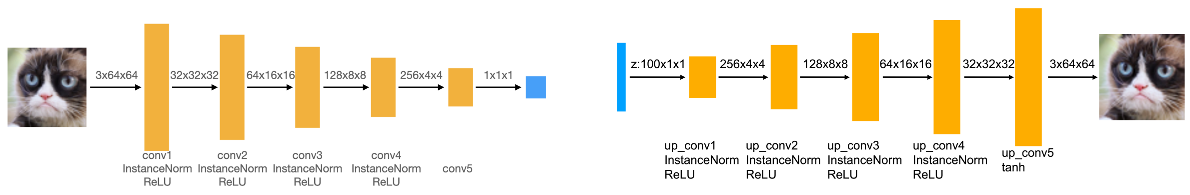
Homework 4: Generating Cats and Fruit Images with GANs

START HERE: Instructions

- **Collaboration policy:** All are encouraged to work together BUT you must do your own work (code and write up). If you work with someone, please include their name in your write-up and cite any code that has been discussed. If we find highly identical write-ups or code or lack of proper accreditation of collaborators, we will take action according to strict university policies. See the Academic Integrity Section detailed in the initial lecture for more information.
- **Late Submission Policy:** There are a **total of 5** late days across all homework submissions. Submissions more than 5 days after the deadline will receive a 0.
- **Submitting your work:**
  - We will be using Gradescope (`https://gradescope.com/`) to submit the Problem Sets. Please use the provided template only. Submissions must be written in LaTeX. All submissions not adhering to the template will not be graded and receive a zero.
  - **Deliverables:** Please submit all the `.py` files. Add all relevant plots and text answers in the boxes provided in this file. TO include plots you can simply modify the already provided latex code. Submit the compiled `.pdf` report as well.

NOTE: Partial points will be given for implementing parts of the homework even if you don't get the mentioned numbers as long as you include partial results in this pdf.



(a) Discriminator (b) Generator

Figure 1: DCGAN model architecture. (a) Architecture of the discriminator. (b) Architecture of the generator.

1 Introduction

In this assignment, you will explore hands-on-experience coding and training Generative Adversarial Networks (GAN) Goodfellow et al. [2014]. This homework consists of three parts: the first part includes the implementation of Deep Convolutional GAN (DCGAN) Radford et al. [2016] and training it to generate grumpy cats images; the second part explores different architectures and training objectives of DCGAN; the third part involves a more complex GAN model called CycleGAN Zhu et al. [2020] for image-to-image translation, where we will train it to convert between different types of two kinds of cats (Grumpy and Russian Blue). In both parts, you will gain experience implementing GANs by writing code for the generator, discriminator, and training loop, for each model. In all parts of this homework, you will gain experience implementing GANs, designing its architecture and objectives, and training loop of GANs. The starter code and data can be found here.

2 Part 1: Deep Convolutional GAN (30 points)

For the first part of this assignment, we will implement a slightly modified version of Deep Convolutional GAN (DCGAN). A DCGAN is simply a GAN that uses a convolutional neural network as the discriminator, and a network composed of transposed convolutions as the generator. In the assignment, instead of using transposed convolutions, we will be using a combination of a upsampling layer and a convolutional layer to replace transposed convolutions. To implement the DCGAN, we need to specify three things: 1) the generator, 2) the discriminator, and 3) the training procedure. We will develop each of these three components in the following subsections.

2.1 Implement the Discriminator (5 Points)

The (modified) discriminator of DCGAN consists of a series of convolutional layers, batch/instance normalization layers, and ReLU activation function, as shown in Fig. 1(a),

Implement this architecture by filling in the *init* function and *forward method* of the *DCDiscriminator* class in models.py, shown below. The *conv_dim* argument does not need to be changed unless you are using larger images, as it should specify the initial image size.

2.2 Implement the Generator (5 Points)

Now, we will implement the generator of the DCGAN, which consists of a sequence of upsample+convolutional layers that progressively upsample the input noise sample to generate a fake image. The generator in this DCGAN has the architecture as shown in Fig. 1(b).

Implement this architecture by filling in the *init* and *forward method* of the DCGenerator class in models.py. Note: Use the *up_conv* function (analogous to the conv function used for the discriminator above) in your generator implementation. We find that for the first layer (*up_conv1*) it is better to directly apply convolution layer without any upsampling to get 4x4 output. To do so, you'll need to think about what you kernel and padding size should be in this case. Feel free to use *up_conv* for the rest of the layers.

2.3 DCGAN Training Loop (20 Points)

Next, you will implement the training loop for the DCGAN. A DCGAN is simply a GAN with a specific type of generator and discriminator; thus, we train it in exactly the same way as a standard GAN. The pseudo-code for the training procedure is shown in Algorithm 1. The actual implementation is simpler than it may seem from the pseudo-code: this will give you practice in translating math to code.

Algorithm 1 DCGAN algorithm

```
1: Draw $m$ training samples $\{x_0, x_1, ..., x_m\}$ from data distribution $p_{data}$.
2: Draw $m$ noise samples $\{z_0, z_1, ..., z_m\}$ from noise distribution $p_z$.
3: Generate fake images from the noise: $G(z_i)$ for $i \in \{1, ..., m\}$.
4: Compute the discriminator loss.
5: Update the parameters of the discriminator.
6: Draw $m$ new noise samples $\{z_0, z_1, ..., z_m\}$ from noise distribution $p_z$.
7: Generate fake images from the noise: $G(z_i)$ for $i \in \{1, ..., m\}$.
8: Compute the generator loss.
9: Update the parameters of the generator.
```

2.3.1 Implementation (5 Point)

Fill the *training_loop* part in vanilla_ganṗy. There are 4 numbered bullets in the code to fill in for the discriminator and 3 bullets for the generator. Each of these can be done in a single line of code, although you will not lose marks for using multiple lines.

DCGAN will perform poorly without data augmentation on a small-sized dataset because the discriminator can easily overfit to a real dataset. To rescue, we need to add some data augmentation such as random crop and random horizontal flip. You need to fill in advanced version of data augmentation in data_loaderṗy. We provide some script for you to begin with. You need to compose them into a transform object which is passed to CustomDataset.

2.3.2 Training and Visualization (15 Point)

Train DCGAN by running the completed *vanilla_ganṗy*. Provide your training hyper-parameters. Show a grid of generated cats images using basic augmentation and advanced augmentation for the beginning of training (e.g. 200 or 400 iterations) and the end of training (e.g. 3000 or 6000 iterations) here respectively. Also plot the training loss correspondingly. (In total 6 images are expected to shown here.)

training hyper-parameters:
image_size =64
conv_dim=32
noise_size=100
num_epochs=500
batch_size=16
num_workers=2
lr=0.0002
beta1=0.5
beta2=0.999
The result images:
basic augmentation:(the image is in next or last pages):
loss: 2
beginning: 3
end: 4
advanced augmentation:(the image is in last pages):
loss: 5
beginning: 6
end: 7

3 Part 2: Architectures and Objective Functions (20 points)

In this section, you are encouraged to try different architecture of the generator and discriminator in DCGAN and different loss functions to train model for generating cats images with better visual quality. You can choose to change either architecture or the objective function, or both.

3.1 (Option 1) Architecture

You can try residual connections, different number of layers, different normalization layers, and regularization on weights here. Describe your architecture here, and modify the model code in model_variant.py accordingly.

I have tried two architectures:

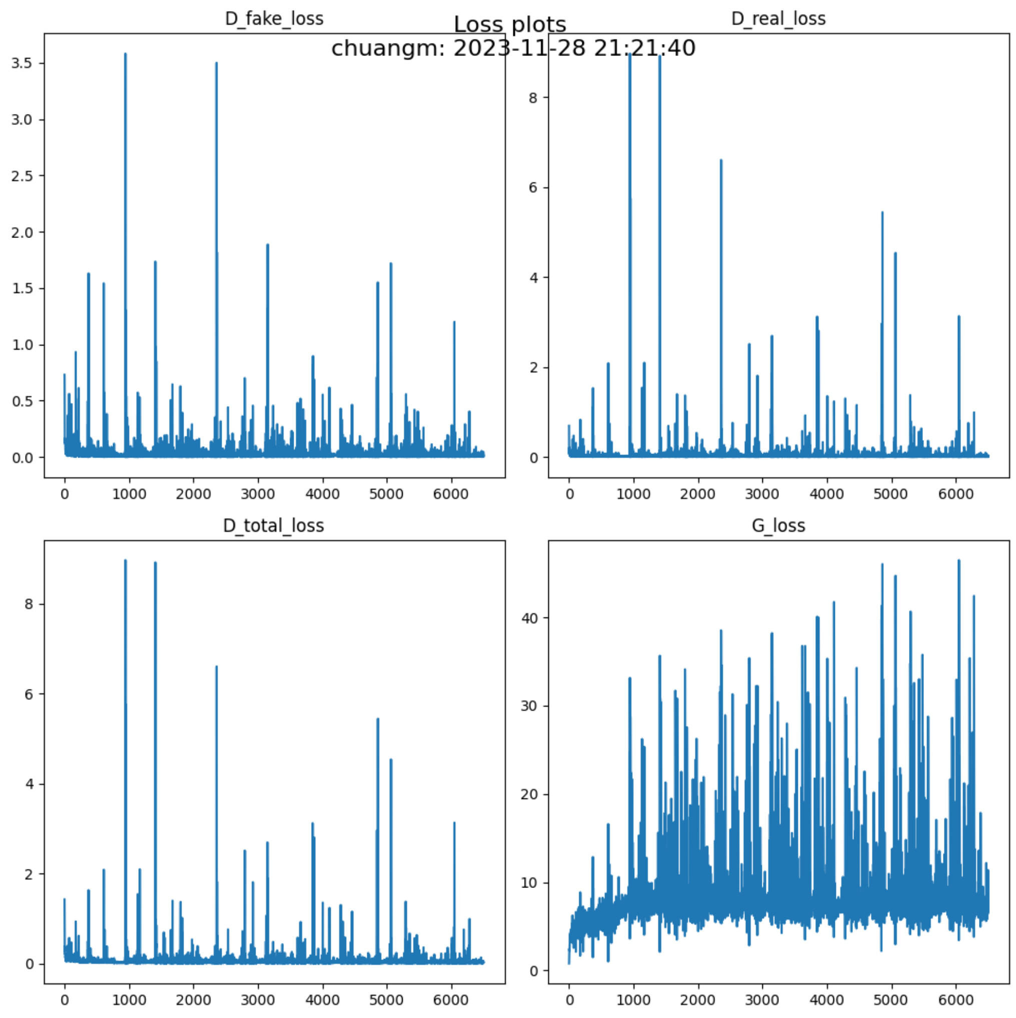


Figure 2: Loss for basic augmentation

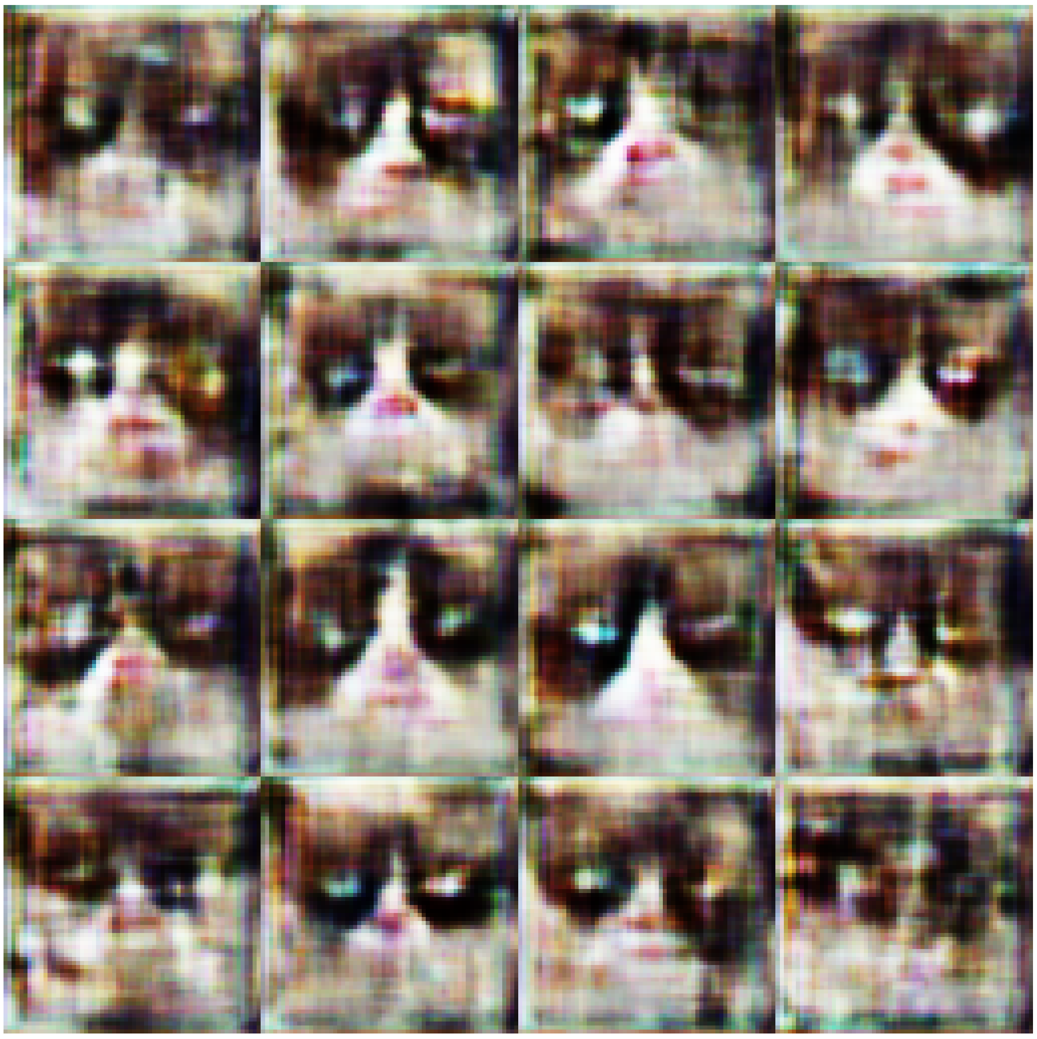


Figure 3: beginning of training for basic augmentation

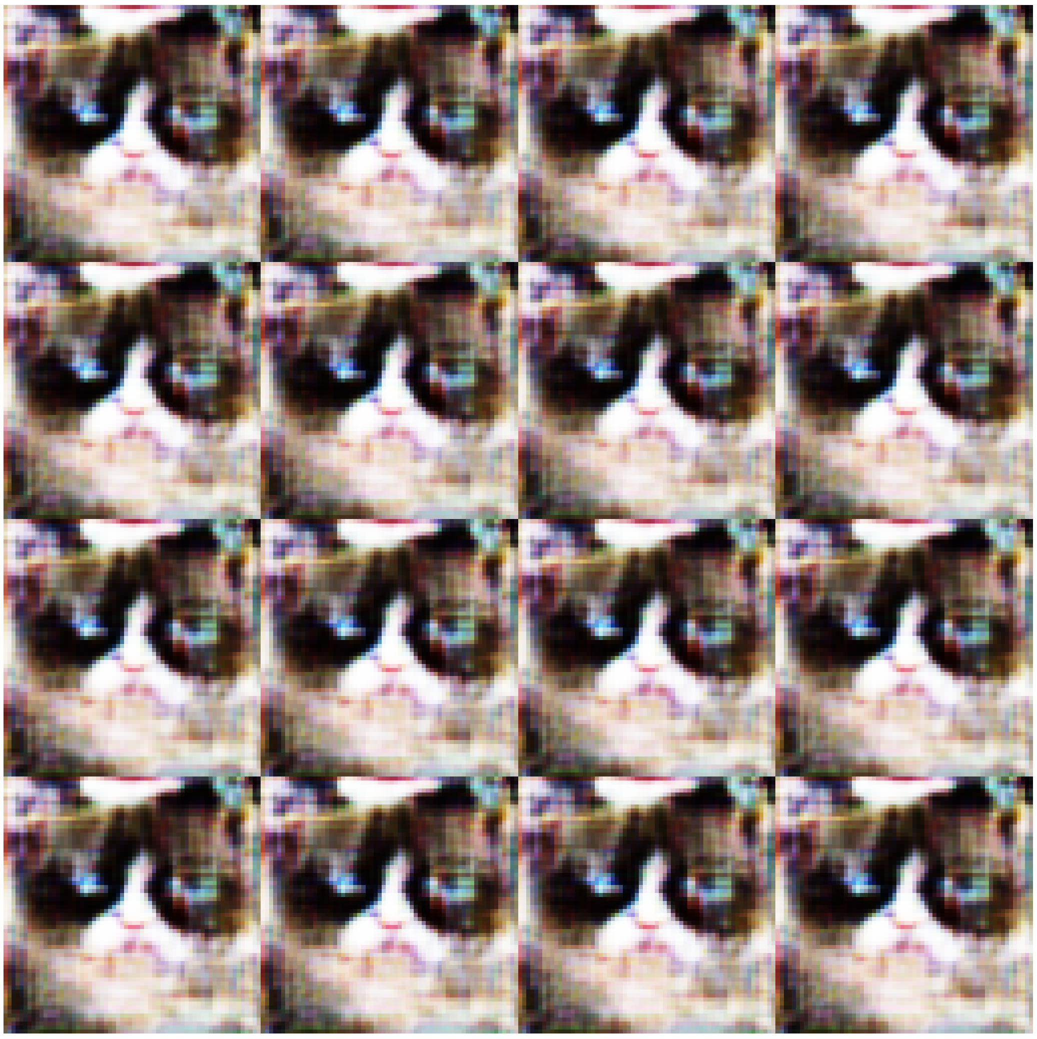


Figure 4: end of training for basic augmentation

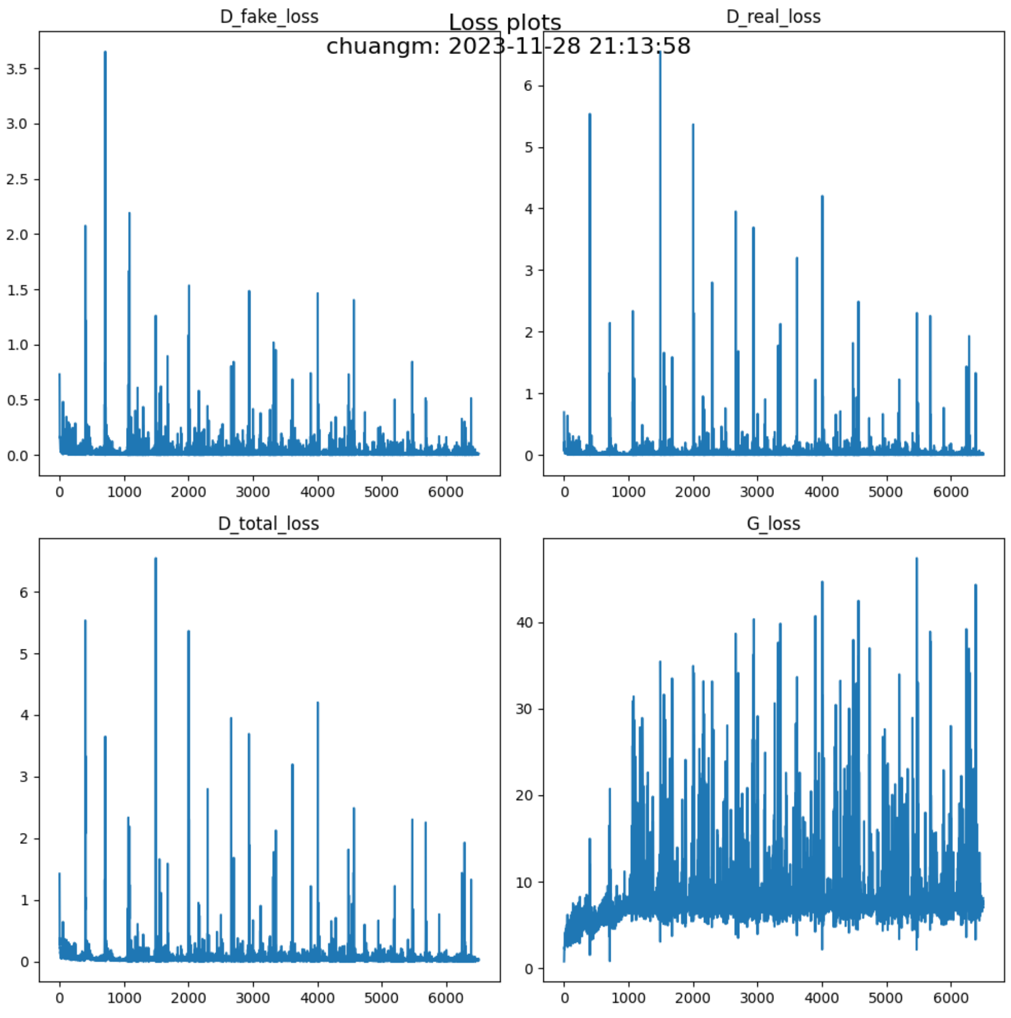


Figure 5: Loss for advanced augmentation

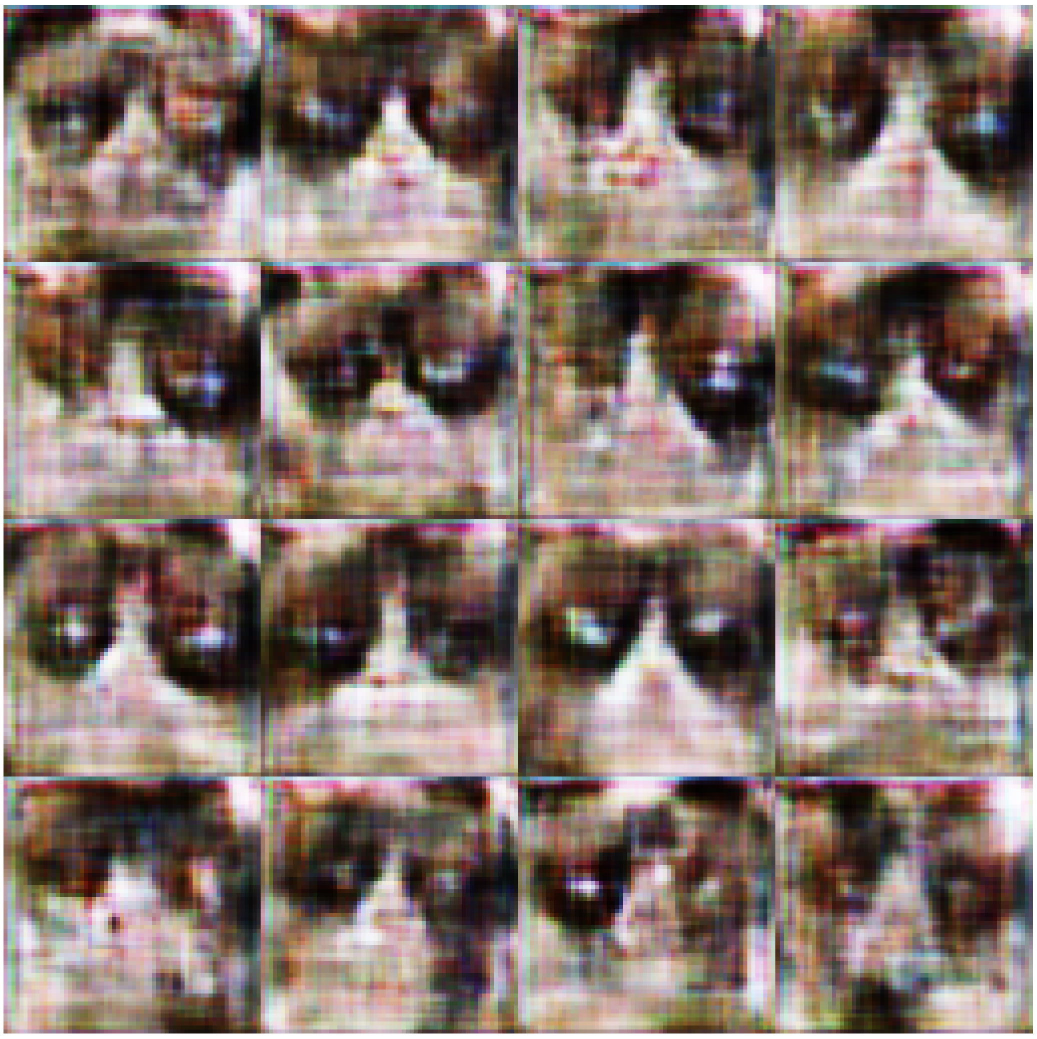


Figure 6: beginning of training for advanced augmentation

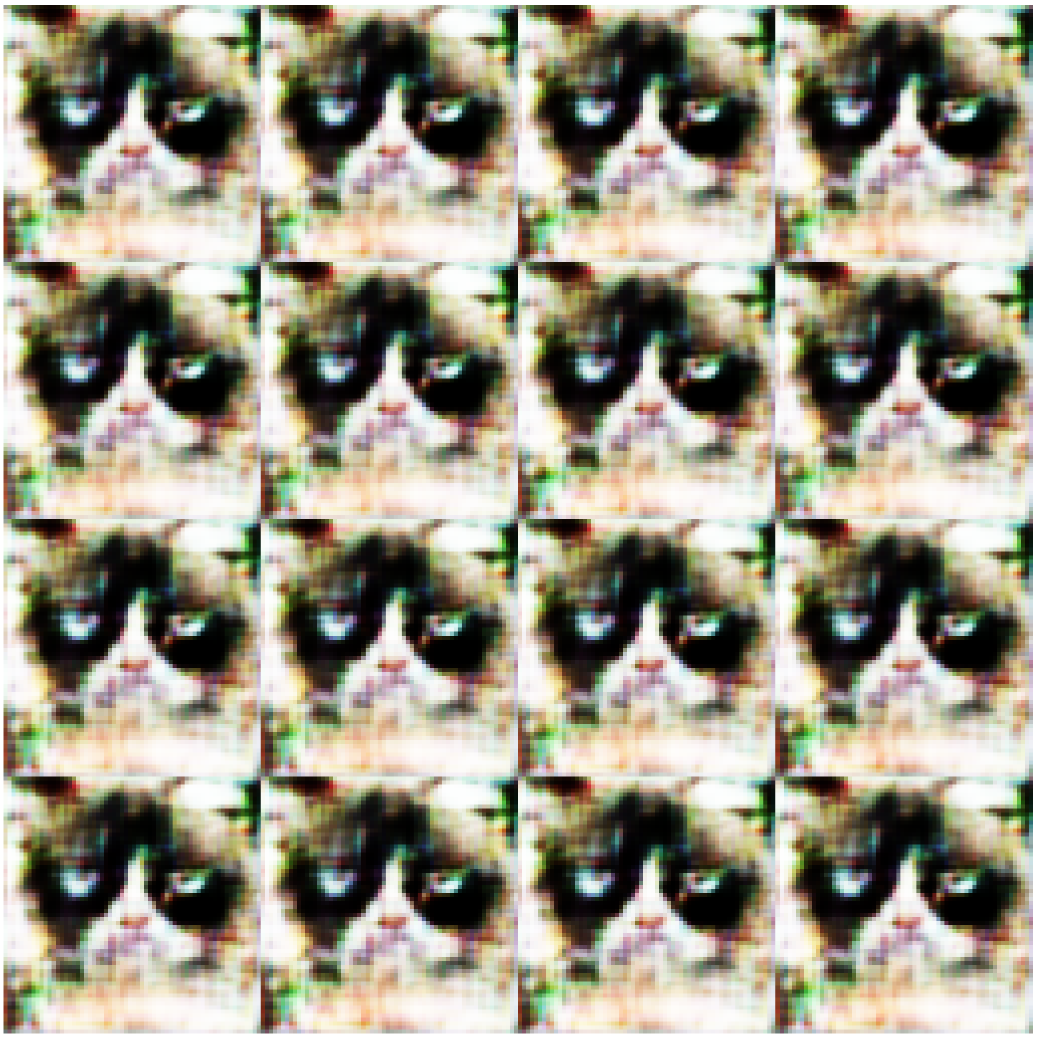


Figure 7: end of training for advanced augmentation

1.adding residual connection in both generator and discriminator
2. adding residual connection in only generator.
The other parts are same.
This is the first one:

```
                    G
---------------------------------------
 DCGenerator(
 (up_conv1): Sequential(
 (0): Conv2d(100, 256, kernel_size=(4, 4), stride=(1, 1), padding=(3, 3))
 (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
 (2): ReLU()
 )
 (resnet2): ResnetBlock(
 (conv_layer): Sequential(
 (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
 (2): ReLU()
 )
 )
 (up_conv2): Sequential(
 (0): Upsample(scale_factor=2.0, mode='nearest')
 (1): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=same, bias=False)
 (2): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
 (3): ReLU()
 )
 (resnet3): ResnetBlock(
 (conv_layer): Sequential(
 (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (1): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
 (2): ReLU()
 )
 )
 (up_conv3): Sequential(
 (0): Upsample(scale_factor=2.0, mode='nearest')
 (1): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=same, bias=False)
 (2): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
 (3): ReLU()
 )
 (resnet4): ResnetBlock(
 (conv_layer): Sequential(
 (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (1): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
 (2): ReLU()
 )
 )
 (up_conv4): Sequential(
 (0): Upsample(scale_factor=2.0, mode='nearest')
 (1): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=same, bias=False)
 (2): InstanceNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
 (3): ReLU()
 )
 (resnet5): ResnetBlock(
 (conv_layer): Sequential(
 (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
 (1): InstanceNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
 (2): ReLU()
 )
 )
 (up_conv5): Sequential(
 (0): Upsample(scale_factor=2.0, mode='nearest')
```

```
(1): Conv2d(32, 3, kernel_size=(3, 3), stride=(1, 1), padding=same, bias=False)
(2): Tanh()
)
)
----------------------------------------
D
----------------------------------------
DCDiscriminator(
(conv1): Sequential(
(0): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(1): InstanceNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
(2): ReLU()
)
(resnet2): ResnetBlock(
(conv_layer): Sequential(
(0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(1): InstanceNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
(2): ReLU()
)
)
(conv2): Sequential(
(0): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(1): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
(2): ReLU()
)
(resnet3): ResnetBlock(
(conv_layer): Sequential(
(0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(1): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
(2): ReLU()
)
)
(conv3): Sequential(
(0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(1): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False
(2): ReLU()
)
(resnet4): ResnetBlock(
(conv_layer): Sequential(
(0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(1): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False
(2): ReLU()
)
)
(conv4): Sequential(
(0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(1): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False
(2): ReLU()
)
(conv5): Sequential(
(0): Conv2d(256, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
)
)
```

3.2 (Option 2) Objective Functions

You can try Wasserstein GAN Arjovsky et al. [2017] loss function or least square GAN Mao et al. [2017] loss function. Other loss functions could also be explored. Implement your loss in the training loop. Provide a description and equations for the loss function used.

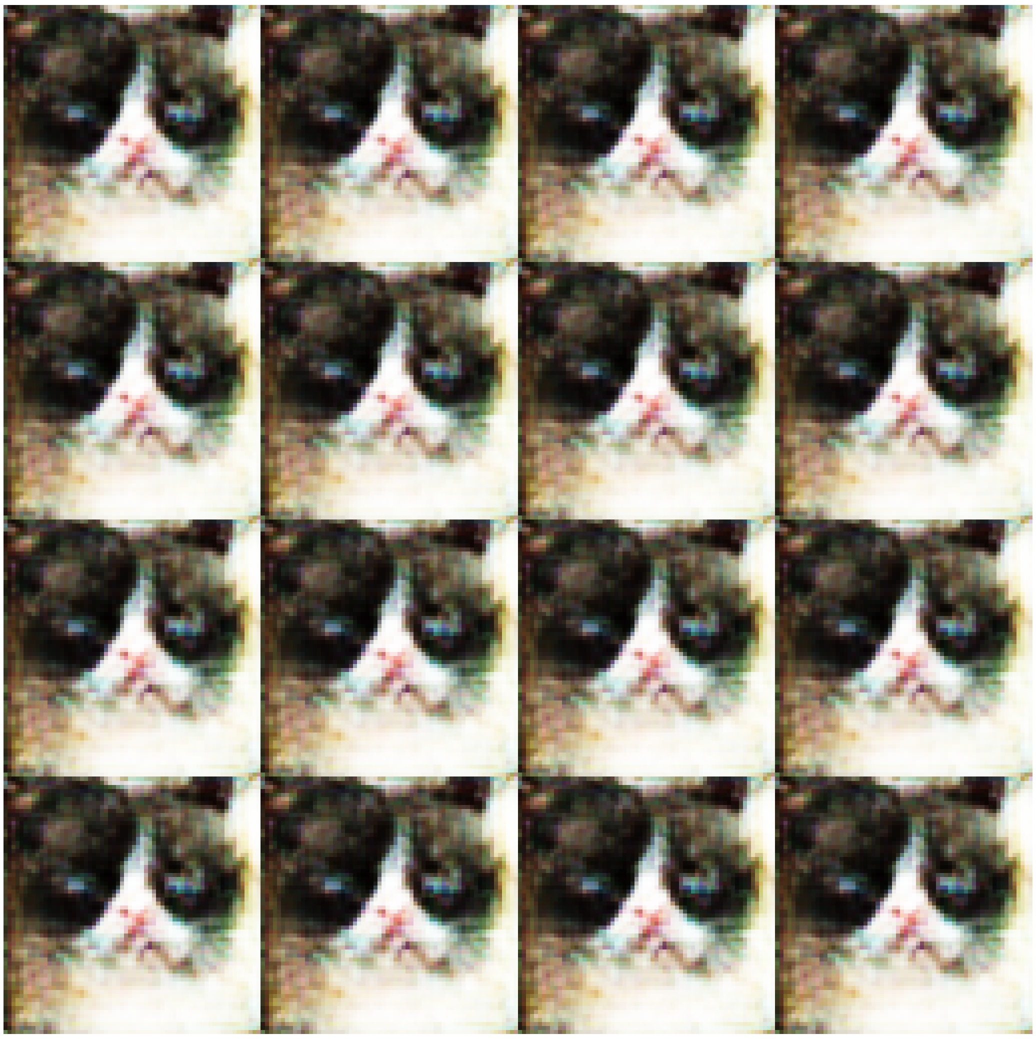


Figure 8: residual connection in generator1

3.3 Training and Visualization (5 Points)

Train DCGAN with modified architecture and loss function again. Give explanation of your modification. Show a grid of generated cats images with better quality here.

I have tried two architectures:

1.adding residual connection in both generator and discriminator

2. adding residual connection in only generator.

Explain:

In both the generator and discriminator, there may be the problem of Vanishing Gradient. Add residual blocks can solve this. Besides, it can act as shortcuts for the flow of gradients, making it easier for the generator to learn to generate good images. This can lead to faster convergence. So I tried to add residual blocks in between each conv and up_conv to get a more stable training of GANs.

Here are two test results I have done:

The result of only adding residual connection in generator:

We can see the image do get improved.

8

9

The result of adding residual connection in both generator and discriminator:

We can see the image do get improved.

10

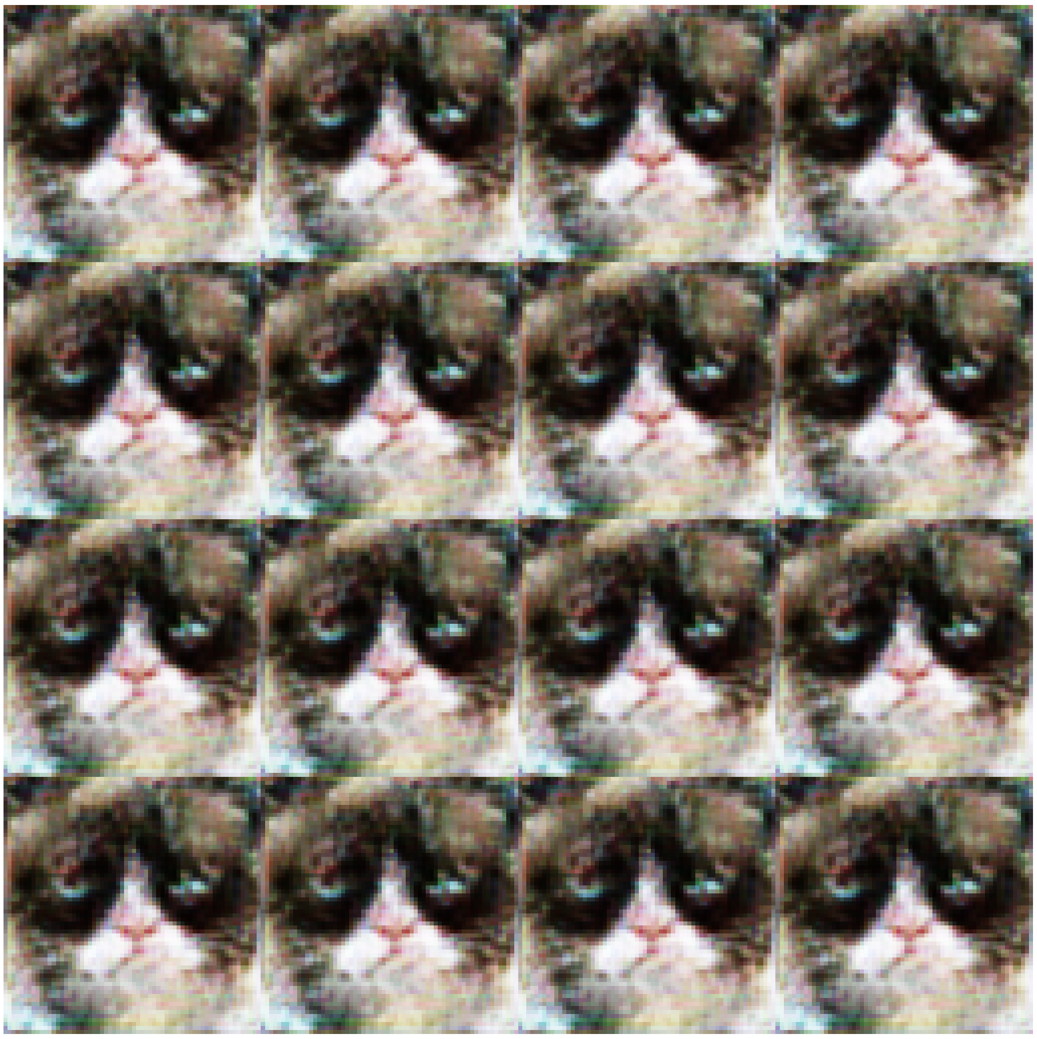


Figure 9: residual connection in generator2

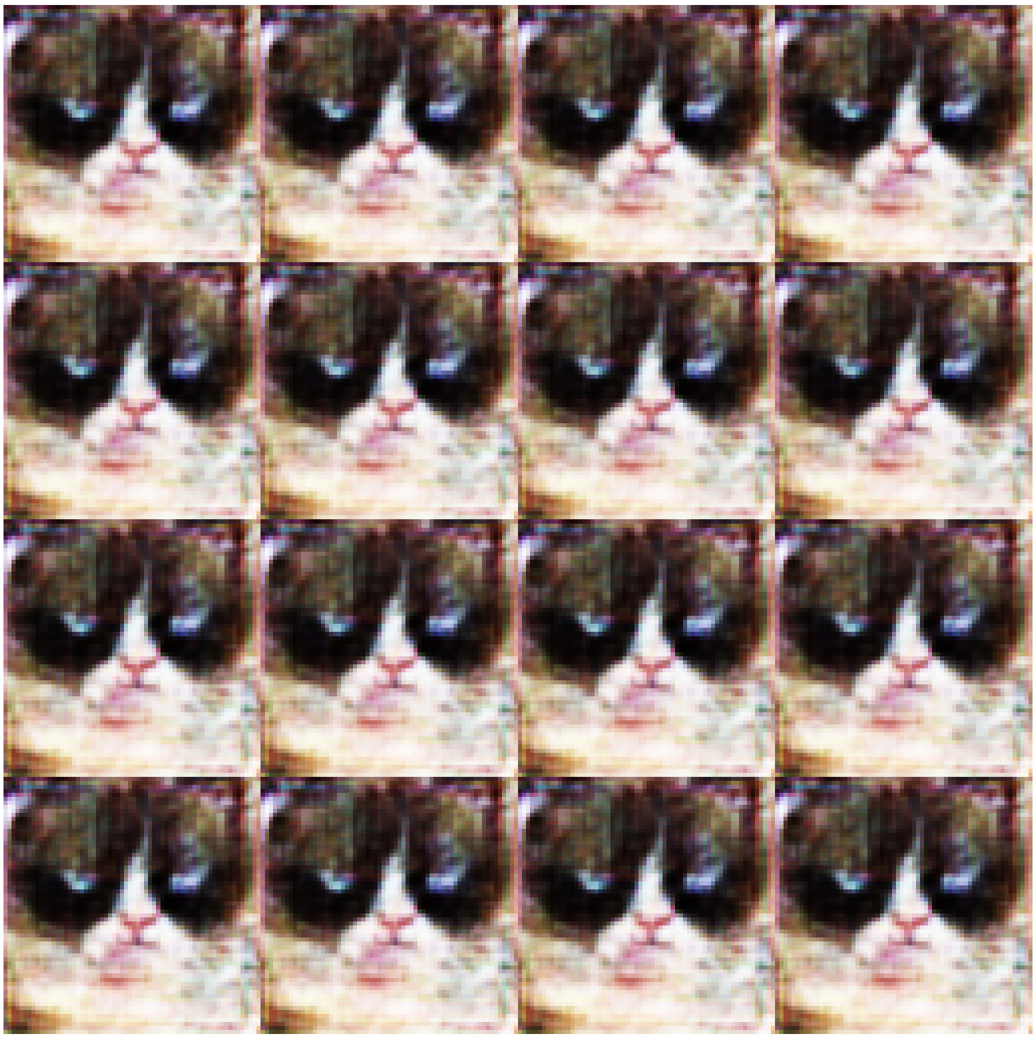


Figure 10: residual connection in both generator and discriminator

References

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks, 2020.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017.

Xudong Mao, Qing Li, Haoran Xie, Raymond Y. K. Lau, Zhen Wang, and Stephen Paul Smolley. Least squares generative adversarial networks, 2017.