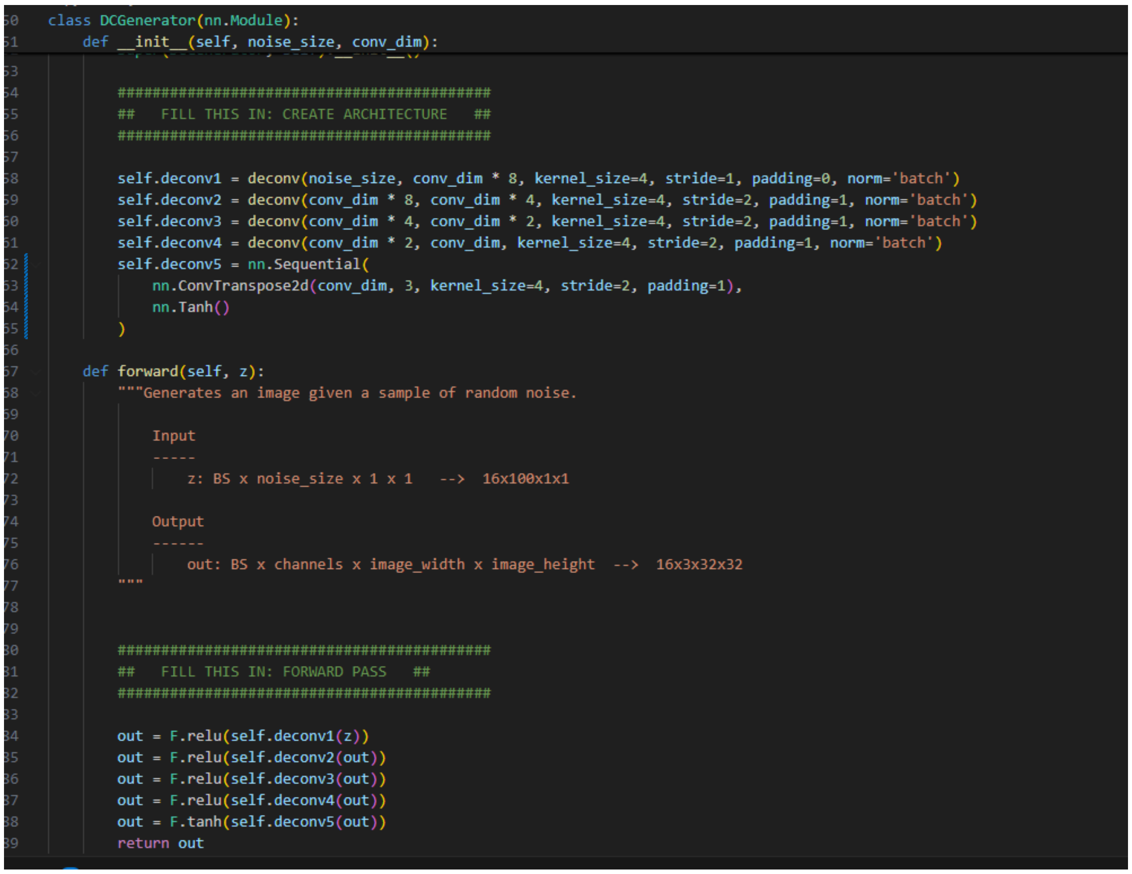
We completed the code for DCDsicriminator as you can see in the following image:



```
class DCDiscriminator(nn.Module):
    """Defines the architecture of the discriminator network.
       Note: Both discriminators D_X and D_Y have the same architecture in this assignment.
    """
    def __init__(self, conv_dim=64, norm='batch'):
        super(DCDiscriminator, self).__init__()

        ###########################################
        ##   FILL THIS IN: CREATE ARCHITECTURE   ##
        ###########################################

        self.conv1 = conv(3, conv_dim, kernel_size=4, stride=2, padding=1, norm=None)
        self.conv2 = conv(conv_dim, conv_dim * 2, kernel_size=4, stride=2, padding=1, norm=norm)
        self.conv3 = conv(conv_dim * 2, conv_dim * 4, kernel_size=4, stride=2, padding=1, norm=norm)
        self.conv4 = conv(conv_dim * 4, conv_dim * 8, kernel_size=4, stride=2, padding=1, norm=norm)
        self.conv5 = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Conv2d(conv_dim * 8, 1, kernel_size=1, stride=1, padding=0, bias=False)
        )

    def forward(self, x):
        ###########################################
        ##   FILL THIS IN: FORWARD PASS   ##
        ###########################################

        out = F.leaky_relu(self.conv1(x), 0.2)
        out = F.leaky_relu(self.conv2(out), 0.2)
        out = F.leaky_relu(self.conv3(out), 0.2)
        out = F.leaky_relu(self.conv4(out), 0.2)
        out = self.conv5(out)
        out = torch.sigmoid(out)
        return out.view(out.size(0), -1).mean(1)
```

1.4 Implement the Generator of the DCGAN [10 points]

1.4.1 DCGenerator class in the models.py file

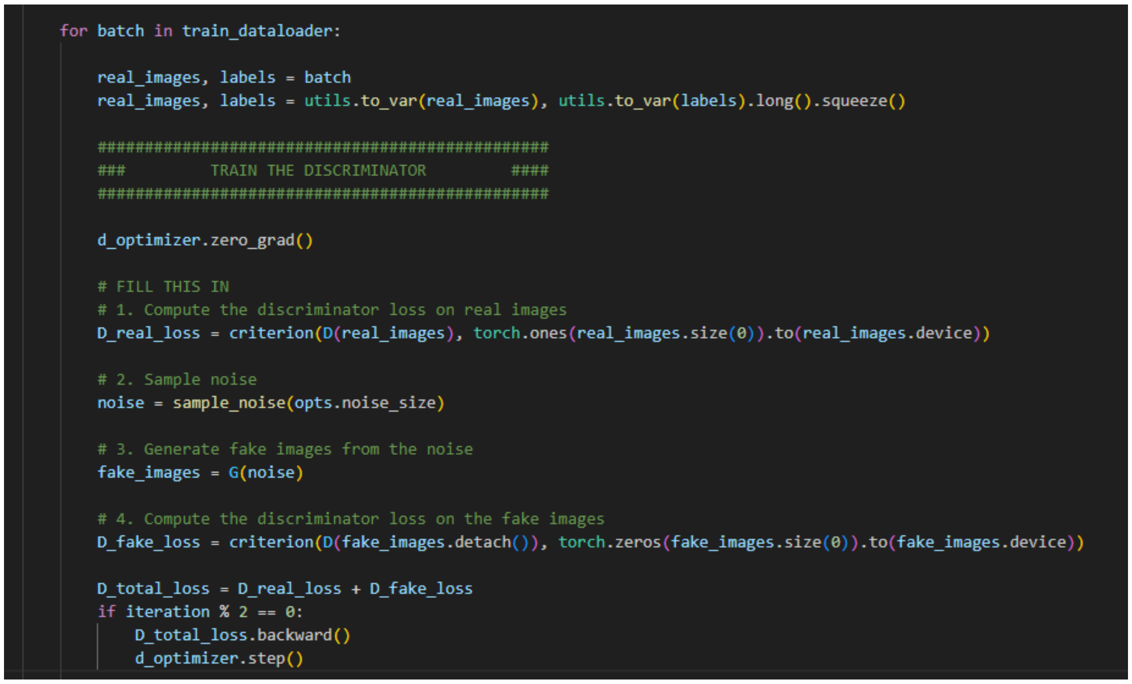


```
class DCGenerator(nn.Module):
    def __init__(self, noise_size, conv_dim):

        ###########################################
        ##   FILL THIS IN: CREATE ARCHITECTURE   ##
        ###########################################

        self.deconv1 = deconv(noise_size, conv_dim * 8, kernel_size=4, stride=1, padding=0, norm='batch')
        self.deconv2 = deconv(conv_dim * 8, conv_dim * 4, kernel_size=4, stride=2, padding=1, norm='batch')
        self.deconv3 = deconv(conv_dim * 4, conv_dim * 2, kernel_size=4, stride=2, padding=1, norm='batch')
        self.deconv4 = deconv(conv_dim * 2, conv_dim, kernel_size=4, stride=2, padding=1, norm='batch')
        self.deconv5 = nn.Sequential(
            nn.ConvTranspose2d(conv_dim, 3, kernel_size=4, stride=2, padding=1),
            nn.Tanh()
        )

    def forward(self, z):
        """Generates an image given a sample of random noise.

            Input
            -----
                z: BS x noise_size x 1 x 1   -->  16x100x1x1

            Output
            ------
                out: BS x channels x image_width x image_height  -->  16x3x32x32
        """


        ###########################################
        ##   FILL THIS IN: FORWARD PASS   ##
        ###########################################

        out = F.relu(self.deconv1(z))
        out = F.relu(self.deconv2(out))
        out = F.relu(self.deconv3(out))
        out = F.relu(self.deconv4(out))
        out = F.tanh(self.deconv5(out))
        return out
```

1.5 Experiments

1.5.1 Implement the DCGAN Training Loop [10 points]

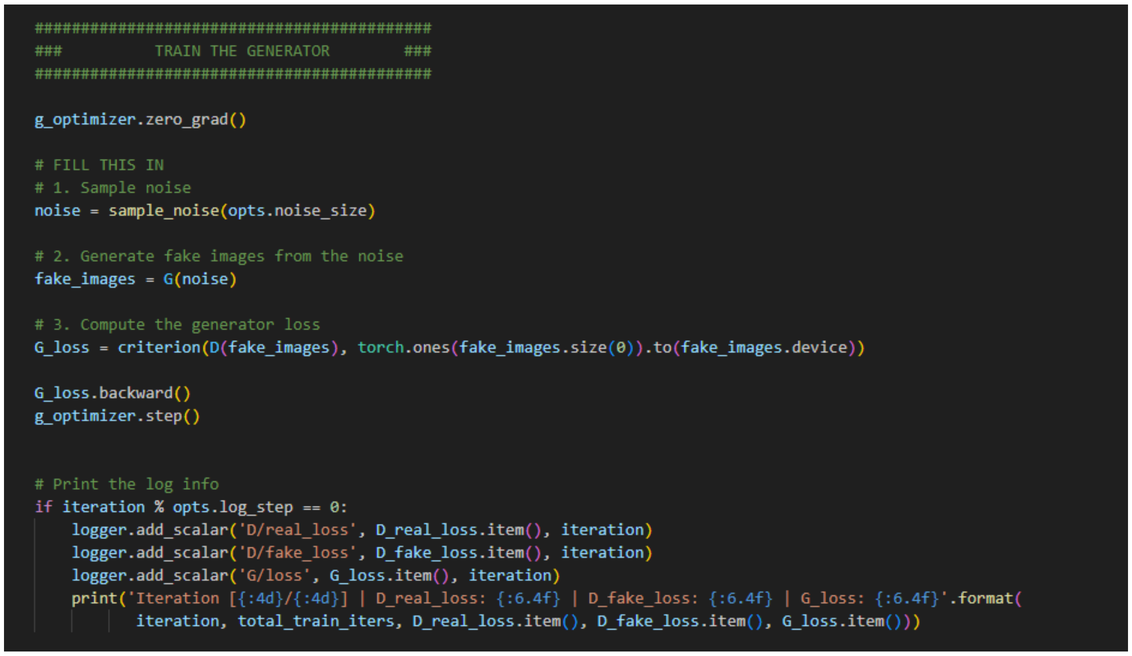
Discriminator



```
for batch in train_dataloader:

    real_images, labels = batch
    real_images, labels = utils.to_var(real_images), utils.to_var(labels).long().squeeze()

    ##################################################
    ###          TRAIN THE DISCRIMINATOR         ####
    ##################################################

    d_optimizer.zero_grad()

    # FILL THIS IN
    # 1. Compute the discriminator loss on real images
    D_real_loss = criterion(D(real_images), torch.ones(real_images.size(0)).to(real_images.device))

    # 2. Sample noise
    noise = sample_noise(opts.noise_size)

    # 3. Generate fake images from the noise
    fake_images = G(noise)

    # 4. Compute the discriminator loss on the fake images
    D_fake_loss = criterion(D(fake_images.detach()), torch.zeros(fake_images.size(0)).to(fake_images.device))

    D_total_loss = D_real_loss + D_fake_loss
    if iteration % 2 == 0:
        D_total_loss.backward()
        d_optimizer.step()
```

Generator

For this part we filled the code and carefully add the logger to work with TensorBoard



```
###########################################
###          TRAIN THE GENERATOR        ###
###########################################

g_optimizer.zero_grad()

# FILL THIS IN
# 1. Sample noise
noise = sample_noise(opts.noise_size)

# 2. Generate fake images from the noise
fake_images = G(noise)

# 3. Compute the generator loss
G_loss = criterion(D(fake_images), torch.ones(fake_images.size(0)).to(fake_images.device))

G_loss.backward()
g_optimizer.step()


# Print the log info
if iteration % opts.log_step == 0:
    logger.add_scalar('D/real_loss', D_real_loss.item(), iteration)
    logger.add_scalar('D/fake_loss', D_fake_loss.item(), iteration)
    logger.add_scalar('G/loss', G_loss.item(), iteration)
    print('Iteration [{:4d}/{:4d}] | D_real_loss: {:6.4f} | D_fake_loss: {:6.4f} | G_loss: {:6.4f}'.format(
            iteration, total_train_iters, D_real_loss.item(), D_fake_loss.item(), G_loss.item()))
```

1.5.2 Train the DCGAN [10 points]

The following code traiin the DCGAN, so this is the first execution we can do to understand the model and to see if there is some kind of error.

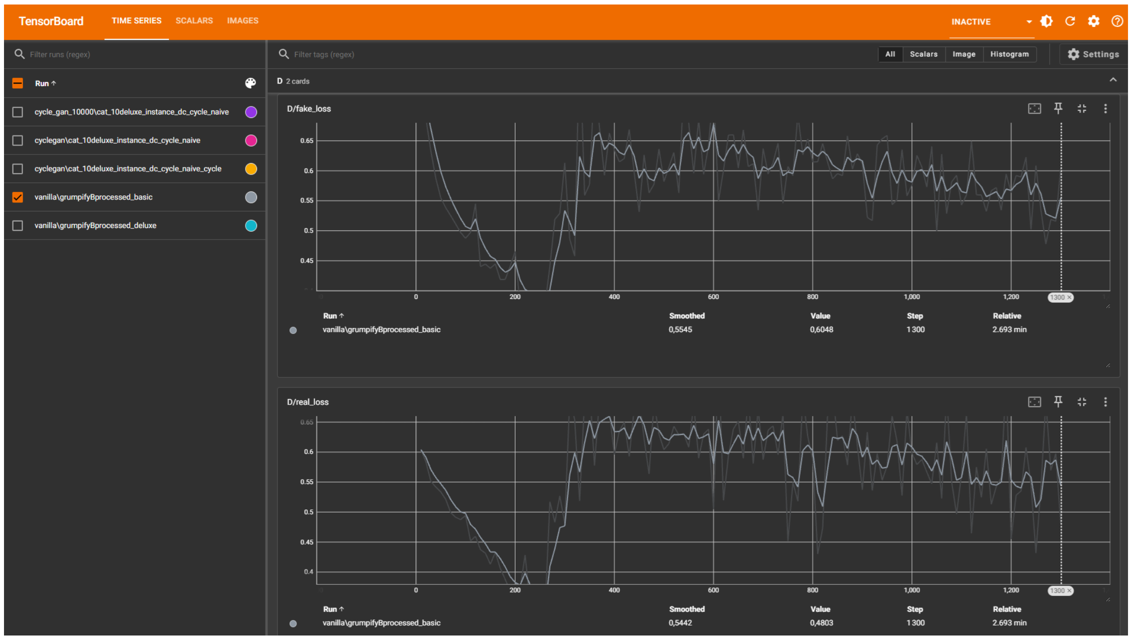
```
[ ]: !python vanilla_gan.py --num_epochs=100
```

Basic Execution

```
[ ]: !python vanilla_gan.py --data_aug=basic  --num_epochs=100
```

Basic Loss Curves On the Generator losses we can see an increase trend over the training steps. This indicates that the discriminator is getting better at separating false images from real ones, making it more difficult for the generator to "fool" it. This increase in loss may suggest that the generator requires additional effort to achieve good images, and that the discriminator is indeed getting better, outperforming the generator. However, it may also indicate that the generator is not performing as well as the training progresses. Looking at the generated images is important to understand which scenario is happening.

The discriminator losses show a decreasing trend in both false images and real images. This indicates that, as training progresses, the discriminator fails to differentiate between false and real images. Now, it is possible to expect that D/false_loss is reduced as the generator gets closer to producing better images, in the same way that a reduction in D/real_loss can be seen. Both reductions would be indications that the generator is managing to "fool" the discriminator, however, it could also indicate that there is a learning problem and that therefore the discriminator is losing the ability to effectively differentiate between the two groups of images.



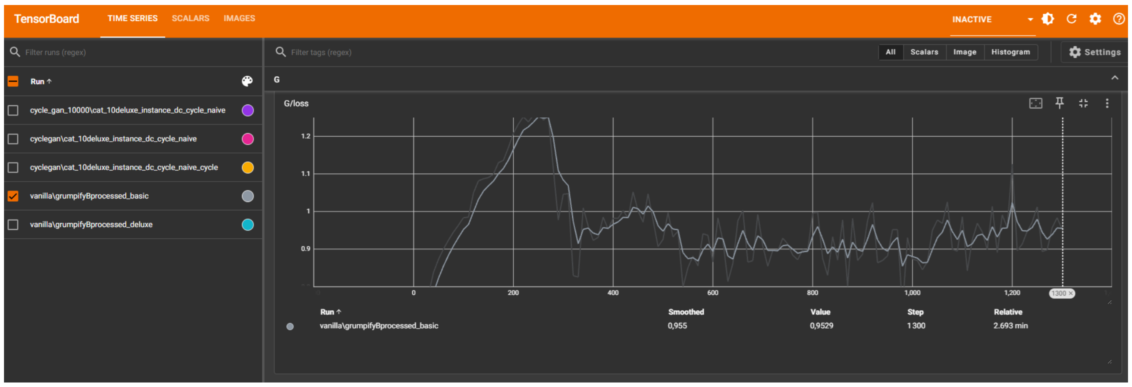
	Run ↑
☐	cycle_gan_10000\cat_10deluxe_instance_dc_cycle_naive
☐	cyclegan\cat_10deluxe_instance_dc_cycle_naive
☐	cyclegan\cat_10deluxe_instance_dc_cycle_naive_cycle
✓	vanilla\grumpifyBprocessed_basic
☐	vanilla\grumpifyBprocessed_deluxe

D/fake_loss

Run ↑	Smoothed	Value	Step	Relative
vanilla\grumpifyBprocessed_basic	0,5545	0,6048	1300	2.693 min

D/real_loss

Run ↑	Smoothed	Value	Step	Relative
vanilla\grumpifyBprocessed_basic	0,5442	0,4803	1300	2.693 min



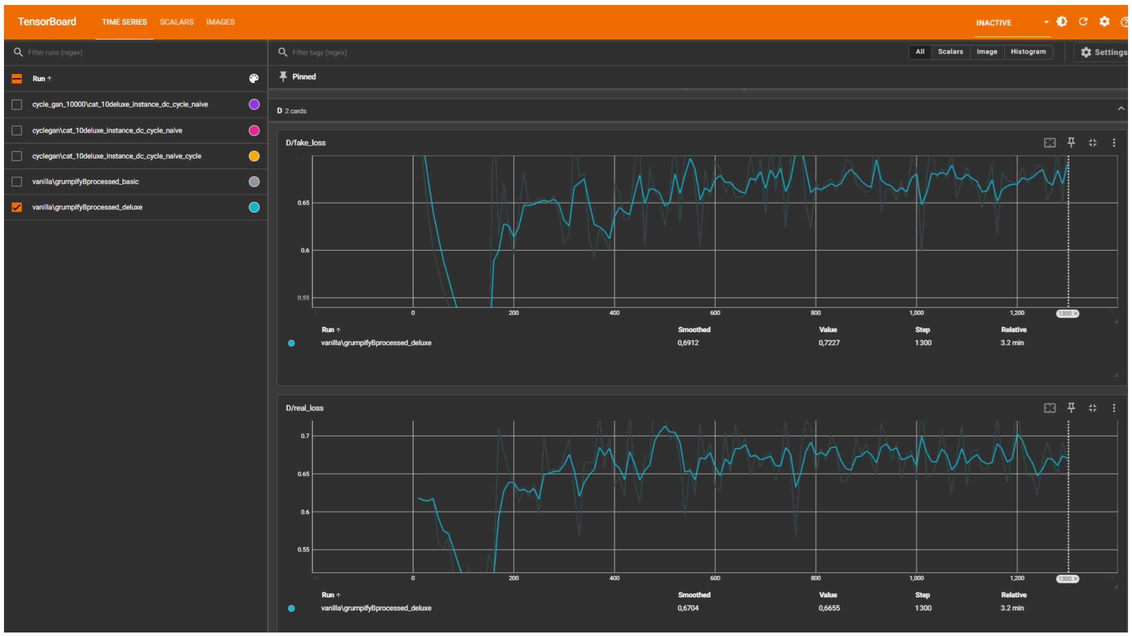
Run ↑	
☐	cycle_gan_10000\cat_10deluxe_instance_dc_cycle_naive
☐	cyclegan\cat_10deluxe_instance_dc_cycle_naive
☐	cyclegan\cat_10deluxe_instance_dc_cycle_naive_cycle
✓	vanilla\grumpify8processed_basic
☐	vanilla\grumpify8processed_deluxe

Run ↑	Smoothed	Value	Step	Relative
vanilla\grumpify8processed_basic	0,955	0,9529	1300	2.693 min

Deluxe Execution

```
[ ]: !python vanilla_gan.py --data_aug=deluxe  --num_epochs=100
```

Deluxe Loss Curves The curve D/false_loss starts high and decreases as the training progresses, showing some fluctuations during the process. Again, at the beginning, the discriminator manages to identify the generated images as false, but its accuracy decreases as the training advances, suggesting that the generator improves its performance. Now, the fluctuations that we see could suggest moments when the discriminator adapts to the strategies of the generator to produce more realistic false images. The same way D/real_loss shows a dropping tendency from a high starting point, this suggests that the performance of the discriminator in trying to classify images decreases over the training process. The presence of the augmentations is an important component for this to happen, as the discriminator learns to identify real images even under various transformations, but the decreasing loss indicates that the generator also improves. The fluctuations could represent the Discriminator continued attempts to learn features and correctly identify images.



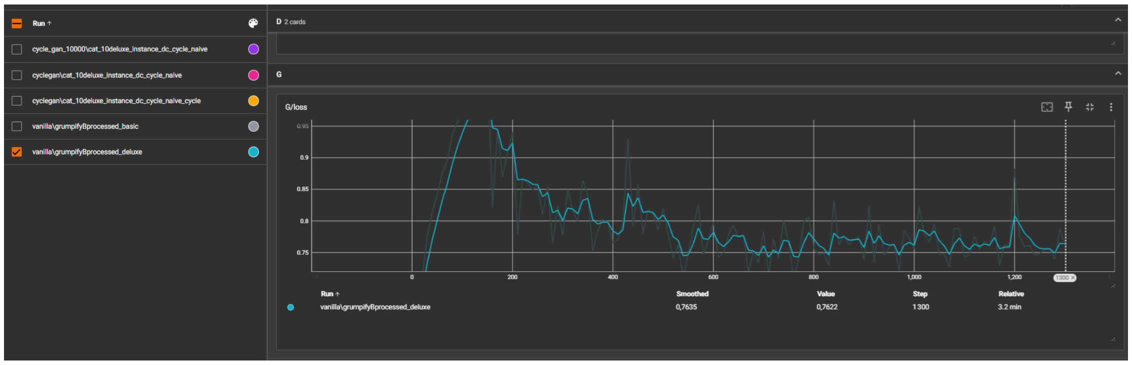
Run ↑	
☐	cycle_gan_10000\cat_10deluxe_instance_dc_cycle_naive
☐	cyclegan\cat_10deluxe_instance_dc_cycle_naive
☐	cyclegan\cat_10deluxe_instance_dc_cycle_naive_cycle
☐	vanilla\grumpify8processed_basic
✓	vanilla\grumpify8processed_deluxe

D/fake_loss

Run ↑	Smoothed	Value	Step	Relative
vanilla\grumpify8processed_deluxe	0,6912	0,7227	1300	3.2 min

D/real_loss

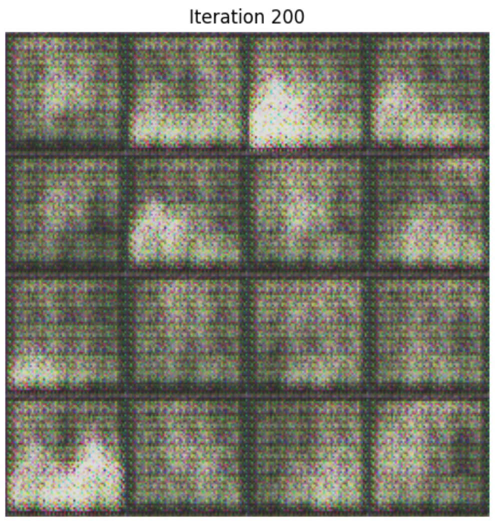
Run ↑	Smoothed	Value	Step	Relative
vanilla\grumpify8processed_deluxe	0,6704	0,6655	1300	3.2 min

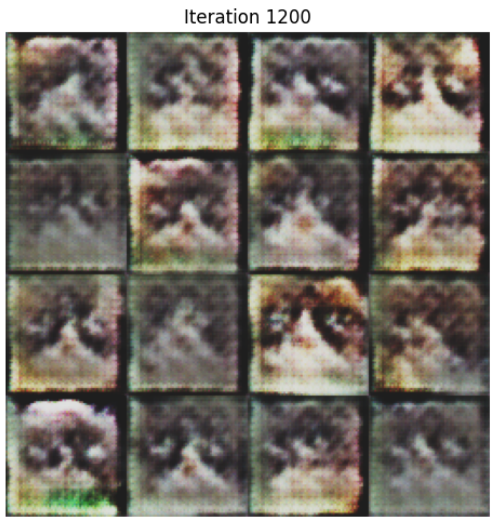


	Run ↑	
☐	cycle_gan_10000\cat_10deluxe_instance_dc_cycle_naive	●
☐	cyclegan\cat_10deluxe_instance_dc_cycle_naive	●
☐	cyclegan\cat_10deluxe_instance_dc_cycle_naive_cycle	●
☐	vanilla\grumpifyBprocessed_basic	●
✓	vanilla\grumpifyBprocessed_deluxe	●

	Run ↑	Smoothed	Value	Step	Relative
●	vanilla\grumpifyBprocessed_deluxe	0,7635	0,7622	1300	3.2 min

[13]:
```python
# Load images from specific iterations
img_early = mpimg.imread("output/vanilla/grumpifyBprocessed_deluxe/
 ↪sample-000200.png")
img_late = mpimg.imread("output/vanilla/grumpifyBprocessed_deluxe/sample-001200.
 ↪png")

# Plot the images for comparison
plt.figure(figsize=(12, 5))

# Plot early stage
plt.subplot(1, 2, 1)
plt.imshow(img_early)
plt.title("Iteration 200")
plt.axis('off')

# Plot late stage
plt.subplot(1, 2, 2)
plt.imshow(img_late)
plt.title("Iteration 1200")
plt.axis('off')

plt.tight_layout()
plt.show()
```

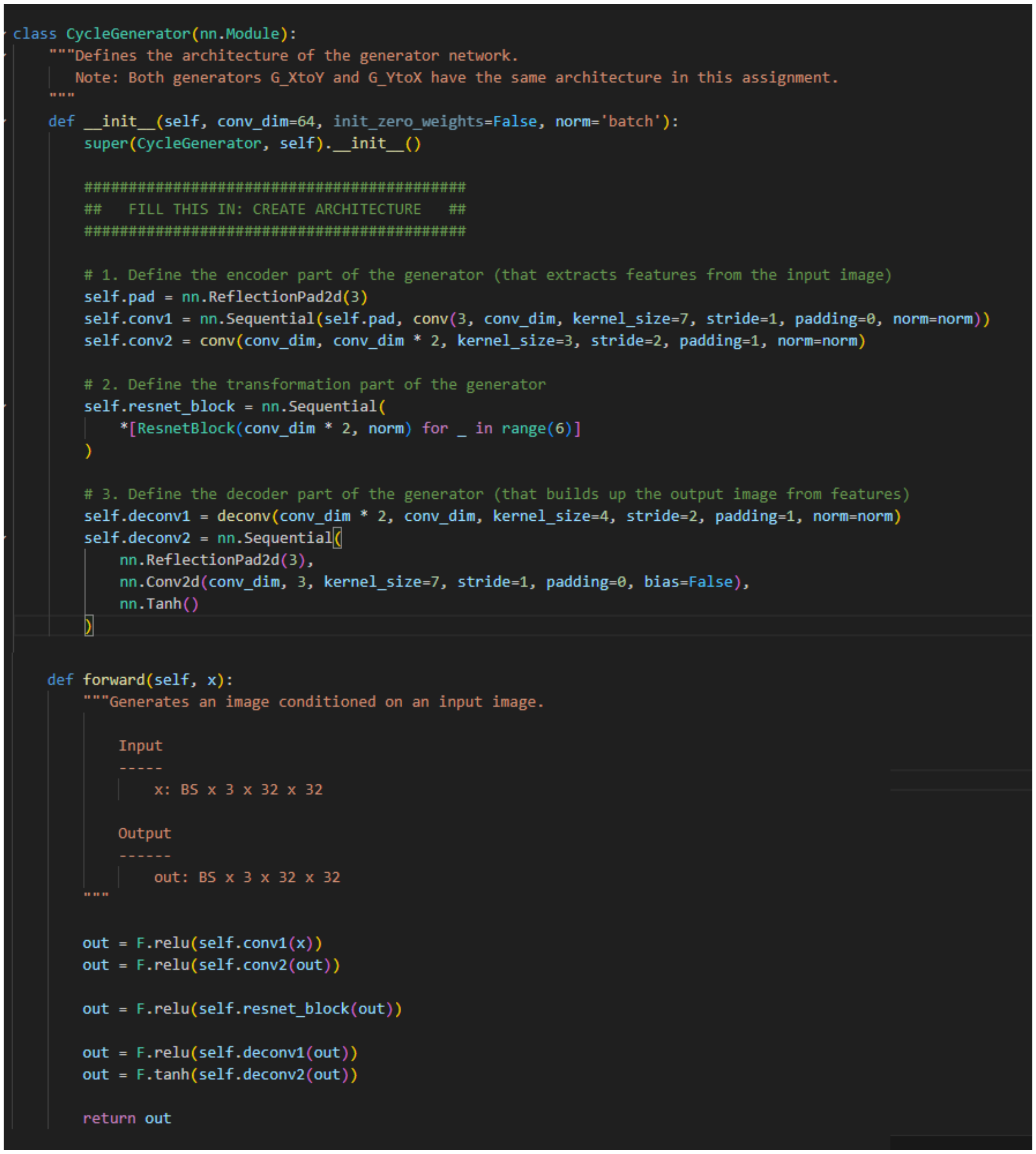




In this comparison of images that Vanilla GAN generated, we can see that at the beginning, around step 200, the images are just messy noise and do not look like anything specific, not even a cat. This is normal because the network that creates the images is just starting to learn and being as basic as Vanilla_Gan is, it may not be as fast in generating good results from the beginning. However, if we look at step 1200, the images improve a bit. However, they are still blurry and although you can see some shapes and colors that maybe look a bit like cats they are not of the best quality. It is as if the network is slowly realizing what a grumpy cat looks like, but it is still not very clear or real, it is clear at this stage that the network requires many more steps to learn how to create cat images.

2 PART 2: CycleGAN

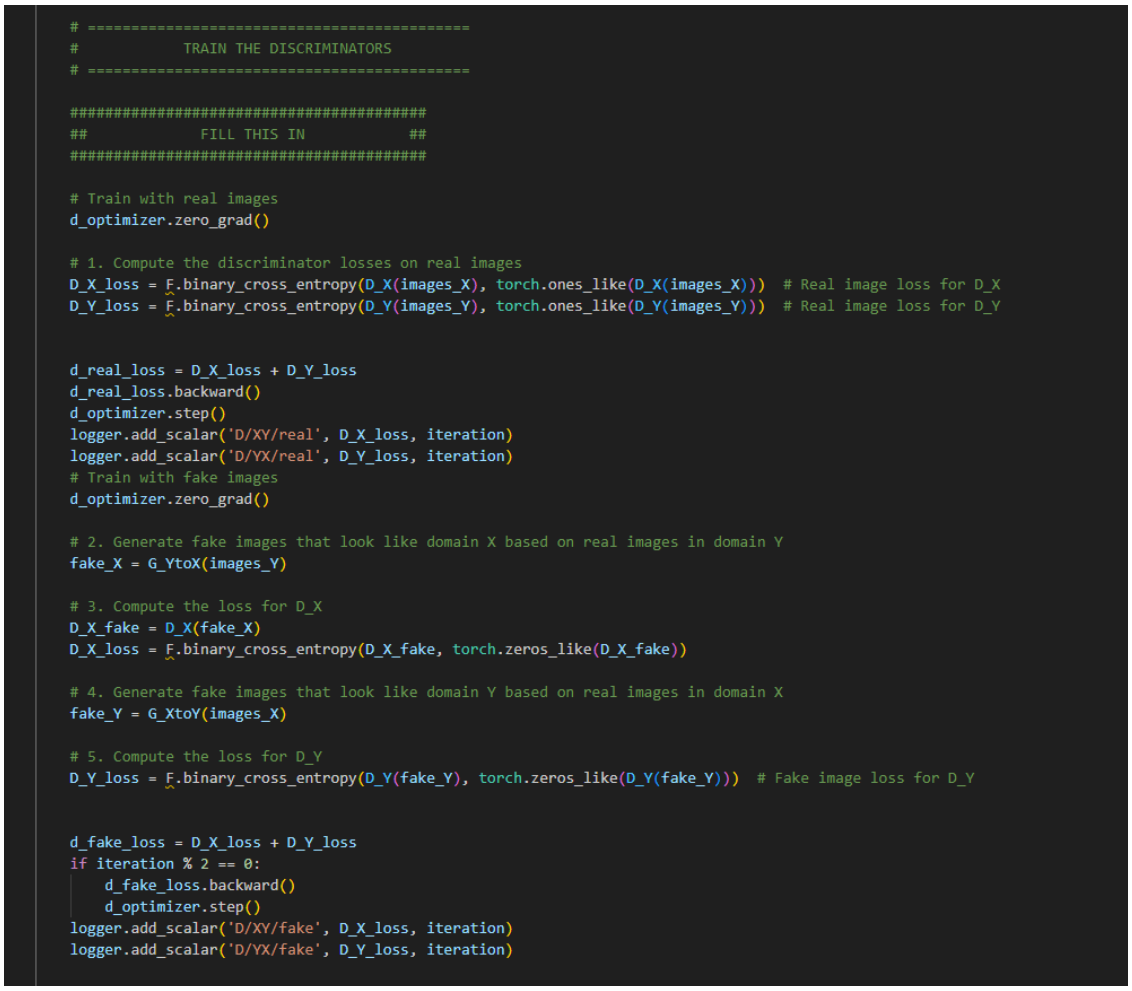
2.1 Generator [20 points]



```
class CycleGenerator(nn.Module):
    """Defines the architecture of the generator network.
       Note: Both generators G_XtoY and G_YtoX have the same architecture in this assignment.
    """
    def __init__(self, conv_dim=64, init_zero_weights=False, norm='batch'):
        super(CycleGenerator, self).__init__()

        ###########################################
        ##   FILL THIS IN: CREATE ARCHITECTURE   ##
        ###########################################

        # 1. Define the encoder part of the generator (that extracts features from the input image)
        self.pad = nn.ReflectionPad2d(3)
        self.conv1 = nn.Sequential(self.pad, conv(3, conv_dim, kernel_size=7, stride=1, padding=0, norm=norm))
        self.conv2 = conv(conv_dim, conv_dim * 2, kernel_size=3, stride=2, padding=1, norm=norm)

        # 2. Define the transformation part of the generator
        self.resnet_block = nn.Sequential(
            *[ResnetBlock(conv_dim * 2, norm) for _ in range(6)]
        )

        # 3. Define the decoder part of the generator (that builds up the output image from features)
        self.deconv1 = deconv(conv_dim * 2, conv_dim, kernel_size=4, stride=2, padding=1, norm=norm)
        self.deconv2 = nn.Sequential(
            nn.ReflectionPad2d(3),
            nn.Conv2d(conv_dim, 3, kernel_size=7, stride=1, padding=0, bias=False),
            nn.Tanh()
        )


    def forward(self, x):
        """Generates an image conditioned on an input image.

            Input
            -----
                x: BS x 3 x 32 x 32

            Output
            ------
                out: BS x 3 x 32 x 32
        """

        out = F.relu(self.conv1(x))
        out = F.relu(self.conv2(out))

        out = F.relu(self.resnet_block(out))

        out = F.relu(self.deconv1(out))
        out = F.tanh(self.deconv2(out))

        return out
```

2.2 CycleGAN Training Loop [20 points]

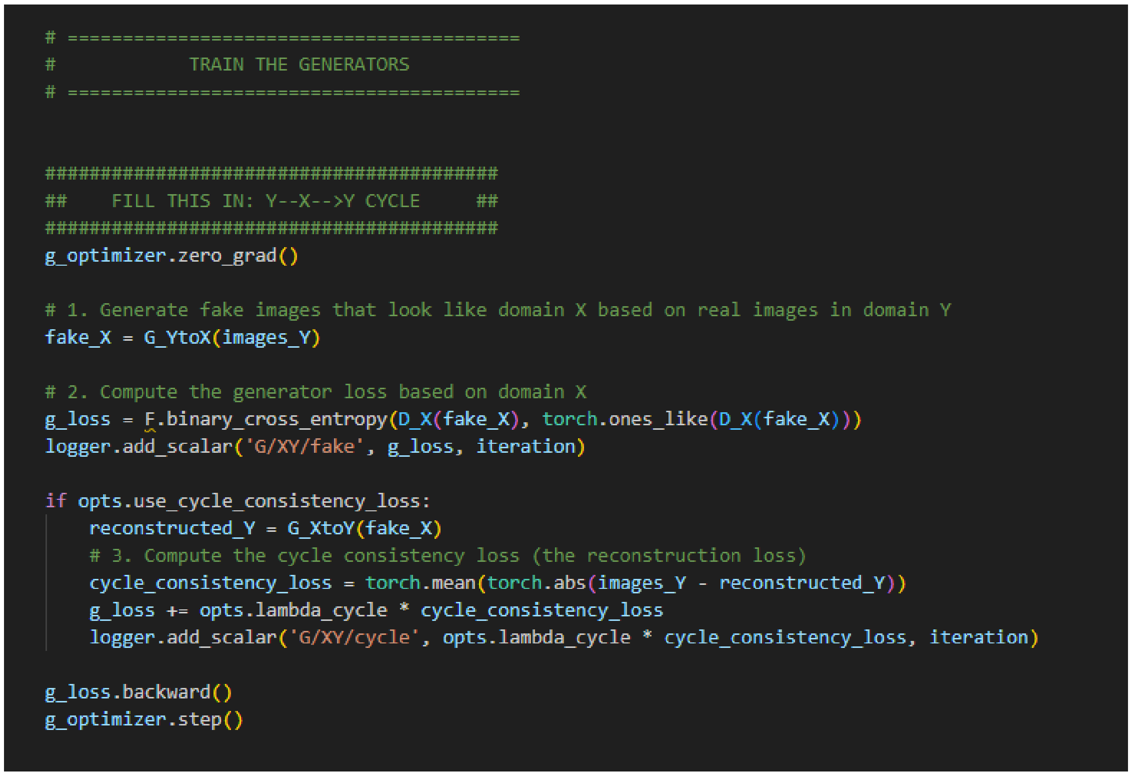
2.2.1 Discriminator



```
# ============================================
#            TRAIN THE DISCRIMINATORS
# ============================================

#########################################
##             FILL THIS IN            ##
#########################################

# Train with real images
d_optimizer.zero_grad()

# 1. Compute the discriminator losses on real images
D_X_loss = F.binary_cross_entropy(D_X(images_X), torch.ones_like(D_X(images_X)))  # Real image loss for D_X
D_Y_loss = F.binary_cross_entropy(D_Y(images_Y), torch.ones_like(D_Y(images_Y)))  # Real image loss for D_Y


d_real_loss = D_X_loss + D_Y_loss
d_real_loss.backward()
d_optimizer.step()
logger.add_scalar('D/XY/real', D_X_loss, iteration)
logger.add_scalar('D/YX/real', D_Y_loss, iteration)
# Train with fake images
d_optimizer.zero_grad()

# 2. Generate fake images that look like domain X based on real images in domain Y
fake_X = G_YtoX(images_Y)

# 3. Compute the loss for D_X
D_X_fake = D_X(fake_X)
D_X_loss = F.binary_cross_entropy(D_X_fake, torch.zeros_like(D_X_fake))

# 4. Generate fake images that look like domain Y based on real images in domain X
fake_Y = G_XtoY(images_X)

# 5. Compute the loss for D_Y
D_Y_loss = F.binary_cross_entropy(D_Y(fake_Y), torch.zeros_like(D_Y(fake_Y)))  # Fake image loss for D_Y


d_fake_loss = D_X_loss + D_Y_loss
if iteration % 2 == 0:
    d_fake_loss.backward()
    d_optimizer.step()
logger.add_scalar('D/XY/fake', D_X_loss, iteration)
logger.add_scalar('D/YX/fake', D_Y_loss, iteration)
```

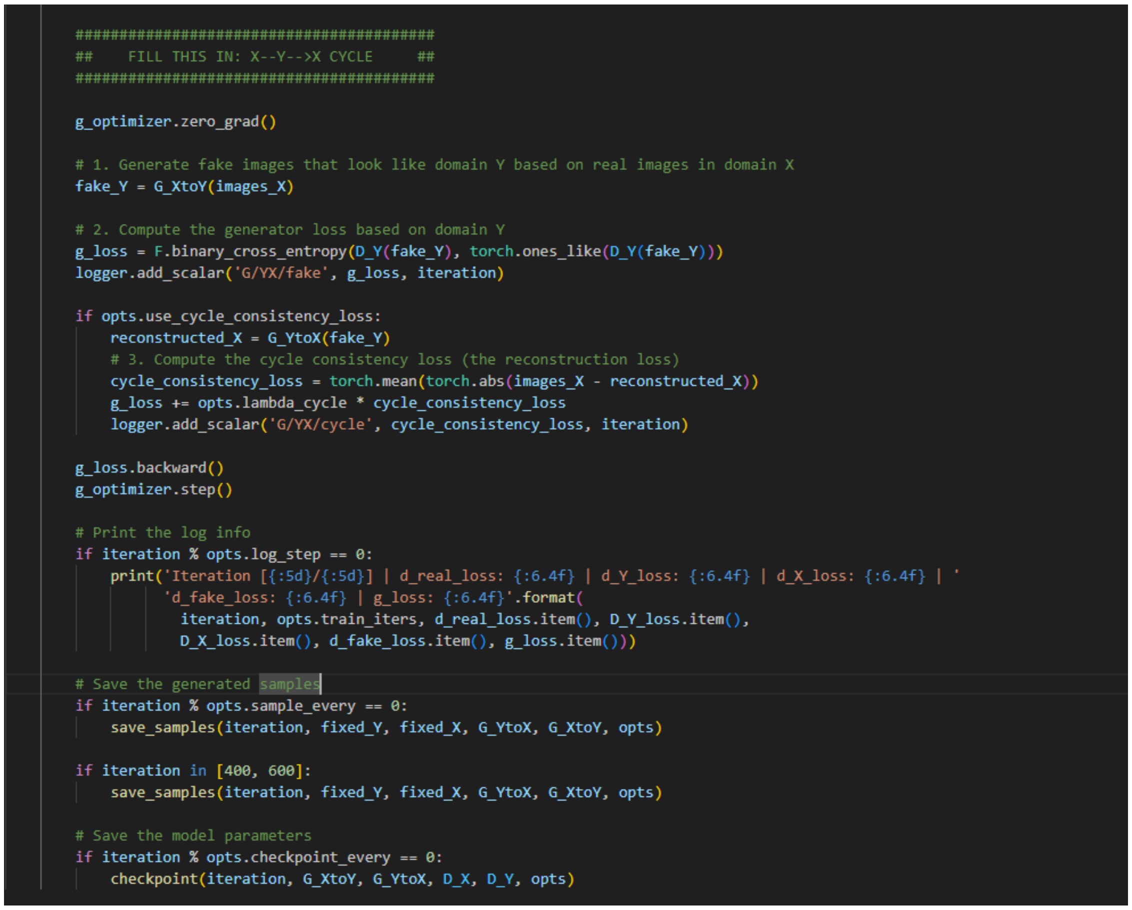
2.2.2 Generator Y–X–>Y CYCLE

We are going to use L1 loss, as suggested in the original paper.



```
# ========================================
#            TRAIN THE GENERATORS
# ========================================


#########################################
##    FILL THIS IN: Y--X-->Y CYCLE     ##
#########################################
g_optimizer.zero_grad()

# 1. Generate fake images that look like domain X based on real images in domain Y
fake_X = G_YtoX(images_Y)

# 2. Compute the generator loss based on domain X
g_loss = F.binary_cross_entropy(D_X(fake_X), torch.ones_like(D_X(fake_X)))
logger.add_scalar('G/XY/fake', g_loss, iteration)

if opts.use_cycle_consistency_loss:
    reconstructed_Y = G_XtoY(fake_X)
    # 3. Compute the cycle consistency loss (the reconstruction loss)
    cycle_consistency_loss = torch.mean(torch.abs(images_Y - reconstructed_Y))
    g_loss += opts.lambda_cycle * cycle_consistency_loss
    logger.add_scalar('G/XY/cycle', opts.lambda_cycle * cycle_consistency_loss, iteration)

g_loss.backward()
g_optimizer.step()
```

2.2.3 Generator X–Y–>X CYCLE



```
##########################################
##    FILL THIS IN: X--Y-->X CYCLE     ##
##########################################

g_optimizer.zero_grad()

# 1. Generate fake images that look like domain Y based on real images in domain X
fake_Y = G_XtoY(images_X)

# 2. Compute the generator loss based on domain Y
g_loss = F.binary_cross_entropy(D_Y(fake_Y), torch.ones_like(D_Y(fake_Y)))
logger.add_scalar('G/YX/fake', g_loss, iteration)

if opts.use_cycle_consistency_loss:
    reconstructed_X = G_YtoX(fake_Y)
    # 3. Compute the cycle consistency loss (the reconstruction loss)
    cycle_consistency_loss = torch.mean(torch.abs(images_X - reconstructed_X))
    g_loss += opts.lambda_cycle * cycle_consistency_loss
    logger.add_scalar('G/YX/cycle', cycle_consistency_loss, iteration)

g_loss.backward()
g_optimizer.step()

# Print the log info
if iteration % opts.log_step == 0:
    print('Iteration [{:5d}/{:5d}] | d_real_loss: {:6.4f} | d_Y_loss: {:6.4f} | d_X_loss: {:6.4f} | '
          'd_fake_loss: {:6.4f} | g_loss: {:6.4f}'.format(
            iteration, opts.train_iters, d_real_loss.item(), D_Y_loss.item(),
            D_X_loss.item(), d_fake_loss.item(), g_loss.item()))

# Save the generated samples
if iteration % opts.sample_every == 0:
    save_samples(iteration, fixed_Y, fixed_X, G_YtoX, G_XtoY, opts)

if iteration in [400, 600]:
    save_samples(iteration, fixed_Y, fixed_X, G_YtoX, G_XtoY, opts)

# Save the model parameters
if iteration % opts.checkpoint_every == 0:
    checkpoint(iteration, G_XtoY, G_YtoX, D_X, D_Y, opts)
```

2.3 CycleGAN Experiments [15 points]

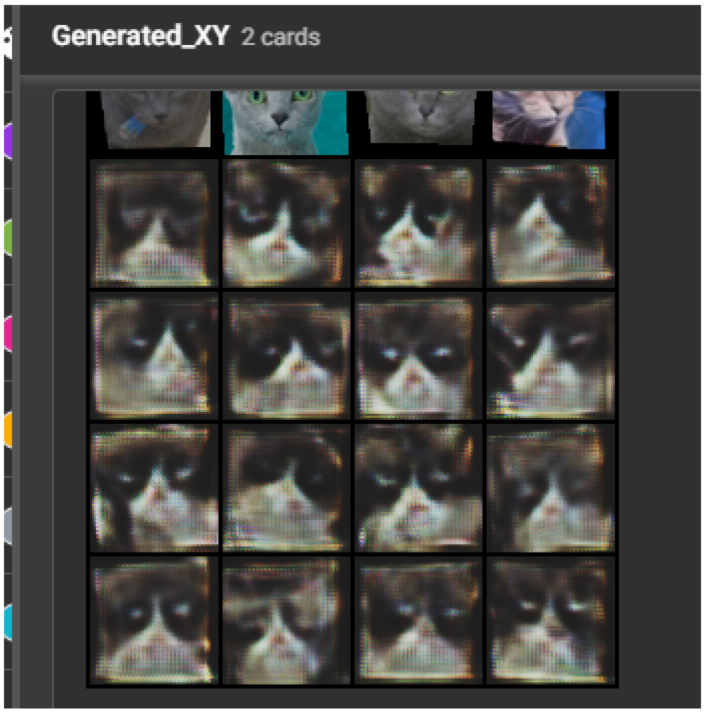
Basic Execution

```
[ ]: !python cycle_gan.py
```

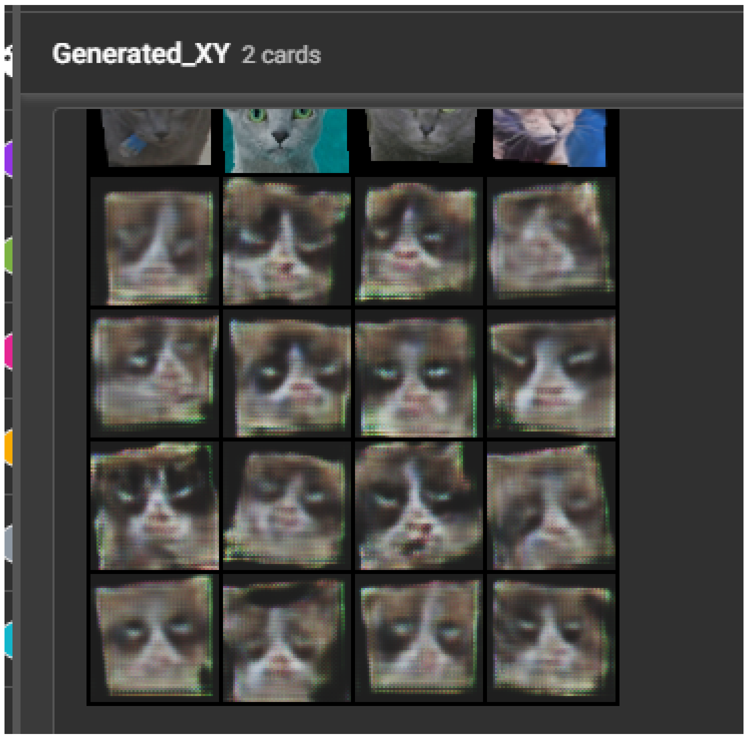
Cycle Consistency Loss

```
[ ]: !python cycle_gan.py --use_cycle_consistency_loss
```

Images 400 Iteraction



700 Iteraction

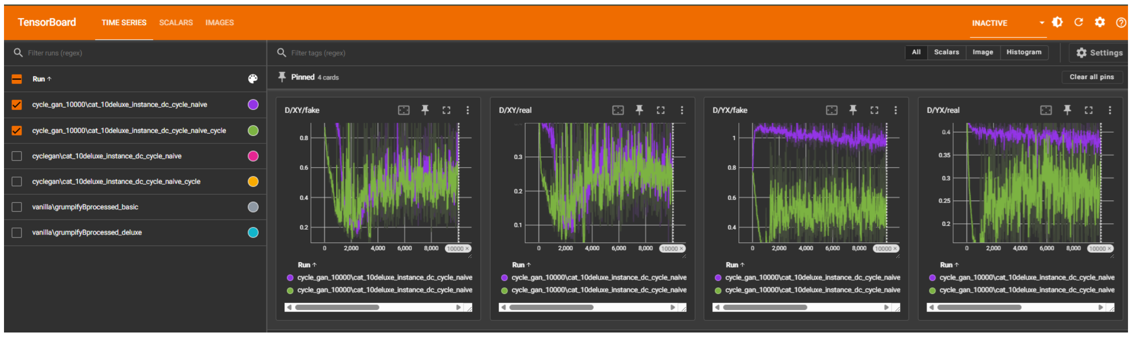


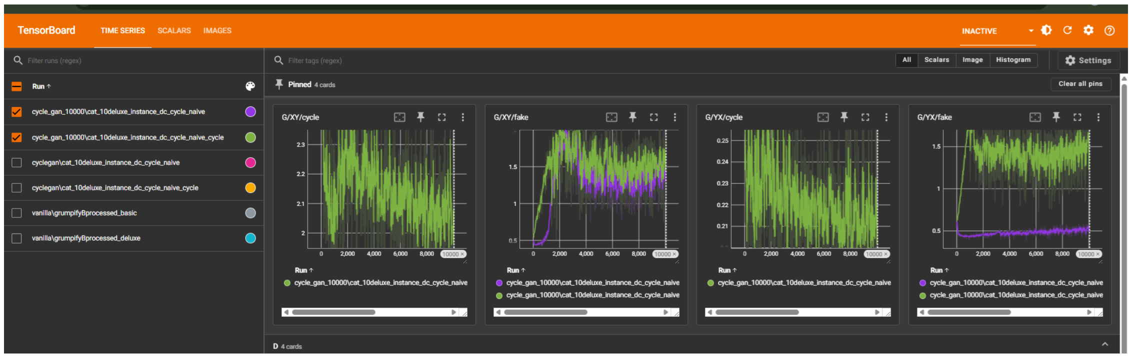
Looking at the images, we can see that the ones from step 400 may be somewhat similar to the ones from step 700, although they change and look somewhat better. This is probably because when the training has just started the generator has not learned very well how to change the first type of image to look like the second type. Now, in this case, both images may not be very well created and we could at a glance identify which are fake and which are real, since if we look at step 700, even though the network tries to get better at making the images look like the other type, it may not be enough to have better textures, match colors and look more real. So we may still see some problems or things that don't look quite right because the model is still learning and trying to improve in the next steps. Something that could be improved if we revisit images in later steps in the training.

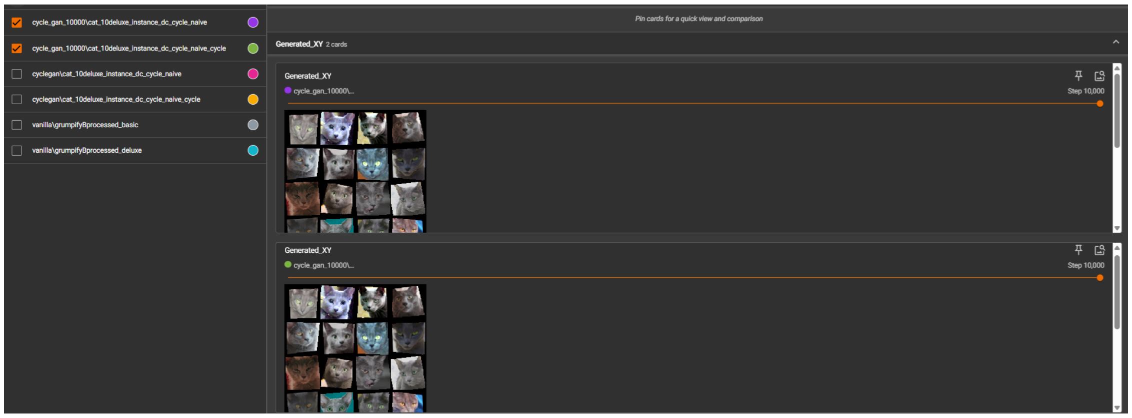
Execution 10000 Iteractions

```
[ ]: !python cycle_gan.py --train_iters=10000 --sample_dir=cycle_gan_10000
```

```
[ ]: !python cycle_gan.py --train_iters=10000 --sample_dir=cycle_gan_10000␣
      ↪--use_cycle_consistency_loss
```


TensorBoard
TIME SERIES
SCALARS
IMAGES
INACTIVE
Settings
Pinned 4 cards
Clear all pins
Run ↑
cycle_gan_10000\cat_10deluxe_instance_dc_cycle_naive
cycle_gan_10000\cat_10deluxe_instance_dc_cycle_naive_cycle
cyclegan\cat_10deluxe_instance_dc_cycle_naive
cyclegan\cat_10deluxe_instance_dc_cycle_naive_cycle
vanilla\grumpifyBprocessed_basic
vanilla\grumpifyBprocessed_deluxe
D/XY/fake
D/XY/real
D/YX/fake
D/YX/real





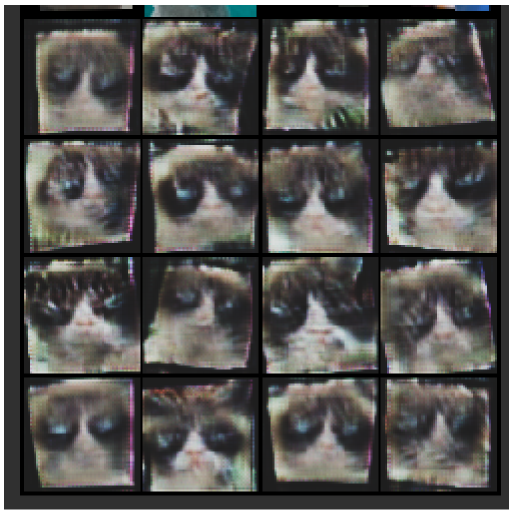
Results TensorBoard

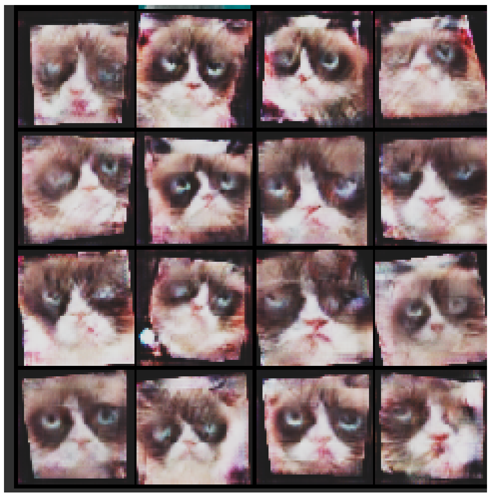
Can you account for these differences? Answer: We can see that when we use cycle consistency, the loss curves tend to be more stable, with gradual slopes and fewer extreme flactuations. This probably happens because the consistency loss acts as a guiding principle that ensures that the networks not only create fake images to "fool" the other network, but also thet they maintain coherence when an image is modified and then reverted to its original state. This guiding principle helps the training to be more stable and helps the networks to learn in a more organized way, instead of simply trying to "fool" each other all the time. The generator seems to learn better when following this principle.

Provide explanations as to why there might or might not be a noticeable difference between the two sets of results. Answer: Analyzing the images, we do not see a big difference in the final images. This could be because the two types of "grumpy cats" we are using are not different in style. If the transformation we want to make is not too significant, the network can probably still learn to perform it correctly, even without the cycle coherence rule. Also, the rule depends on its weight parameter (lambda), and if this parameter needs to be tuned.

In this case, we used L1 loss, as it was recommended in the original paper. Perhaps, if we trained the model longer or used larger networks, or if the difference between cat styles was more noticeable, we would see a significant improvement in the images produced when we use the cycle coherence rule.

Any differences appear to be difficult to detect and would probably require closer examination; at a glance the images generated are the same. This visual similarity supports the explanation given above about the possible reasons for a possible large difference in the final result for this particular data set and training configuration.





To observe a noticeable difference, we can compare the images generated at iteration 1900 with the one from the final iteration. We notice that the images generated in this iteration, show a significant improvenment in quality, which leads us to conclude that the Generator has learned to create better images. Finally, this comparison allows us to conclude that CycleGan performs much better that VanillaGan at generating high-quality images.