Problem 2 - Generative Adversarial Networks (GAN)

Open in Colab

- **Learning Objective**: In this problem, you will implement a Generative Adversarial Network with the network structure proposed in Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. You will also learn a visualization technique: activation maximization.
- **Provided code**: The code for constructing the two parts of the GAN, the discriminator and the generator, is done for you, along with the skeleton code for the training.
- **TODOs**: You will need to figure out how to define the training loop, compute the loss, and update the parameters to complete the training and visualization. In addition, to test your understanding, you will answer some non-coding written questions. Please see details below.

Note:

- If you use the Colab, for faster training of the models in this assignment, you can enable GPU support in the Colab. Navigate to "Runtime" --> "Change Runtime Type" and set the "Hardware Accelerator" to "GPU". **However, Colab has the GPU limit, so be discretional with your GPU usage.**
- If you run into CUDA errors in the Colab, check your code carefully. After fixing your code, if the CUDA error shows up at a previously correct line, restart the Colab. However, this is not a fix to all your CUDA issues. Please check your implementation carefully.

In [1]:
```
# Import required libraries
import torch.nn as nn
import torch
import numpy as np
import matplotlib.pyplot as plt
import math
from torchvision.utils import make_grid
%matplotlib inline

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

Introduction: The forger versus the police

Please read the information below even if you are familiar with GANs. There are some terms below that will be used in the coding part.

Generative models try to model the distribution of the data in an explicit way, in the sense that we can easily sample new data points from this model. This is in contrast to discriminative models that try to infer the output from the input. In class and in the previous problem, we have seen one classic deep generative model, the Variational Autoencoder (VAE). Here, we will learn another generative model that has risen to prominence in recent years, the Generative Adversarial Network (GAN).

As the math of Generative Adversarial Networks are somewhat tedious, a story is often told of a forger and a police officer to illustrate the idea.

> Imagine a forger that makes fake bills, and a police officer that tries to find these forgeries. If the forger were a VAE, his goal would be to take some real bills, and try to replicate the real bills as precisely as possible. With GANs, the forger has a different idea: rather than trying to replicate the real bills, it suffices to make fake bills such that people think they are real.
>
> Now let's start. In the beginning, the police knows nothing about how to distinguish between real and fake bills. The forger knows nothing either and only produces white paper.
>
> In the first round, the police gets the fake bill and learns that the forgeries are white while the real bills are green. The forger then finds out that white papers can no longer fool the police and starts to produce green papers.
>
> In the second round, the police learns that real bills have denominations printed on them while the forgeries do not. The forger then finds out that plain papers can no longer fool the police and starts to print numbers on them.
>
> In the third round, the police learns that real bills have watermarks on them while the forgeries do not. The forger then has to reproduce the watermarks on his fake bills.
>
> ...
>
> Finally, the police is able to spot the tiniest difference between real and fake bills and the forger has to make perfect replicas of real bills to fool the police.

Now in a GAN, the forger becomes the generator and the police becomes the discriminator. The discriminator is a binary classifier with the two classes being "taken from the real data" ("real") and "generated by the generator" ("fake"). Its objective is to minimize the classification loss. The generator's objective is to generate samples so that the discriminator misclassifies them as real.

Here we have some complications: the goal is not to find one perfect fake sample. Such a sample will not actually fool the discriminator: if the forger makes hundreds of the exact same fake bill, they will all have the same serial number and the police will soon find out that they are fake. Instead, we want the generator to be able to generate a variety of fake samples such that

when presented as a distribution alongside the distribution of real samples, these two are indistinguishable by the discriminator.

So how do we generate different samples with a deterministic generator? We provide it with random numbers as input.

Typically, for the discriminator we use *binary cross entropy loss* with label 1 being real and 0 being fake. For the generator, the input is a random vector drawn from a standard normal distribution. Denote the generator by $G_\phi(z)$, discriminator by $D_\theta(x)$, the distribution of the real samples by $p(x)$, and the input distribution to the generator by $q(z)$. Recall that the binary cross entropy loss with classifier output y and label $\hat{y}$ is

$$L(y, \hat{y}) = -\hat{y} \log y - (1 - \hat{y}) \log(1 - y)$$

For the discriminator, the objective is

$$\min_\theta \mathrm{E}_{x \sim p(x)}[L(D_\theta(x), 1)] + \mathrm{E}_{z \sim q(z)}[L(D_\theta(G_\phi(z)), 0)]$$

For the generator, the objective is

$$\max_\phi \mathrm{E}_{z \sim q(z)}[L(D_\theta(G_\phi(z)), 0)]$$

The generator's objective corresponds to maximizing the classification loss of the discriminator on the generated samples. Alternatively, we can **minimize** the *classification loss* of the discriminator on the generated samples **when labelled as real:**

$$\min_\phi \mathrm{E}_{z \sim q(z)}[L(D_\theta(G_\phi(z)), 1)]$$

And this is what we will use in our implementation. The strength of the two networks should be balanced, so we train the two networks alternatingly, updating the parameters in both networks once in each iteration.

Problem 2-1: Implementing the GAN (20 pts)

Correctly filling out `__init__`: 7 pts

Correctly filling out training loop: 13 pts

We first load the data (CIFAR-10) and define some convenient functions. You can run the cell below to download the dataset to `./data`.

In [2]:
```
!wget http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz -P data
!tar -xzvf data/cifar-10-python.tar.gz --directory data
!rm data/cifar-10-python.tar.gz
```

```
--2023-04-04 03:53:21--  http://www.cs.toronto.edu/~kriz/cifar-10-python.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: 170498071 (163M) [application/x-gzip]
Saving to: ‘data/cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  29.6MB/s    in 6.1s

2023-04-04 03:53:27 (26.6 MB/s) - ‘data/cifar-10-python.tar.gz’ saved [170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
```

In [3]:

```python
def unpickle(file):
    import sys
    if sys.version_info.major == 2:
        import cPickle
        with open(file, 'rb') as fo:
            dict = cPickle.load(fo)
        return dict['data'], dict['labels']
    else:
        import pickle
        with open(file, 'rb') as fo:
            dict = pickle.load(fo, encoding='bytes')
        return dict[b'data'], dict[b'labels']
def load_train_data():
    X = []
    for i in range(5):
        X_, _ = unpickle('data/cifar-10-batches-py/data_batch_%d' % (i + 1))
        X.append(X_)
    X = np.concatenate(X)
    X = X.reshape((X.shape[0], 3, 32, 32))
    return X

def load_test_data():
    X_, _ = unpickle('data/cifar-10-batches-py/test_batch')
    X = X_.reshape((X_.shape[0], 3, 32, 32))
    return X

def set_seed(seed):
    np.random.seed(seed)
    torch.manual_seed(seed)

# Load cifar-10 data
train_samples = load_train_data() / 255.0
test_samples= load_test_data() / 255.0
```

To save you some mundane work, we have defined a discriminator and a generator for you. Look at the code to see what layers are there.

For this part, you need to complete code blocks marked with "Prob 2-1":

- **Build the Discriminator and Generator, define the loss objectives**
- **Define the optimizers**
- **Build the training loop and compute the losses**: As per How to Train a GAN? Tips and tricks to make GANs work, we put real samples and fake samples in different batches when training the discriminator.

Note: use the advice on that page with caution if you are using GANs for your team project. It is already 4 years old, which is a really long time in deep learning research. It does not reflect the latest results.

In [4]:
```
class Generator(nn.Module):
    def __init__(self, starting_shape):
        super(Generator, self).__init__()
        self.fc = nn.Linear(starting_shape, 4 * 4 * 128)
        self.upsample_and_generate = nn.Sequential(
            nn.BatchNorm2d(128),
            nn.LeakyReLU(),
            nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=4, stride
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.ConvTranspose2d(in_channels=64, out_channels=32, kernel_size=4, stride=
            nn.BatchNorm2d(32),
            nn.LeakyReLU(),
            nn.ConvTranspose2d(in_channels=32, out_channels=3, kernel_size=4, stride=2
            nn.Sigmoid()
        )
    def forward(self, input):
        transformed_random_noise = self.fc(input)
        reshaped_to_image = transformed_random_noise.reshape((-1, 128, 4, 4))
        generated_image = self.upsample_and_generate(reshaped_to_image)
        return generated_image
```

In [5]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.downsample = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=4, stride=2, padding
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2, paddi
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2, padd
            nn.BatchNorm2d(128),
            nn.LeakyReLU(),
        )
        self.fc = nn.Linear(4 * 4 * 128, 1)
    def forward(self, input):
        downsampled_image = self.downsample(input)
        reshaped_for_fc = downsampled_image.reshape((-1, 4 * 4 * 128))
```

```
        classification_probs = self.fc(reshaped_for_fc)
        return classification_probs
```

In [6]:

```
# Use this to put tensors on GPU/CPU automatically when defining tensors
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

class DCGAN(nn.Module):
    def __init__(self):
        super(DCGAN, self).__init__()
        self.num_epoch = 25
        self.batch_size = 128
        self.log_step = 100
        self.visualize_step = 2
        self.code_size = 64  # size of latent vector (size of generator input)
        self.learning_rate = 2e-4
        self.vis_learning_rate = 1e-2

        # IID N(0, 1) Sample
        self.tracked_noise = torch.randn([64, self.code_size], device=device)
        self._actmax_label = torch.ones([64, 1], device=device)

        ########################################################################
        # Prob 2-1: Define the generator and discriminator, and loss functions  #
        # Also, apply the custom weight initialization (see link:
        # https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html)
        ########################################################################
        # To-Do: Initialize generator and discriminator
        # use variable name "self._generator" and "self._discriminator", respectively
        # (also move them to torch device for accelerating the training later)
        self._generator = Generator(self.code_size).to(device = device)
        self._discriminator = Discriminator().to(device = device)

        # To-Do: Apply weight initialization (first implement the weight initializatio
        # function below by following the given link)
        self._weight_initialization()

        ########################################################################
        # Prob 2-1: Define the generator and discriminators' optimizers
        # HINT: Use Adam, and the provided momentum values (betas)
        ########################################################################
        betas = (0.5, 0.999)
        # To-Do: Initialize the generator's and discriminator's optimizers
        self.optimizerD = torch.optim.Adam(self._discriminator.parameters(), betas= be
        self.optimizerG = torch.optim.Adam(self._generator.parameters(), betas= betas,

    # To-Do: Define weight initialization function
    # see link: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
    def _weight_initialization(self):
        pass

        def weights_init(m):
            """ Normal weight initialization as suggested for DCGANs """
            classname = m.__class__.__name__
            if classname.find('Conv') != -1:
                nn.init.normal_(m.weight.data, 0.0, 0.02)
            elif classname.find('BatchNorm') != -1:
                nn.init.normal_(m.weight.data, 1.0, 0.02)
                nn.init.constant_(m.bias.data, 0)
```

```
        self._generator.apply(weights_init)
        self._discriminator.apply(weights_init)
        # for layer in self._discriminator:
        #     nn.init.normal_(layer.weight.data, 0.0, 0.02)
        #     nn.init.constant_(layer.bias.data, 0)
        # for layer in self._generator.layers:
        #     nn.init.normal_(layer.weight.data, 0.0, 0.02)
        #     nn.init.constant_(layer.bias.data, 0)

    # To-Do: Define classification loss function (binary cross entropy loss)
    def _classification_loss(self,logits, labels):
        pass
        # real_labels : For the discriminator (D), the true target (y = 1) corresponds
        # Thus, for the scores of real images, the target is always 1 (a vector).
        # fake_labels : For D, the false target (y = 0) corresponds to "fake" images.
        # Thus, for the scores of fake images, the target is always 0 (a vector).
        # Compute the BCE logits : real/fake images, labels : real/fake labels.

        loss = nn.functional.binary_cross_entropy_with_logits(logits, labels)
        return loss
        ##########################################################################
        #                                END OF YOUR CODE
        ##########################################################################

    # Training function
    def train(self, train_samples):
        num_train = train_samples.shape[0]
        step = 0

        # smooth the loss curve so that it does not fluctuate too much
        smooth_factor = 0.95
        plot_dis_s = 0
        plot_gen_s = 0
        plot_ws = 0

        dis_losses = []
        gen_losses = []
        max_steps = int(self.num_epoch * (num_train // self.batch_size))
        fake_label = torch.zeros([self.batch_size, 1], device=device)
        real_label = torch.ones([self.batch_size, 1], device=device)
        self._generator.train()
        self._discriminator.train()
        print('Start training ...')
        for epoch in range(self.num_epoch):
            np.random.shuffle(train_samples)
            for i in range(num_train // self.batch_size):
                step += 1

                batch_samples = train_samples[i * self.batch_size : (i + 1) * self.bat
                batch_samples = torch.Tensor(batch_samples).to(device)

                ##########################################################################
                # Prob 2-1: Train the discriminator on all real images first
                ##########################################################################
                # To-Do: HINT: Remember to eliminate all discriminator gradients first
                self.optimizerD.zero_grad()

                # To-Do: feed real samples to the discriminator
                real_logits = self._discriminator(batch_samples)
```

```
                # To-Do: calculate the discriminator loss for real samples
                # use the variable name "real_dis_loss"
                real_dis_loss = self._classification_loss(real_logits, real_label)

                ##########################################################################
                # Prob 2-1: Train the discriminator with an all fake batch
                ##########################################################################
                # To-Do: sample noises from IID Normal(0, 1)^d on the torch device
                fake_data = torch.randn((self.batch_size, self.code_size), device=devi

                # To-Do: generate fake samples from the noise using the generator
                fake_samples = self._generator(fake_data)

                # To-Do: feed fake samples to discriminator
                # Make sure to detach the fake samples from the gradient calculation
                # when feeding to the discriminator, we don't want the discriminator t
                # receive gradient info from the Generator
                fake_logits = self._discriminator(fake_samples.detach())

                # To-Do: calculate the discriminator loss for fake samples
                # use the variable name "fake_dis_loss"
                fake_dis_loss = self._classification_loss(fake_logits, fake_label)

                # To-Do: calculate the total discriminator loss (real loss + fake loss
                dis_loss = real_dis_loss + fake_dis_loss

                # To-Do: calculate the gradients for the total discriminator loss
                dis_loss.backward()

                # To-Do: update the discriminator weights
                self.optimizerD.step()

                ##########################################################################
                # Prob 2-1: Train the generator
                ##########################################################################
                # To-Do: Remember to eliminate all generator gradients first! (.zero_g
                self.optimizerG.zero_grad()

                # To-Do: sample noises from IID Normal(0, 1)^d on the torch device
                fake_data_g = torch.randn((self.batch_size, self.code_size), device=de

                # To-Do: generate fake samples from the noise using the generator
                fake_samples_g = self._generator(fake_data_g)

                # To-Do: feed fake samples to the discriminator
                # No need to detach from gradient calculation here, we want the
                # generator to receive gradient info from the discriminator
                # so it can learn better.
                logits_fake_g = self._discriminator(fake_samples_g)

                # To-Do: calculate the generator loss
                # hint: the goal of the generator is to make the discriminator
                # consider the fake samples as real
                gen_loss = self._classification_loss(logits_fake_g, real_label)

                # To-Do: Calculate the generator loss gradients
                gen_loss.backward()

                # To-Do: Update the generator weights
                self.optimizerG.step()
```

```
            ##########################################################################
            #                                 END OF YOUR CODE
            ##########################################################################

            dis_loss = real_dis_loss + fake_dis_loss

            plot_dis_s = plot_dis_s * smooth_factor + dis_loss * (1 - smooth_facto
            plot_gen_s = plot_gen_s * smooth_factor + gen_loss * (1 - smooth_facto
            plot_ws = plot_ws * smooth_factor + (1 - smooth_factor)
            dis_losses.append(plot_dis_s / plot_ws)
            gen_losses.append(plot_gen_s / plot_ws)

            if step % self.log_step == 0:
                print('Iteration {0}/{1}: dis loss = {2:.4f}, gen loss = {3:.4f}'.

        if epoch % self.visualize_step == 0:
            fig = plt.figure(figsize = (8, 8))
            ax1 = plt.subplot(111)
            ax1.imshow(make_grid(self._generator(self.tracked_noise.detach()).cpu(
            plt.show()

            dis_losses_cpu = [_.cpu().detach() for _ in dis_losses]
            plt.plot(dis_losses_cpu)
            plt.title('discriminator loss')
            plt.xlabel('iterations')
            plt.ylabel('loss')
            plt.show()

            gen_losses_cpu = [_.cpu().detach() for _ in gen_losses]
            plt.plot(gen_losses_cpu)
            plt.title('generator loss')
            plt.xlabel('iterations')
            plt.ylabel('loss')
            plt.show()
    print('... Done!')



############################################################
# Prob 2-4: Find the reconstruction of a batch of samples
# **skip this part when working on problem 2-1 and come back for problem 2-4
############################################################################
# Prob 2-4: To-Do: Define squared L2-distance function (or Mean-Squared-Error)
# as reconstruction loss
############################################################################
def _reconstruction_loss(self,reconstruction, x):
  pass
  rec_loss = nn.functional.mse_loss(reconstruction,x)
  return rec_loss


def reconstruct(self, samples):
    recon_code = torch.zeros([samples.shape[0], self.code_size], device=device, re
    samples = torch.tensor(samples, device=device, dtype=torch.float32)

    # Set the generator to evaluation mode, to make batchnorm stats stay fixed
    self._generator.eval()

    ##########################################################################
```

```
        # Prob 2-4: complete the definition of the optimizer.
        # **skip this part when working on problem 2-1 and come back for problem 2-4
        ##############################################################################
        # To-Do: define the optimizer
        # Hinit: Use self.vis_learning_rate as one of the parameters for Adam optimize
        betas = (0.5, 0.999)
        self.optimizer_gen = torch.optim.Adam([recon_code], lr=self.vis_learning_rate)
        for i in range(500):
            ##############################################################################
            # Prob 2-4: Fill in the training loop for reconstruciton
            # **skip this part when working on problem 2-1 and come back for problem 2
            ##############################################################################
            # To-Do: eliminate the gradients
            self.optimizer_gen.zero_grad()

            # To-Do: feed the reconstruction codes to the generator for generating rec
            # use the variable name "recon_samples"
            recon_samples = [] # comment out this line when you are coding
            recon_samples = self._generator(recon_code)

            # To-Do: calculate reconstruction loss
            # use the variable name "recon_loss"
            recon_loss = 0.0 # comment out this line when you are coding
            recon_loss = self._reconstruction_loss(recon_samples, samples)

            # To-Do: calculate the gradient of the reconstruction loss
            recon_loss.backward()

            # To-Do: update the weights
            self.optimizer_gen.step()

            #Self Note : Result checked @Piazza : https://piazza.com/class/lcpa44ep1pk
            ##############################################################################
            #                                 END OF YOUR CODE
            ##############################################################################

        return recon_loss, recon_samples.detach().cpu()


    # Perform activation maximization on a batch of different initial codes
    def actmax(self, actmax_code):
        self._generator.eval()
        self._discriminator.eval()
        ##############################################################################
        # Prob 2-4: just check this function. You do not need to code here
        # skip this part when working on problem 2-1 and come back for problem 2-4
        ##############################################################################
        actmax_code = torch.tensor(actmax_code, device=device, dtype=torch.float32, re
        actmax_optimizer = torch.optim.Adam([actmax_code], lr=self.vis_learning_rate)
        for i in range(500):
            actmax_optimizer.zero_grad()
            actmax_sample = self._generator(actmax_code)
            actmax_dis = self._discriminator(actmax_sample)
            actmax_loss = self._classification_loss(actmax_dis, self._actmax_label)
            actmax_loss.backward()
            actmax_optimizer.step()
        return actmax_sample.detach().cpu()
```

Now let's do the training!

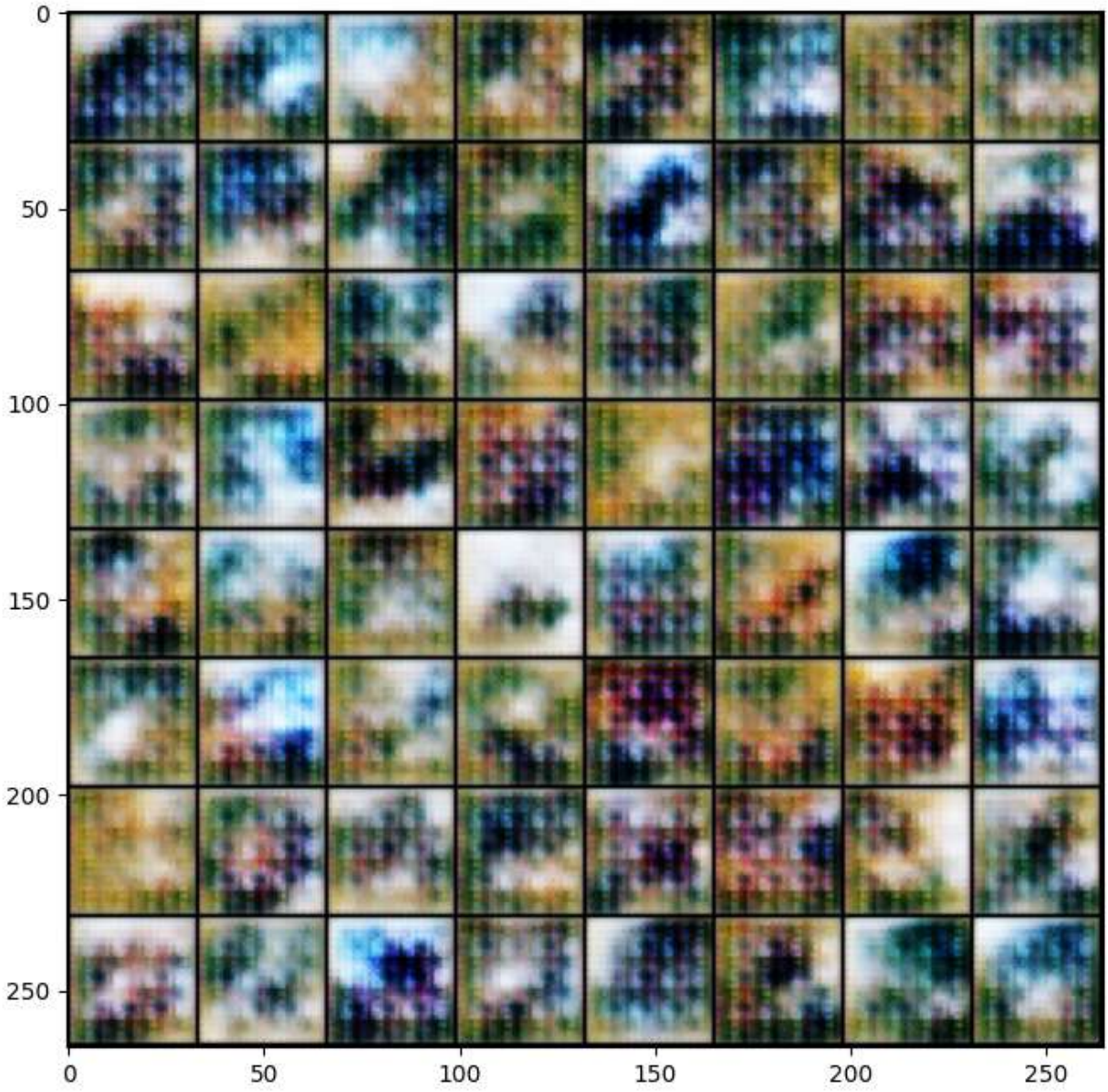
Don't panic if the loss curve goes wild. The two networks are competing for the loss curve to go different directions, so virtually anything can happen. If your code is correct, the generated samples should have a high variety.

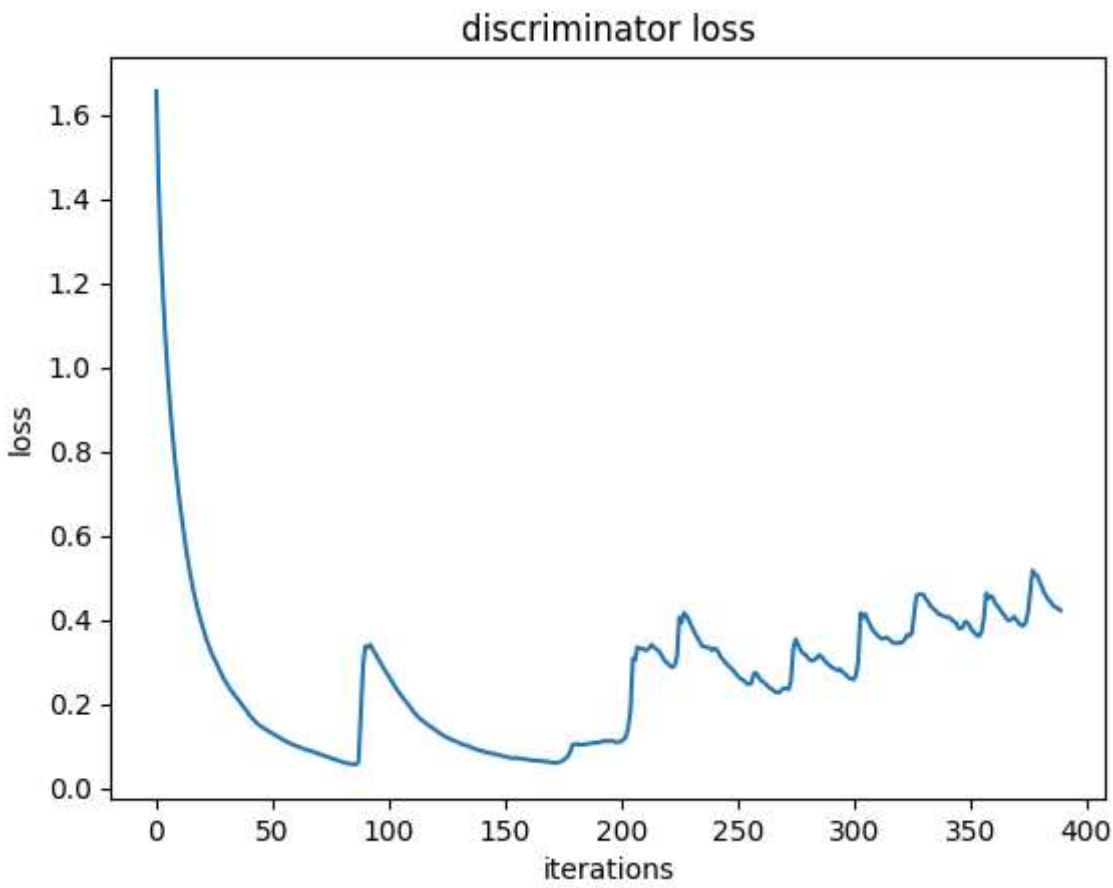
Do NOT change the number of epochs, learning rate, or batch size. If you're using Google Colab, the batch size will not be an issue during training.

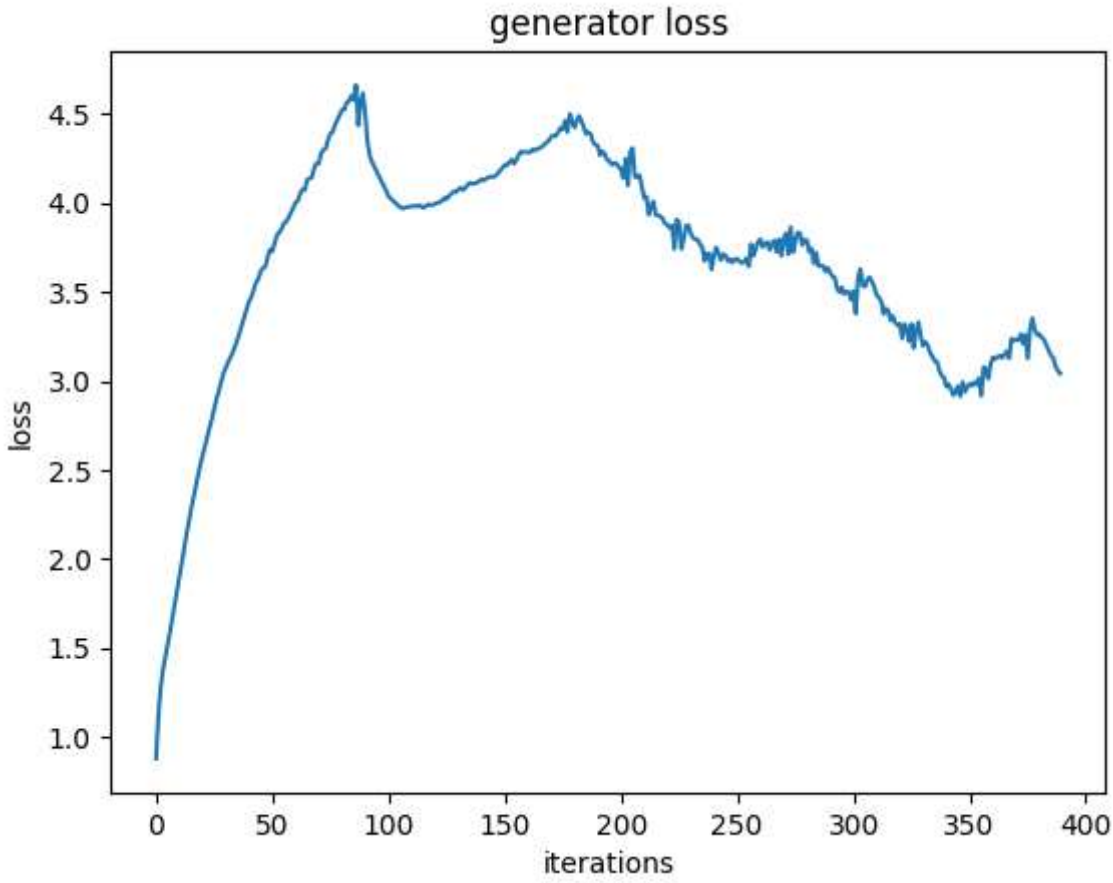
In [7]:
```
set_seed(42)

dcgan = DCGAN()
dcgan.train(train_samples)
torch.save(dcgan.state_dict(), "dcgan.pt")
```

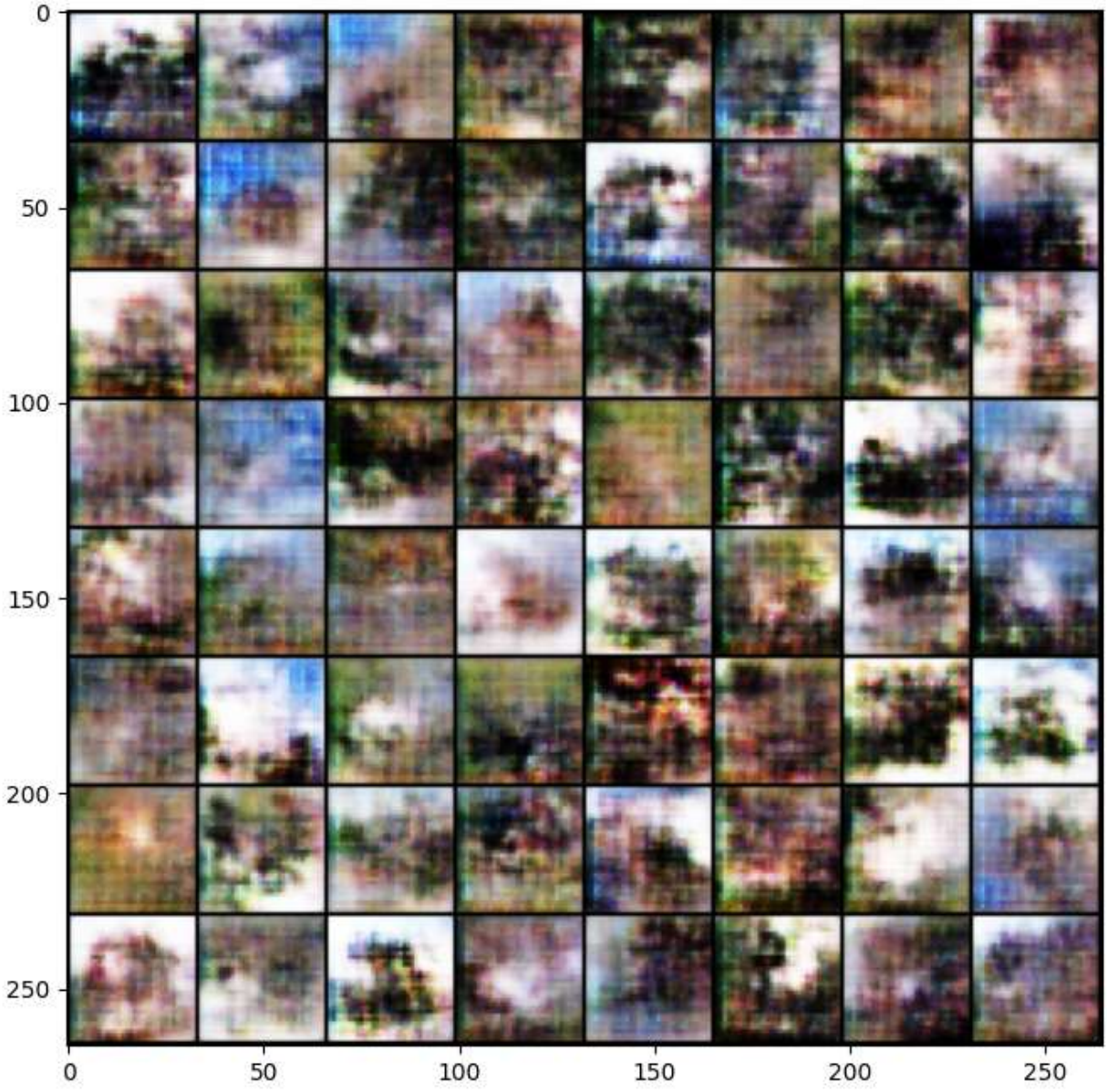
```
Start training ...
Iteration 100/9750: dis loss = 0.0953, gen loss = 3.5958
Iteration 200/9750: dis loss = 0.1105, gen loss = 3.8032
Iteration 300/9750: dis loss = 0.2278, gen loss = 2.6574
```

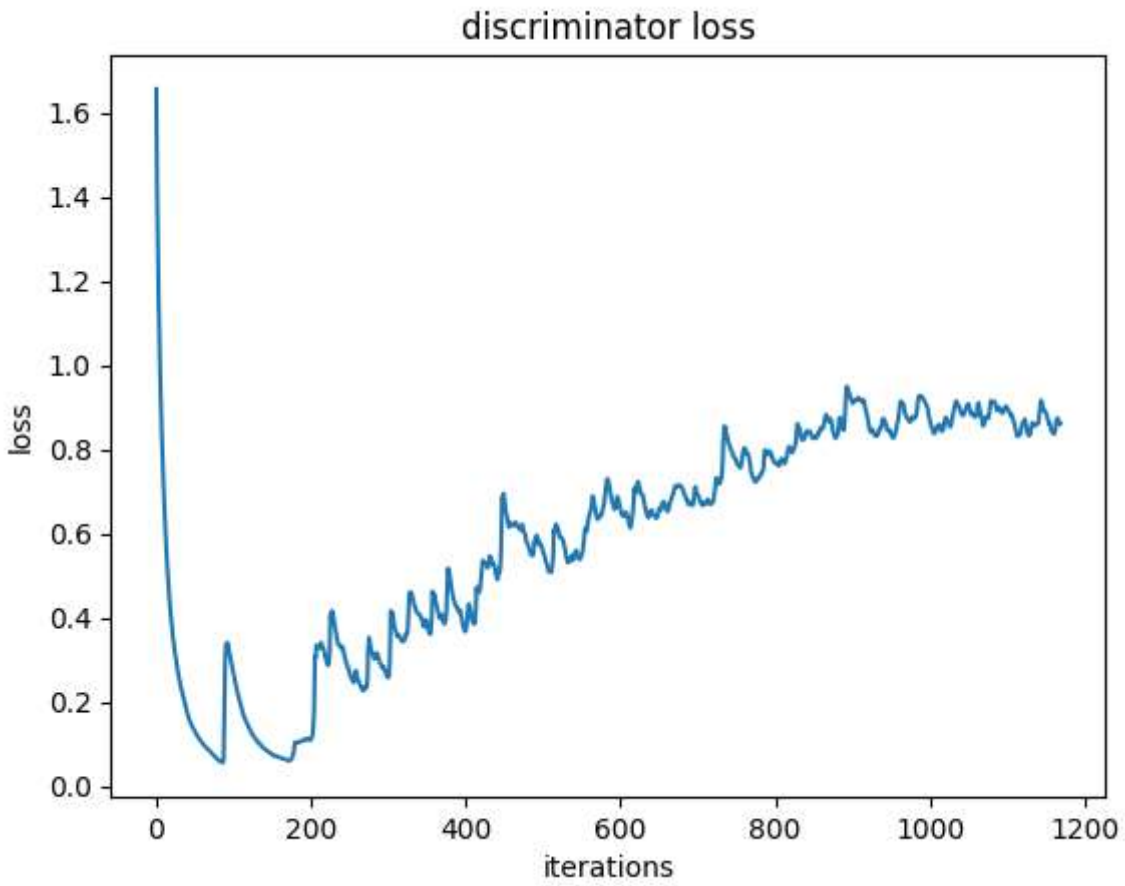


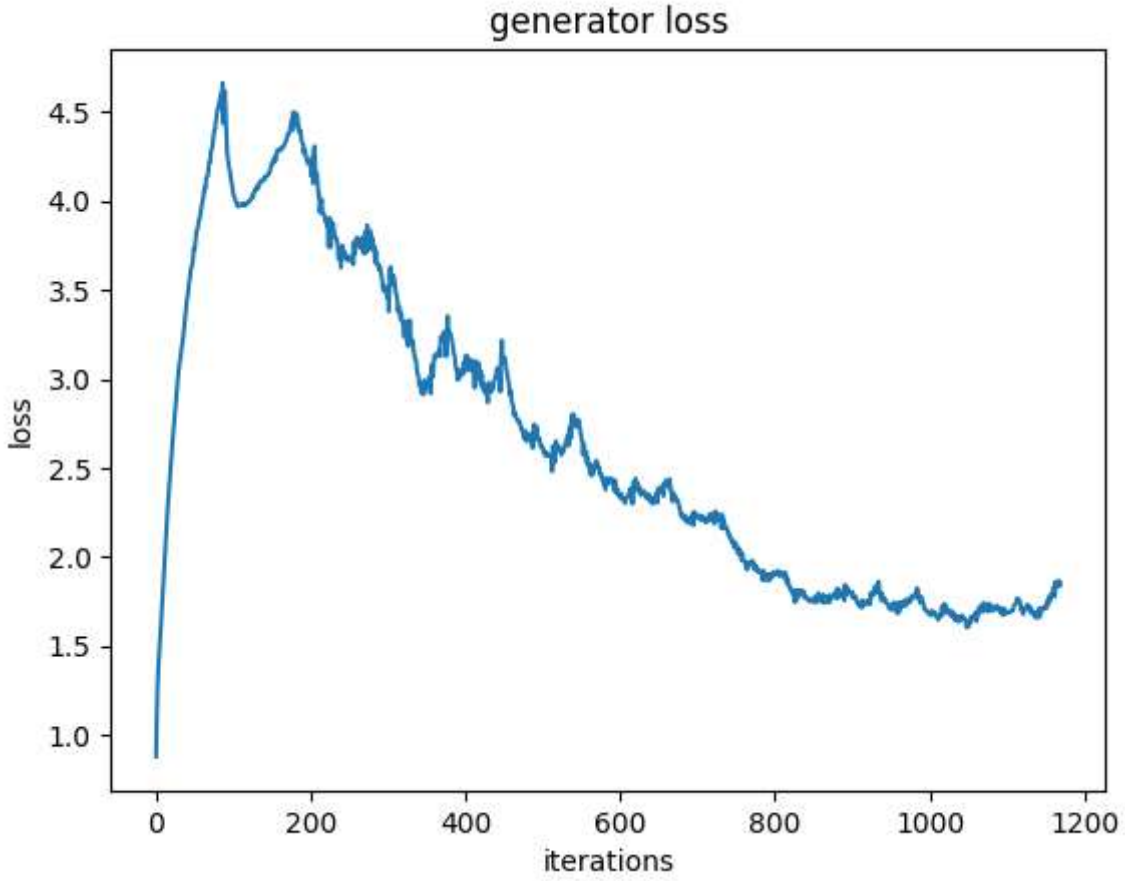

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
50
100
150
200
250
300
350
400


generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
50
100
150
200
250
300
350
400

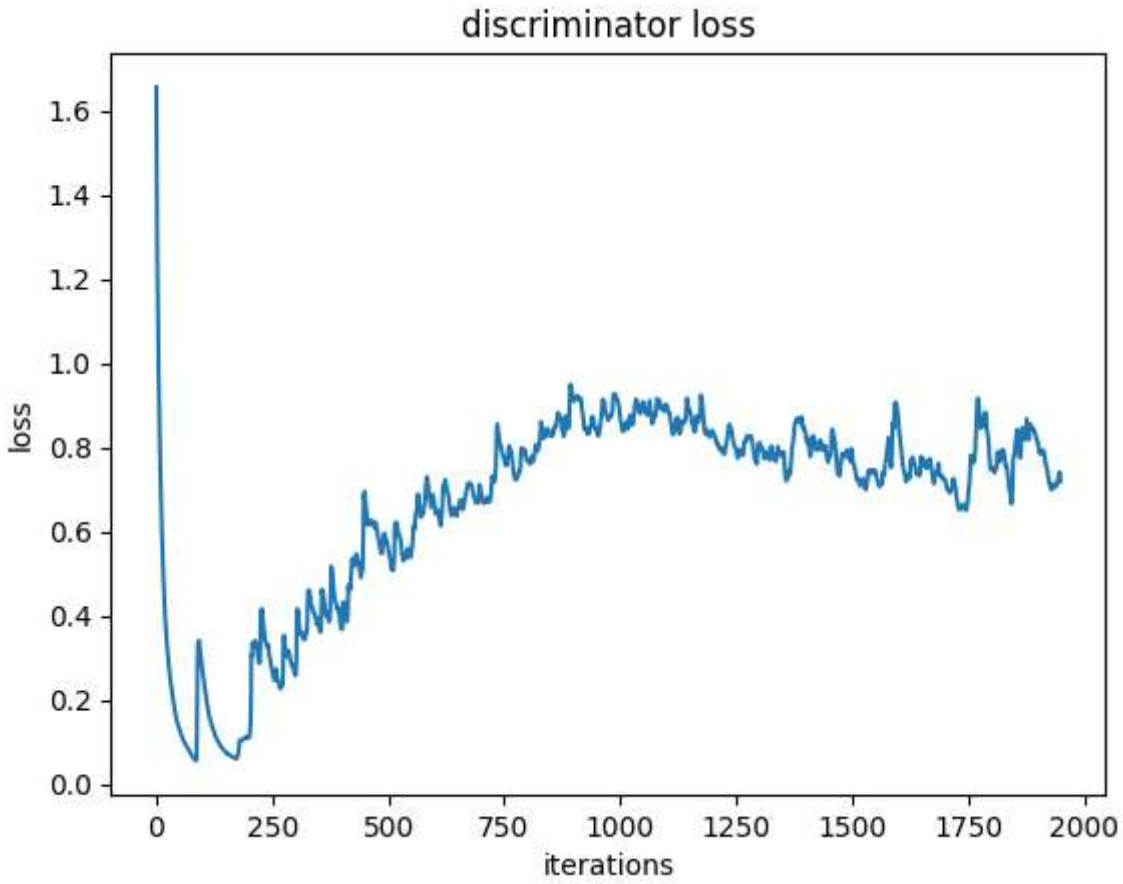
```
Iteration 400/9750: dis loss = 0.2411, gen loss = 3.2598
Iteration 500/9750: dis loss = 0.4755, gen loss = 2.3896
Iteration 600/9750: dis loss = 0.4494, gen loss = 2.6805
Iteration 700/9750: dis loss = 0.5846, gen loss = 2.4771
Iteration 800/9750: dis loss = 0.7731, gen loss = 1.5734
Iteration 900/9750: dis loss = 0.7992, gen loss = 1.6001
Iteration 1000/9750: dis loss = 0.6892, gen loss = 1.5109
Iteration 1100/9750: dis loss = 0.9042, gen loss = 1.6330
```

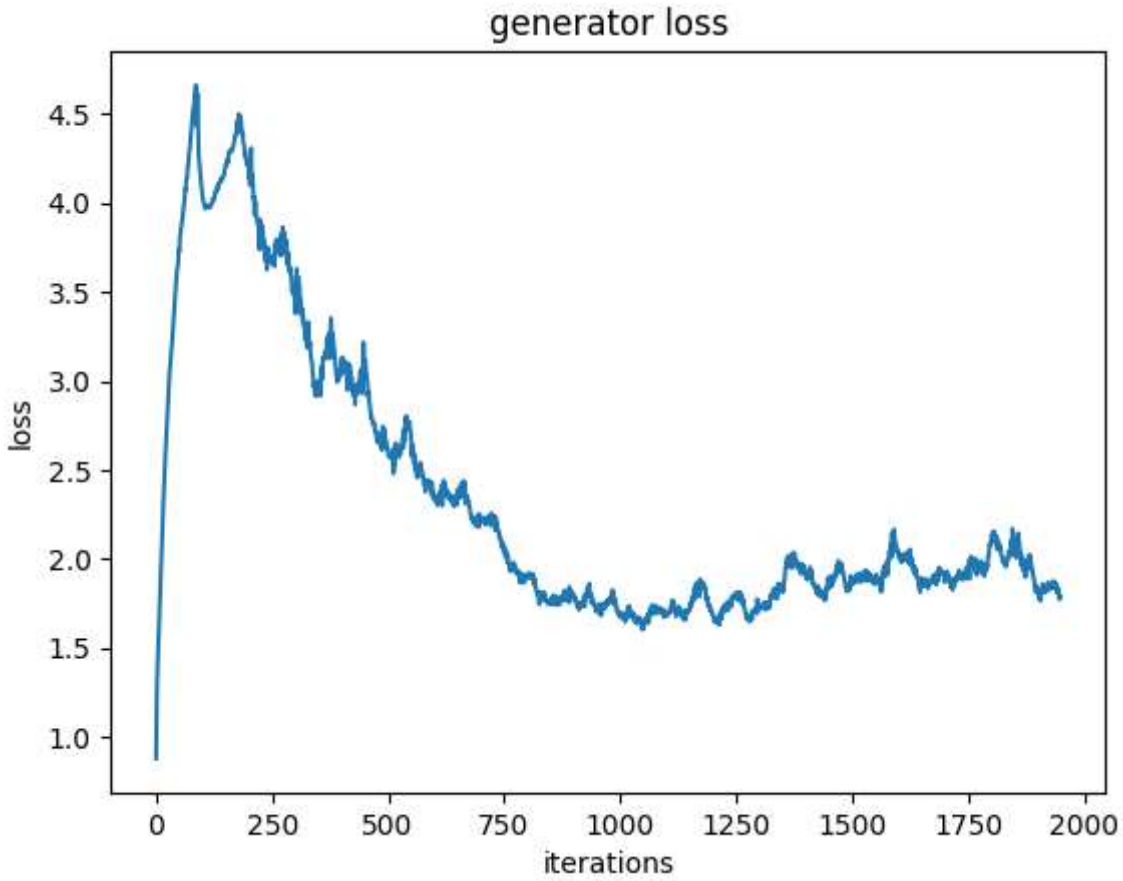



discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
200
400
600
800
1000
1200

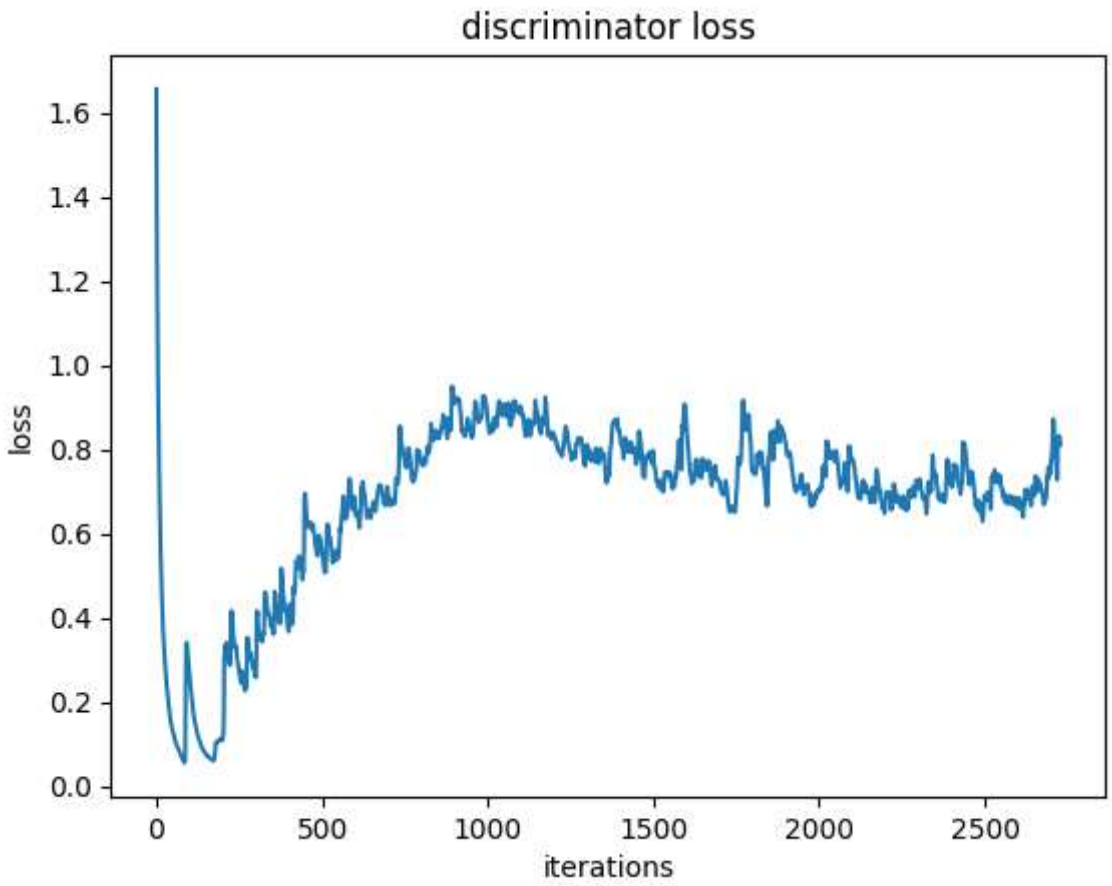

generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
200
400
600
800
1000
1200

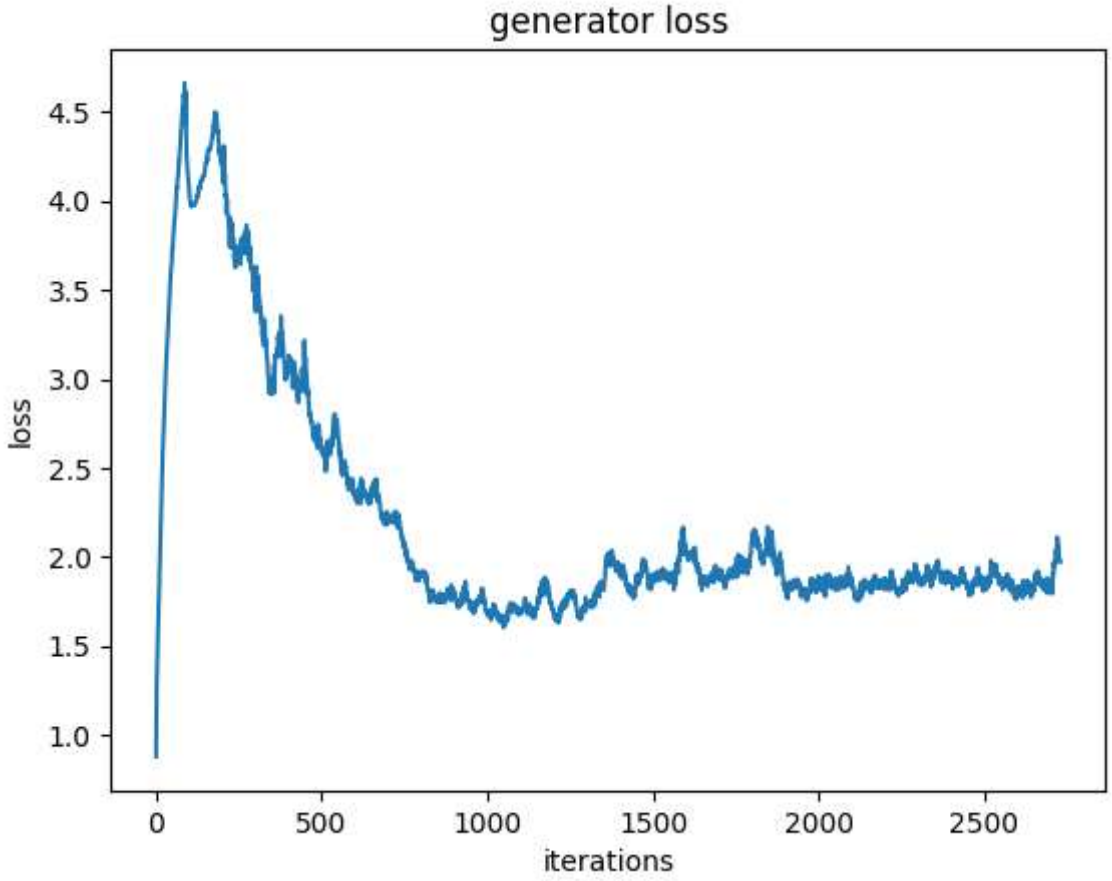
```
Iteration 1200/9750: dis loss = 0.7188, gen loss = 1.3433
Iteration 1300/9750: dis loss = 0.8105, gen loss = 1.8667
Iteration 1400/9750: dis loss = 0.7622, gen loss = 1.6995
Iteration 1500/9750: dis loss = 0.7505, gen loss = 1.8094
Iteration 1600/9750: dis loss = 0.5673, gen loss = 2.1067
Iteration 1700/9750: dis loss = 0.7281, gen loss = 1.0625
Iteration 1800/9750: dis loss = 0.8169, gen loss = 2.4069
Iteration 1900/9750: dis loss = 0.8075, gen loss = 1.4481
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
250
500
750
1000
1250
1500
1750
2000

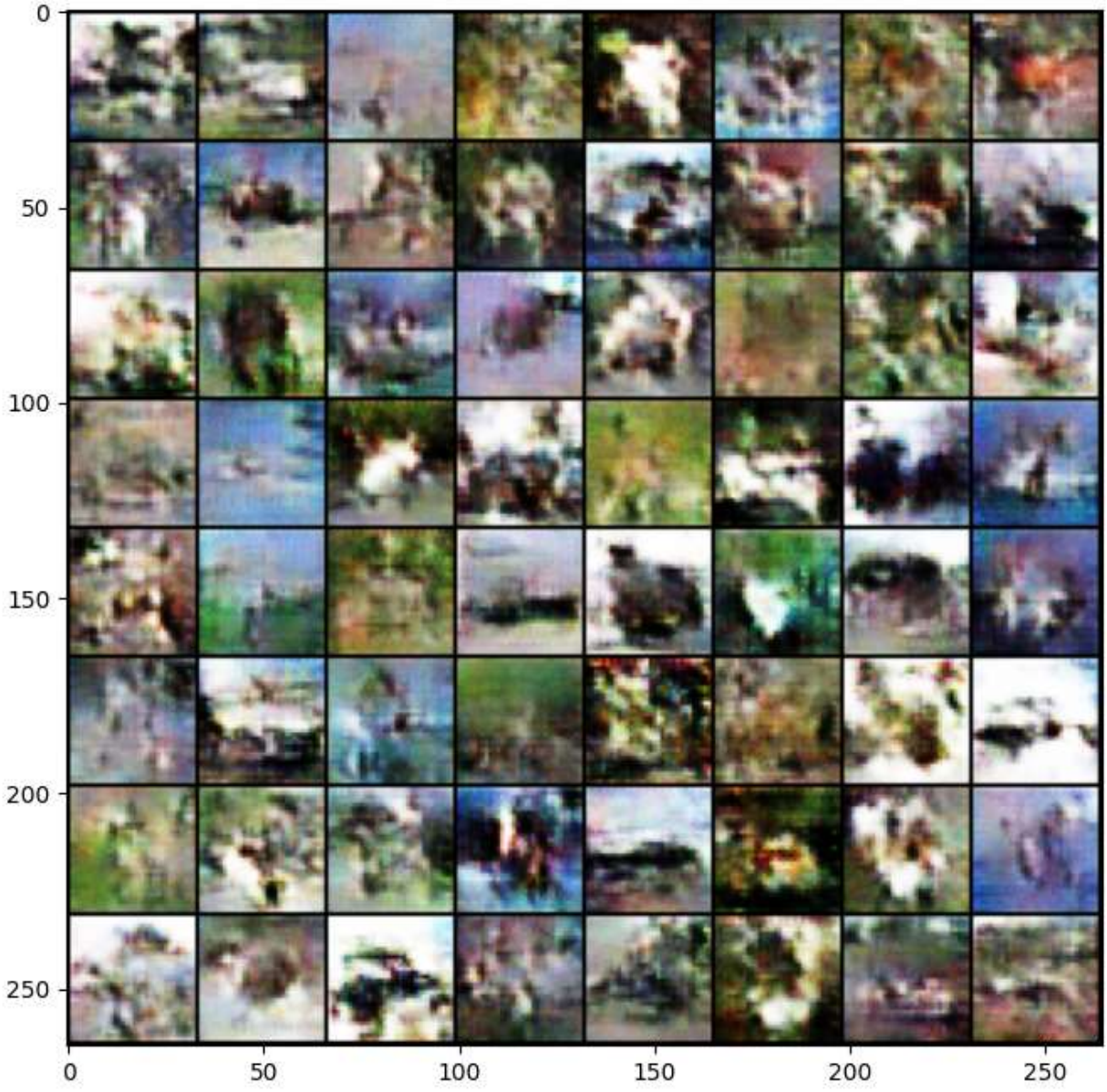

generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
250
500
750
1000
1250
1500
1750
2000

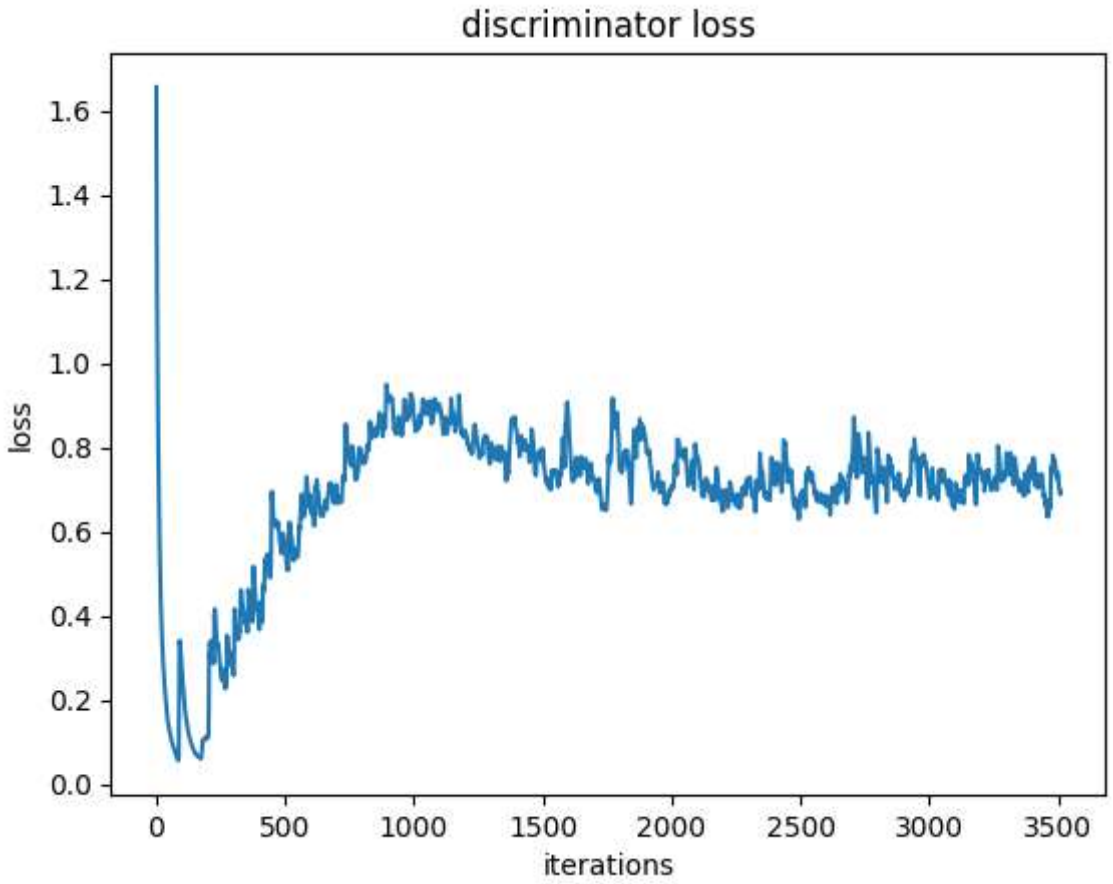
```
Iteration 2000/9750: dis loss = 0.6492, gen loss = 2.3444
Iteration 2100/9750: dis loss = 0.7097, gen loss = 2.2165
Iteration 2200/9750: dis loss = 0.7428, gen loss = 0.8236
Iteration 2300/9750: dis loss = 0.8005, gen loss = 0.9398
Iteration 2400/9750: dis loss = 0.6104, gen loss = 1.7903
Iteration 2500/9750: dis loss = 0.8580, gen loss = 1.7810
Iteration 2600/9750: dis loss = 0.6064, gen loss = 2.3278
Iteration 2700/9750: dis loss = 1.0269, gen loss = 2.4842
```

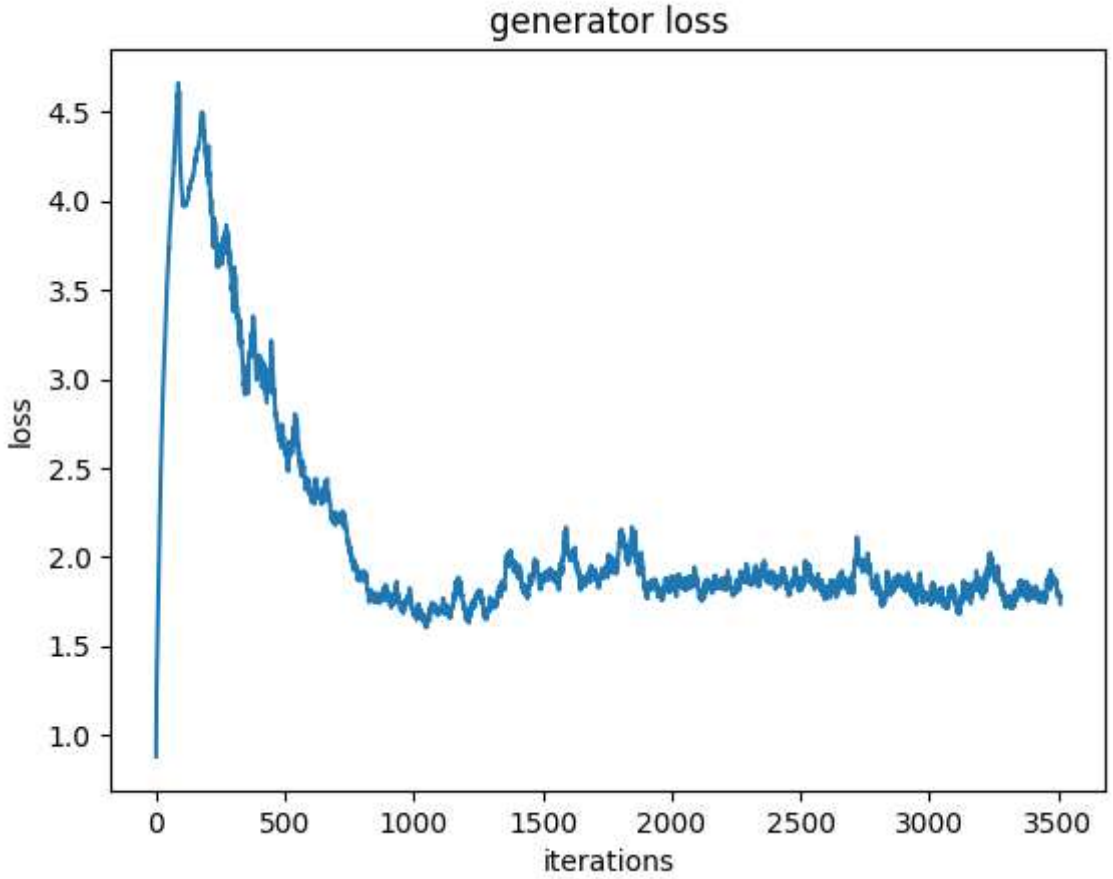

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
500
1000
1500
2000
2500


generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
500
1000
1500
2000
2500

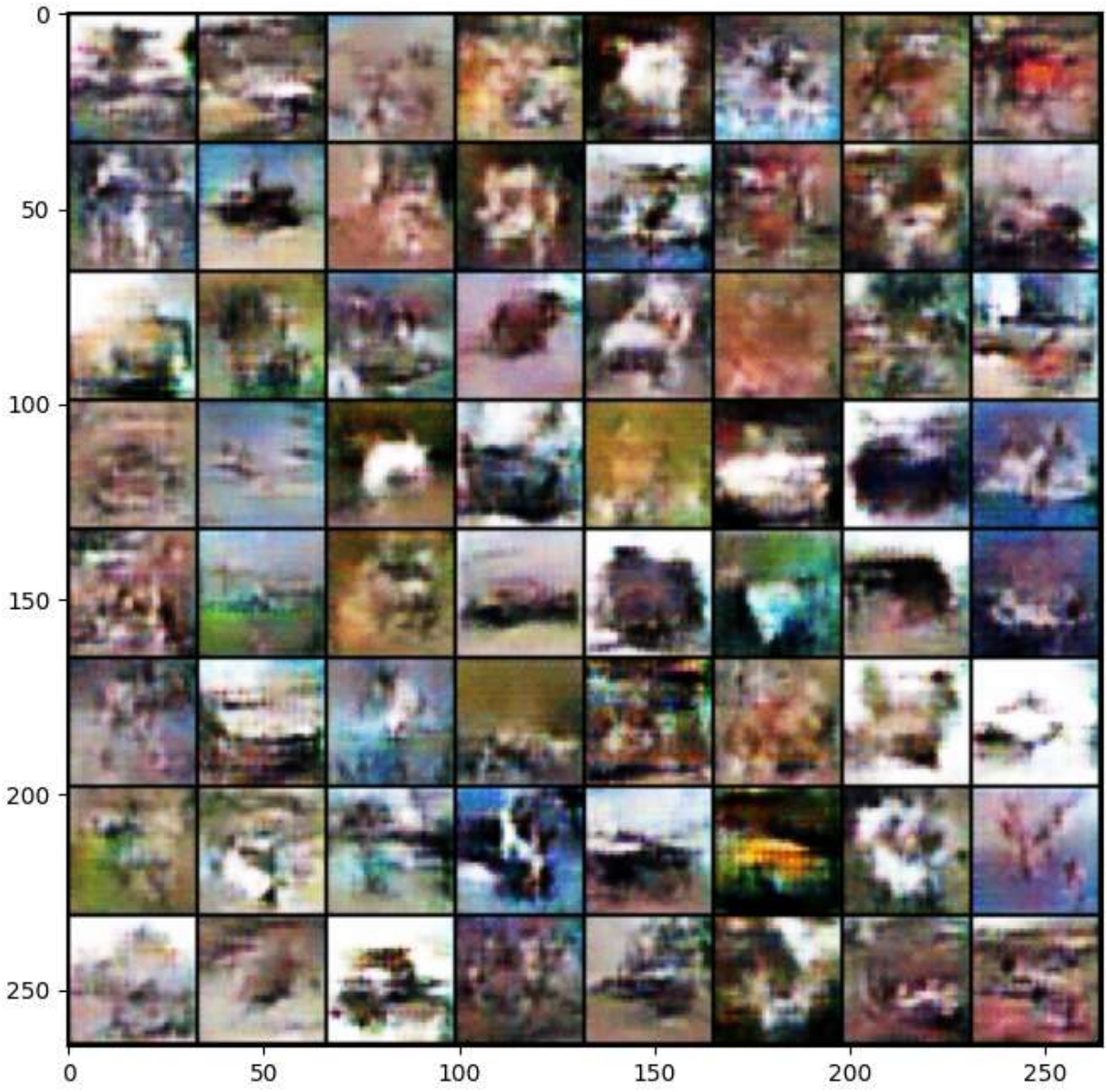
```
Iteration 2800/9750: dis loss = 0.7116, gen loss = 0.7718
Iteration 2900/9750: dis loss = 0.7384, gen loss = 2.4888
Iteration 3000/9750: dis loss = 0.5600, gen loss = 2.5261
Iteration 3100/9750: dis loss = 0.7684, gen loss = 0.7016
Iteration 3200/9750: dis loss = 0.8099, gen loss = 1.2117
Iteration 3300/9750: dis loss = 0.9215, gen loss = 2.0010
Iteration 3400/9750: dis loss = 0.8146, gen loss = 2.0311
Iteration 3500/9750: dis loss = 0.6292, gen loss = 1.3984
```

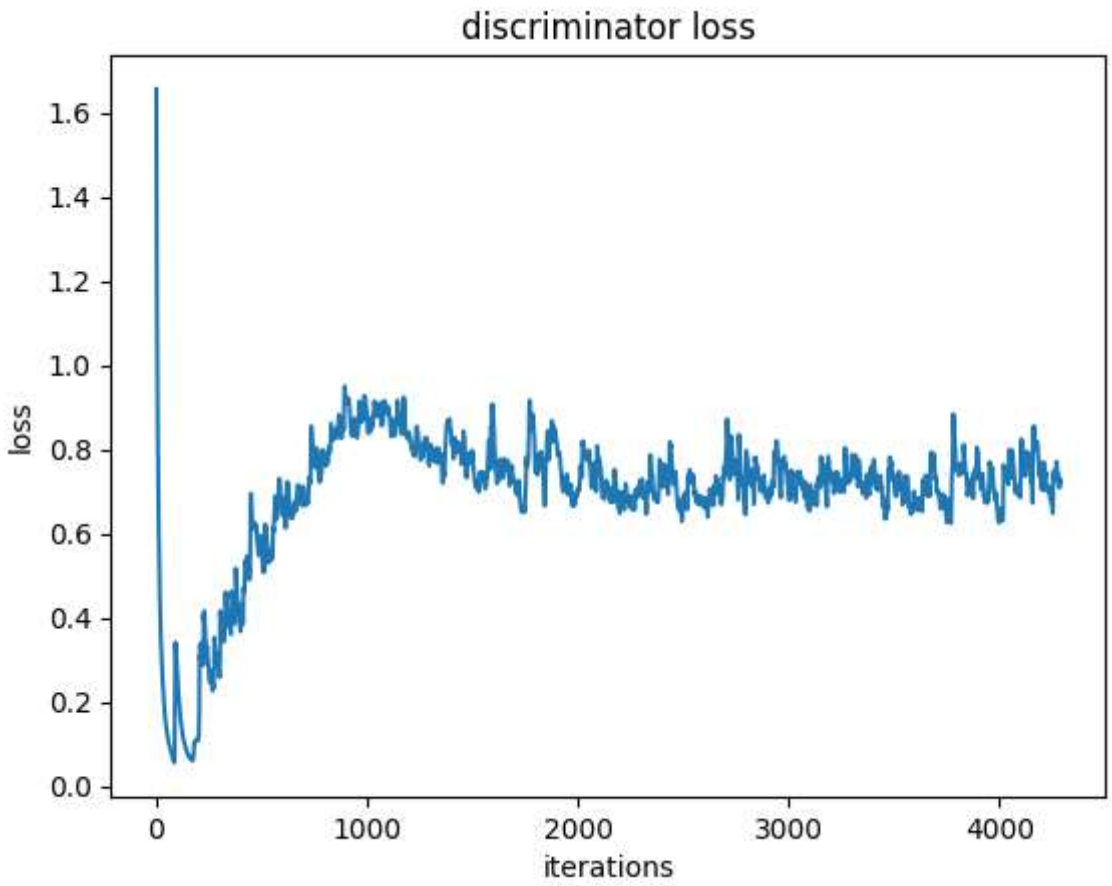


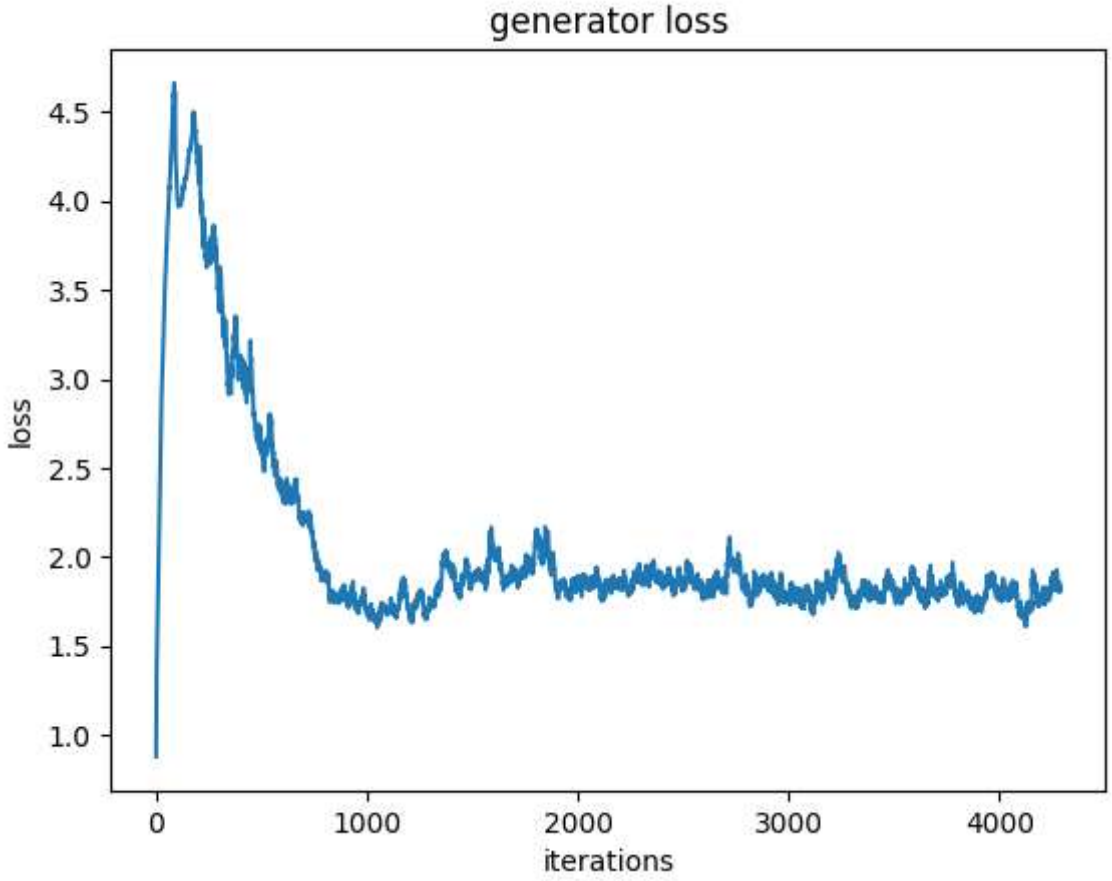

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
500
1000
1500
2000
2500
3000
3500


generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
500
1000
1500
2000
2500
3000
3500

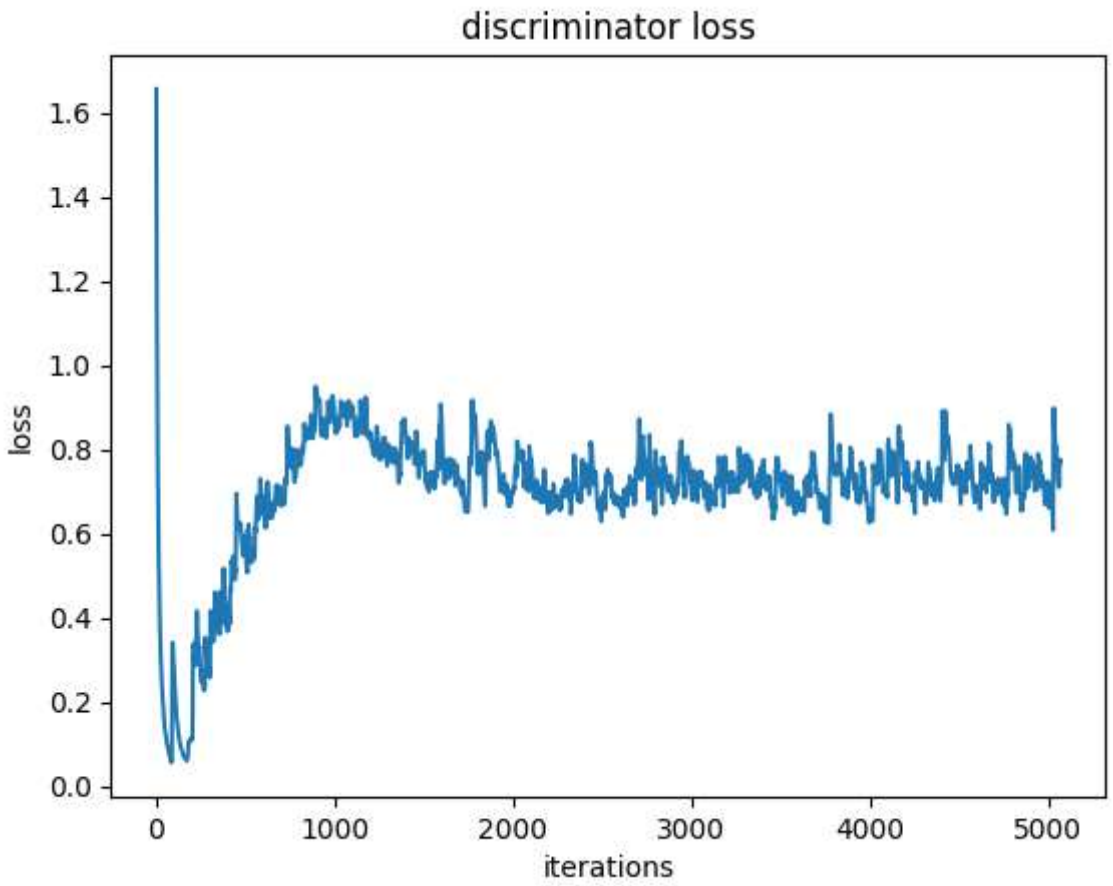
```
Iteration 3600/9750: dis loss = 0.6065, gen loss = 1.1370
Iteration 3700/9750: dis loss = 0.6248, gen loss = 1.4318
Iteration 3800/9750: dis loss = 0.6810, gen loss = 1.8791
Iteration 3900/9750: dis loss = 0.5437, gen loss = 2.0844
Iteration 4000/9750: dis loss = 0.7945, gen loss = 0.7114
Iteration 4100/9750: dis loss = 1.3948, gen loss = 0.1828
Iteration 4200/9750: dis loss = 0.5138, gen loss = 2.1947
```

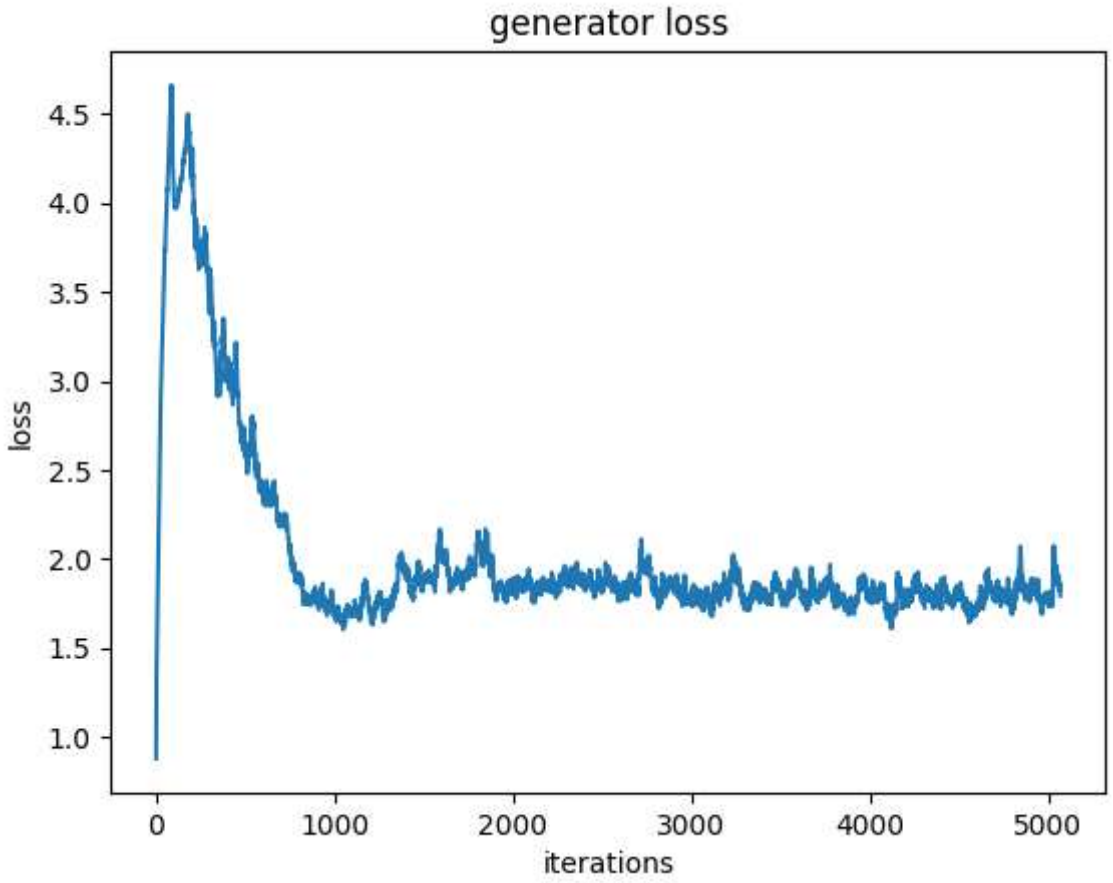



discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000

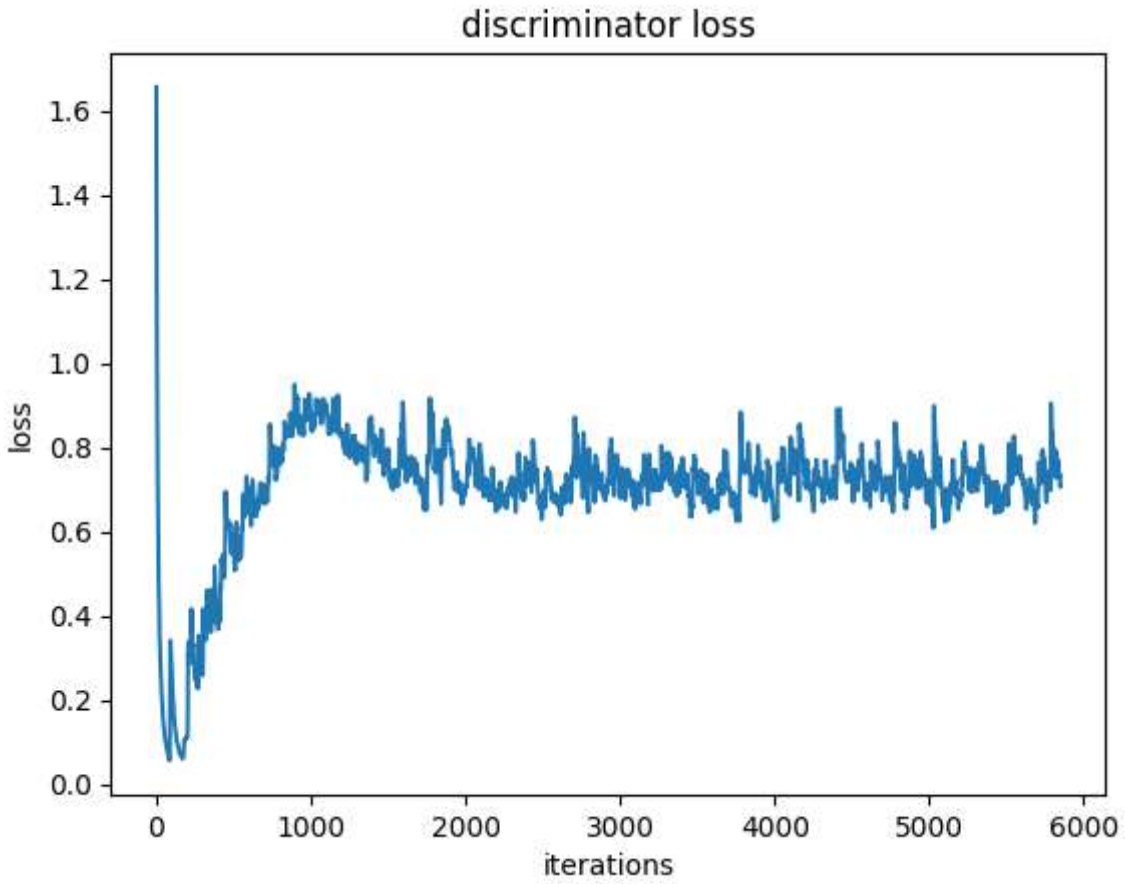

generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
1000
2000
3000
4000

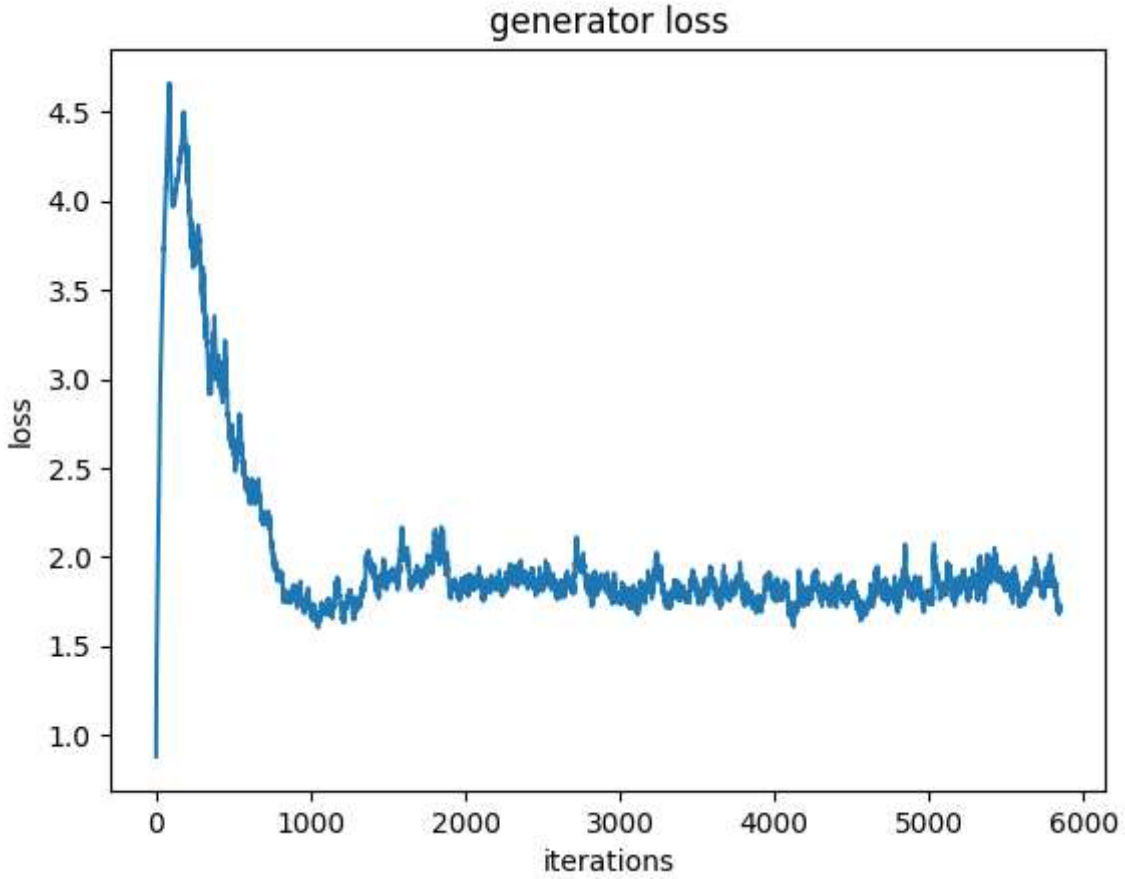
```
Iteration 4300/9750: dis loss = 0.7646, gen loss = 1.0014
Iteration 4400/9750: dis loss = 0.8287, gen loss = 1.8302
Iteration 4500/9750: dis loss = 0.5478, gen loss = 2.1332
Iteration 4600/9750: dis loss = 0.6272, gen loss = 1.7995
Iteration 4700/9750: dis loss = 0.9461, gen loss = 1.2611
Iteration 4800/9750: dis loss = 0.6889, gen loss = 2.4528
Iteration 4900/9750: dis loss = 1.0661, gen loss = 2.3733
Iteration 5000/9750: dis loss = 0.5158, gen loss = 2.3432
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000

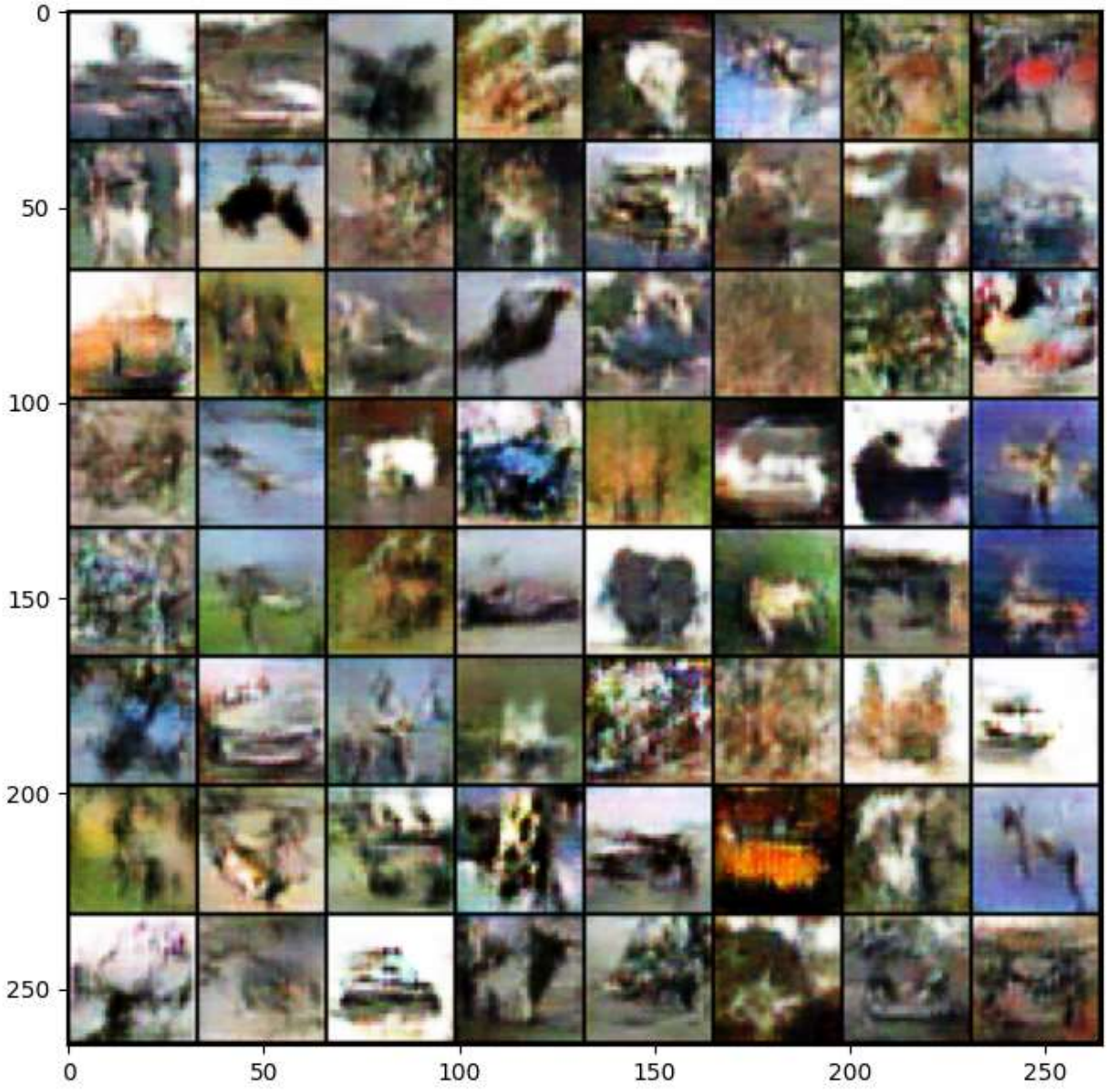

generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
1000
2000
3000
4000
5000

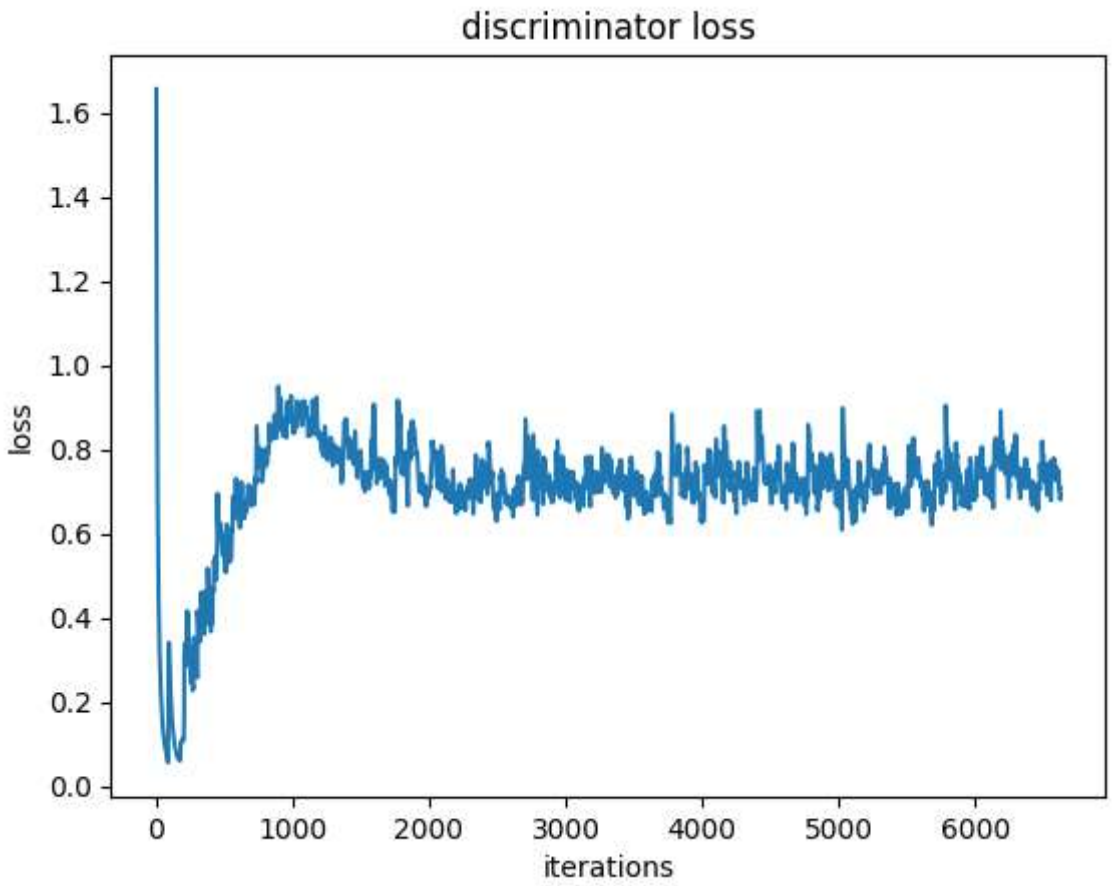
```
Iteration 5100/9750: dis loss = 0.6226, gen loss = 1.7267
Iteration 5200/9750: dis loss = 0.6259, gen loss = 1.2621
Iteration 5300/9750: dis loss = 0.7424, gen loss = 0.9041
Iteration 5400/9750: dis loss = 0.6071, gen loss = 2.0161
Iteration 5500/9750: dis loss = 0.5766, gen loss = 2.0669
Iteration 5600/9750: dis loss = 0.7845, gen loss = 2.7495
Iteration 5700/9750: dis loss = 0.6107, gen loss = 1.2174
Iteration 5800/9750: dis loss = 0.7183, gen loss = 1.8444
```

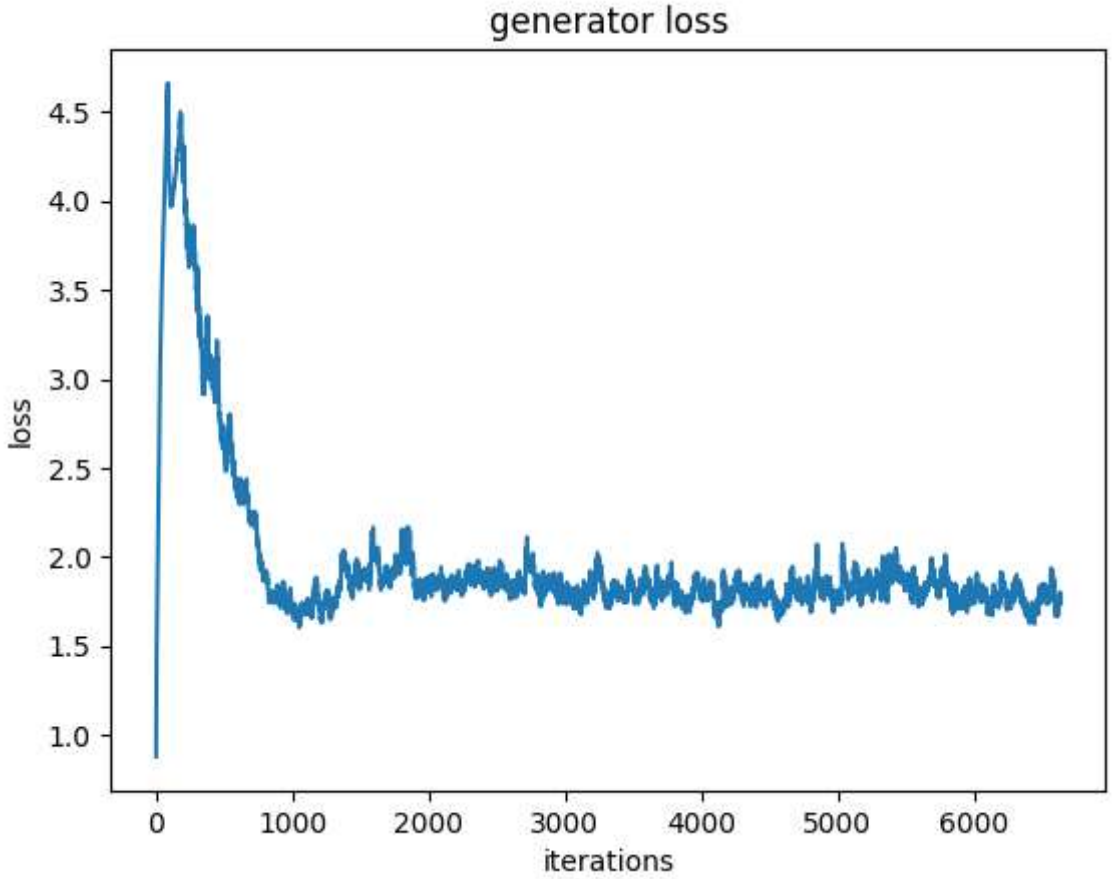

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000
6000


generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
1000
2000
3000
4000
5000
6000

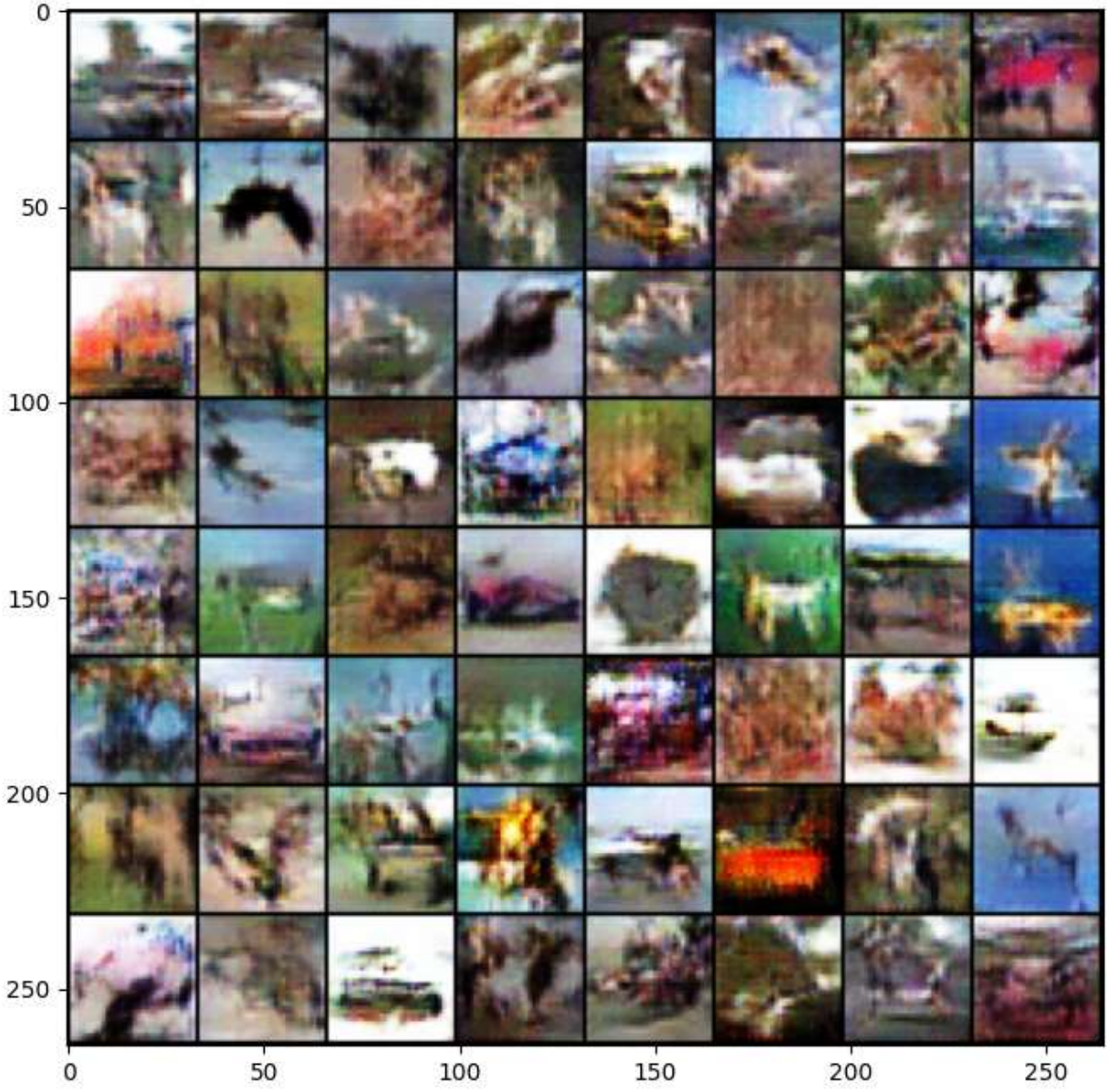
```
Iteration 5900/9750: dis loss = 0.4918, gen loss = 2.0310
Iteration 6000/9750: dis loss = 0.5834, gen loss = 2.0547
Iteration 6100/9750: dis loss = 0.6690, gen loss = 1.4920
Iteration 6200/9750: dis loss = 0.8586, gen loss = 1.9981
Iteration 6300/9750: dis loss = 1.3219, gen loss = 1.0969
Iteration 6400/9750: dis loss = 0.7523, gen loss = 1.4249
Iteration 6500/9750: dis loss = 0.7433, gen loss = 1.3394
Iteration 6600/9750: dis loss = 0.7925, gen loss = 0.9170
```

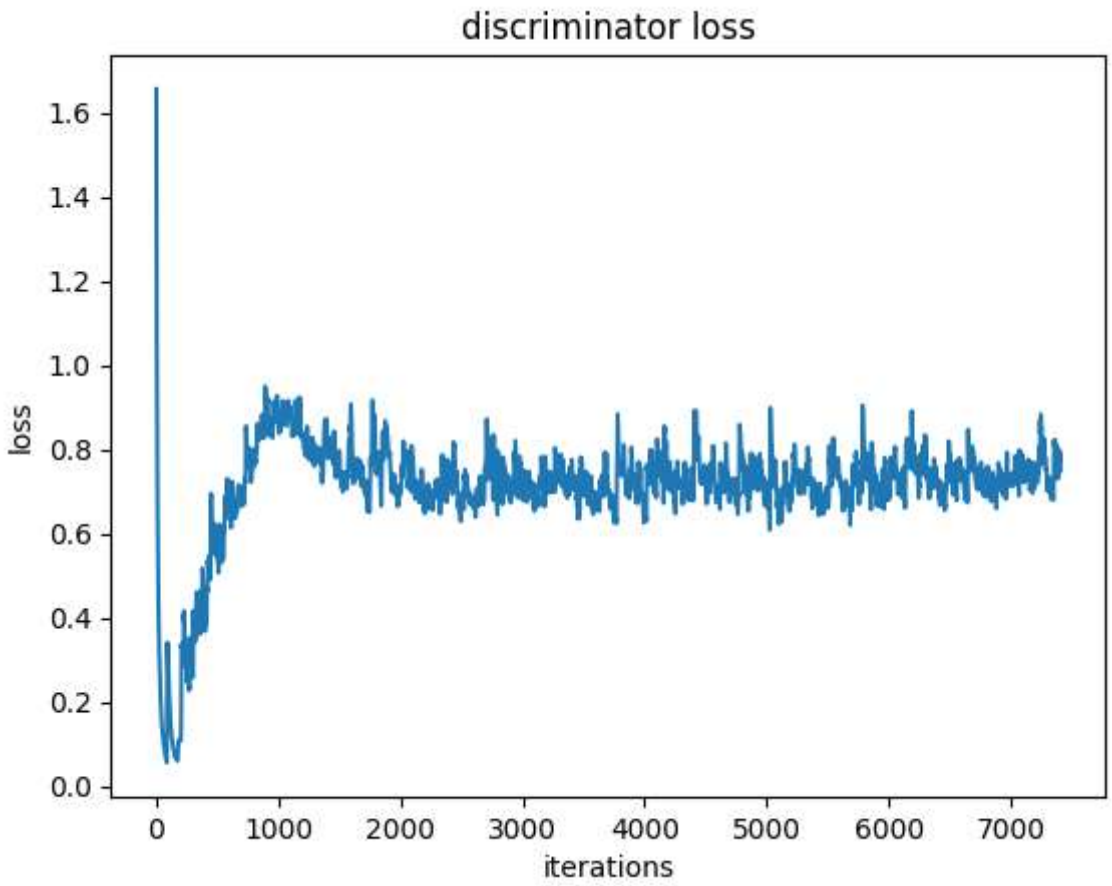


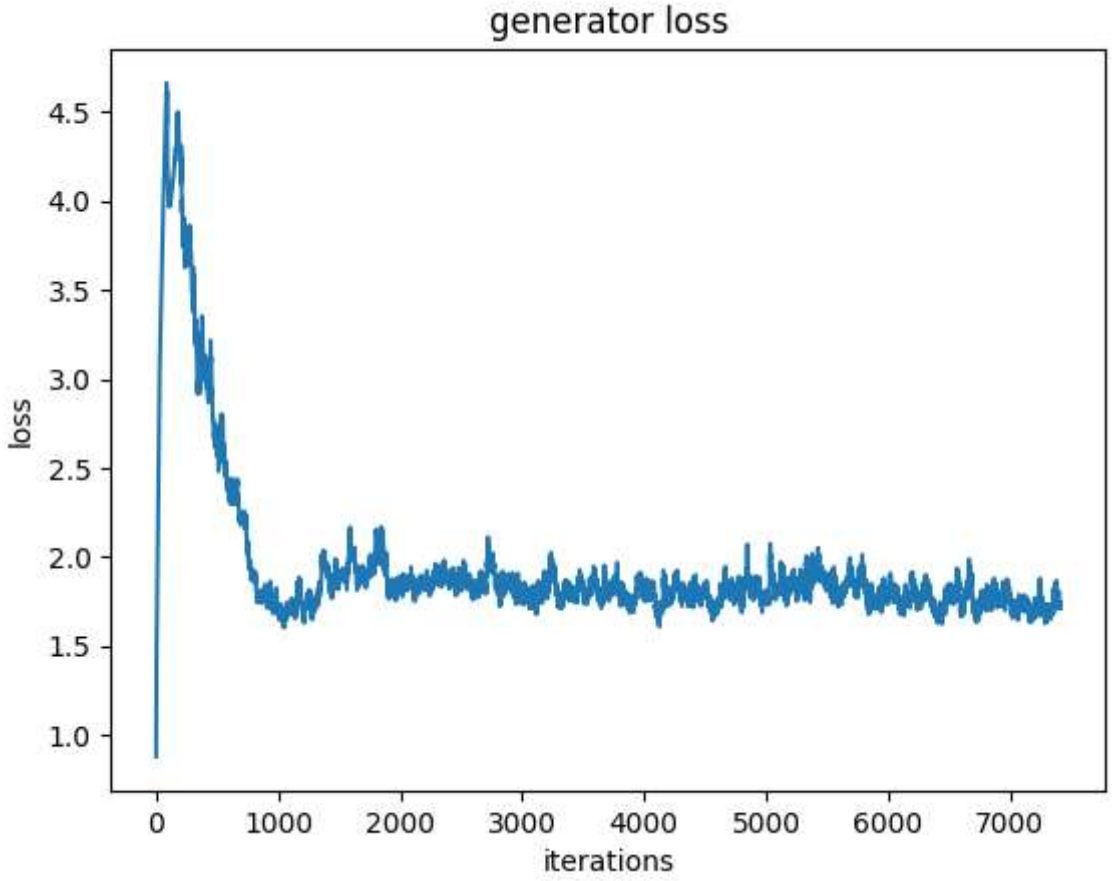

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000
6000


generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
1000
2000
3000
4000
5000
6000

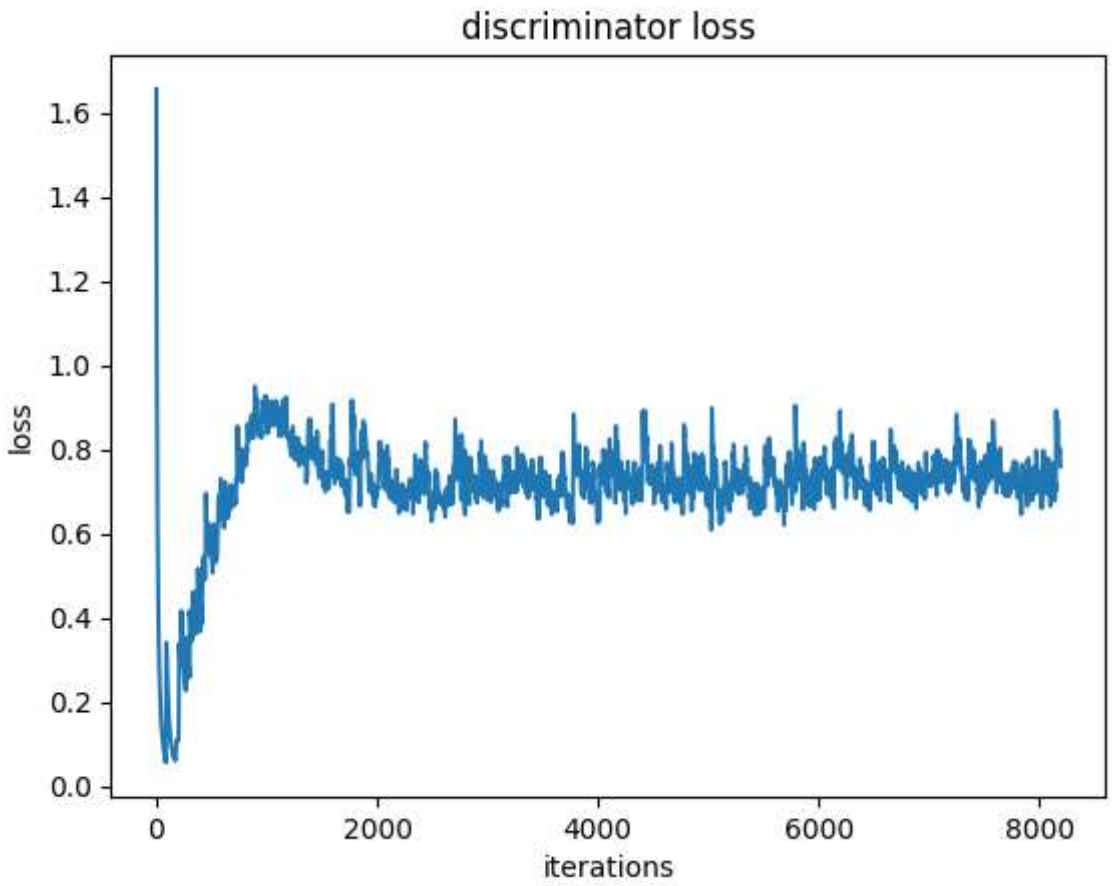
```
Iteration 6700/9750: dis loss = 0.9115, gen loss = 0.6281
Iteration 6800/9750: dis loss = 0.8106, gen loss = 1.2545
Iteration 6900/9750: dis loss = 1.1449, gen loss = 0.6257
Iteration 7000/9750: dis loss = 0.6973, gen loss = 2.0166
Iteration 7100/9750: dis loss = 0.6274, gen loss = 2.3588
Iteration 7200/9750: dis loss = 0.9542, gen loss = 0.5797
Iteration 7300/9750: dis loss = 0.5640, gen loss = 2.1212
Iteration 7400/9750: dis loss = 0.4236, gen loss = 2.9073
```

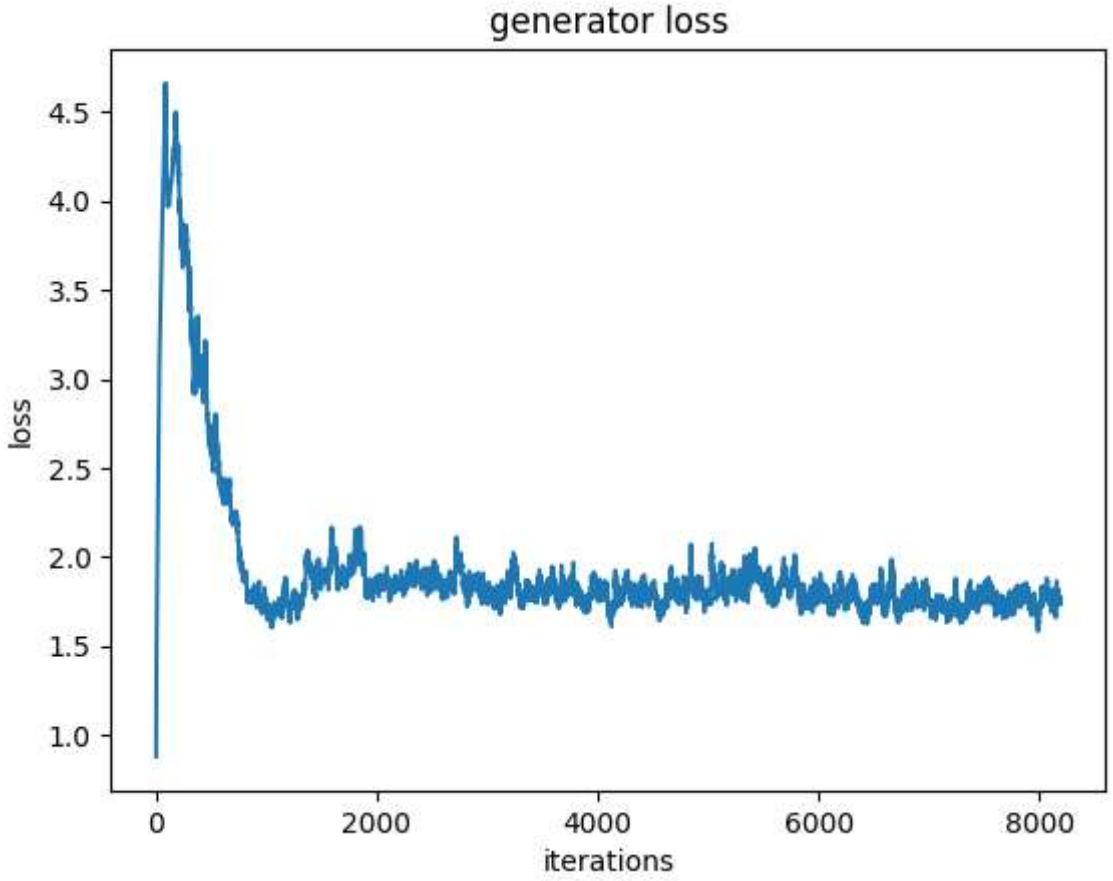



discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000
6000
7000

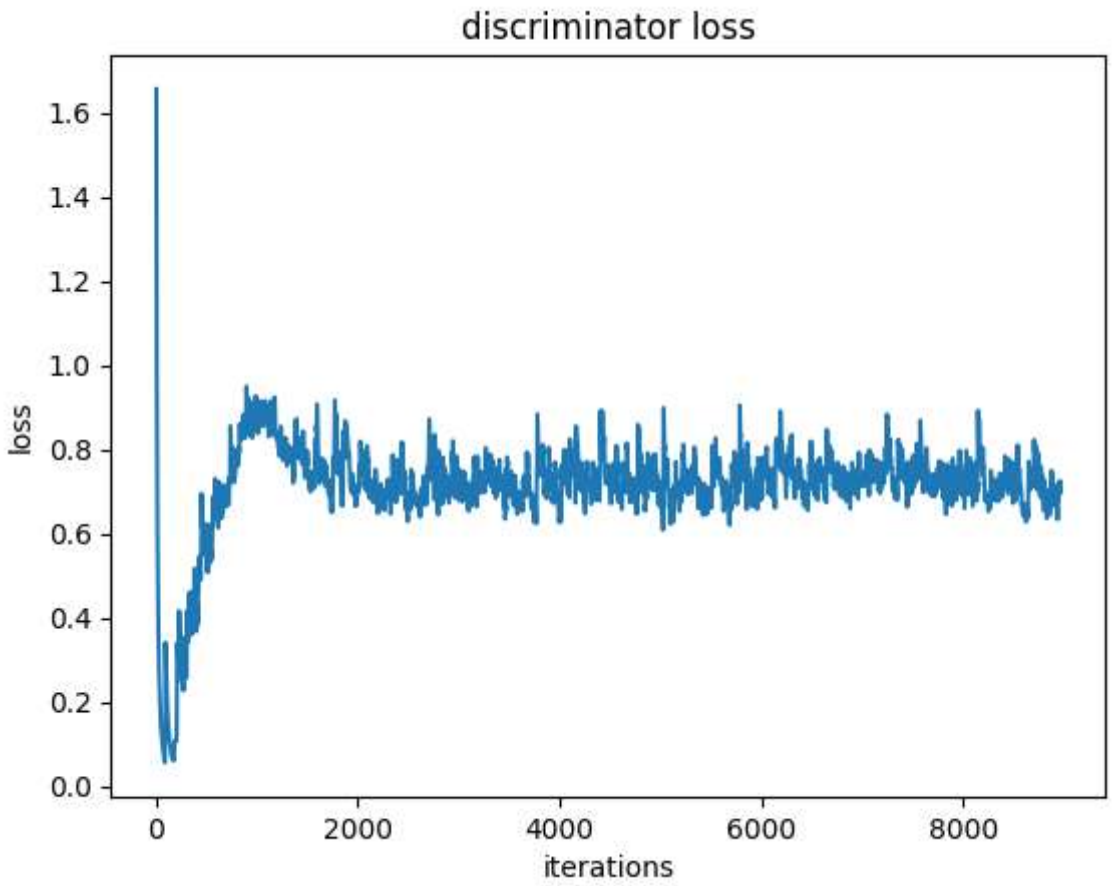

generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
1000
2000
3000
4000
5000
6000
7000

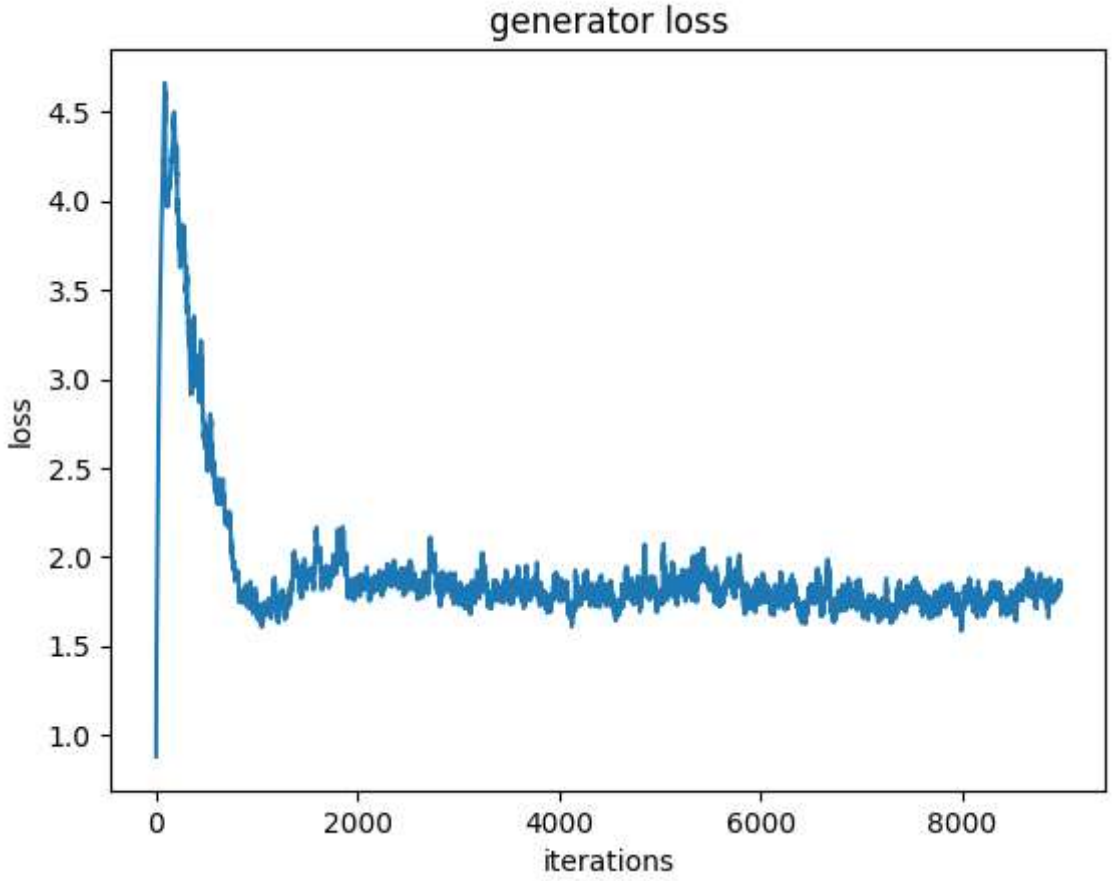
```
Iteration 7500/9750: dis loss = 1.0016, gen loss = 2.9740
Iteration 7600/9750: dis loss = 0.6582, gen loss = 1.6531
Iteration 7700/9750: dis loss = 1.0099, gen loss = 2.5040
Iteration 7800/9750: dis loss = 0.6249, gen loss = 2.1127
Iteration 7900/9750: dis loss = 0.9093, gen loss = 0.6909
Iteration 8000/9750: dis loss = 0.6810, gen loss = 0.9040
Iteration 8100/9750: dis loss = 0.9112, gen loss = 0.5876
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
2000
4000
6000
8000

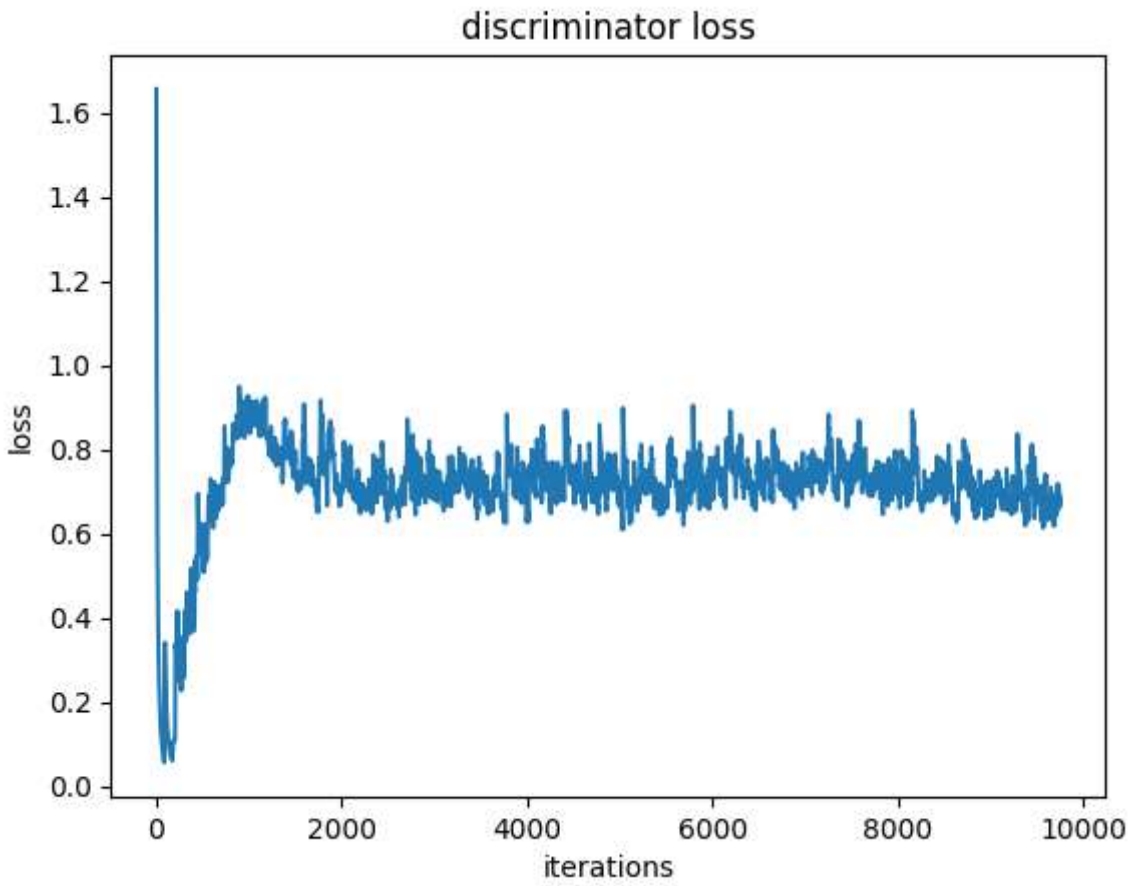

generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
2000
4000
6000
8000

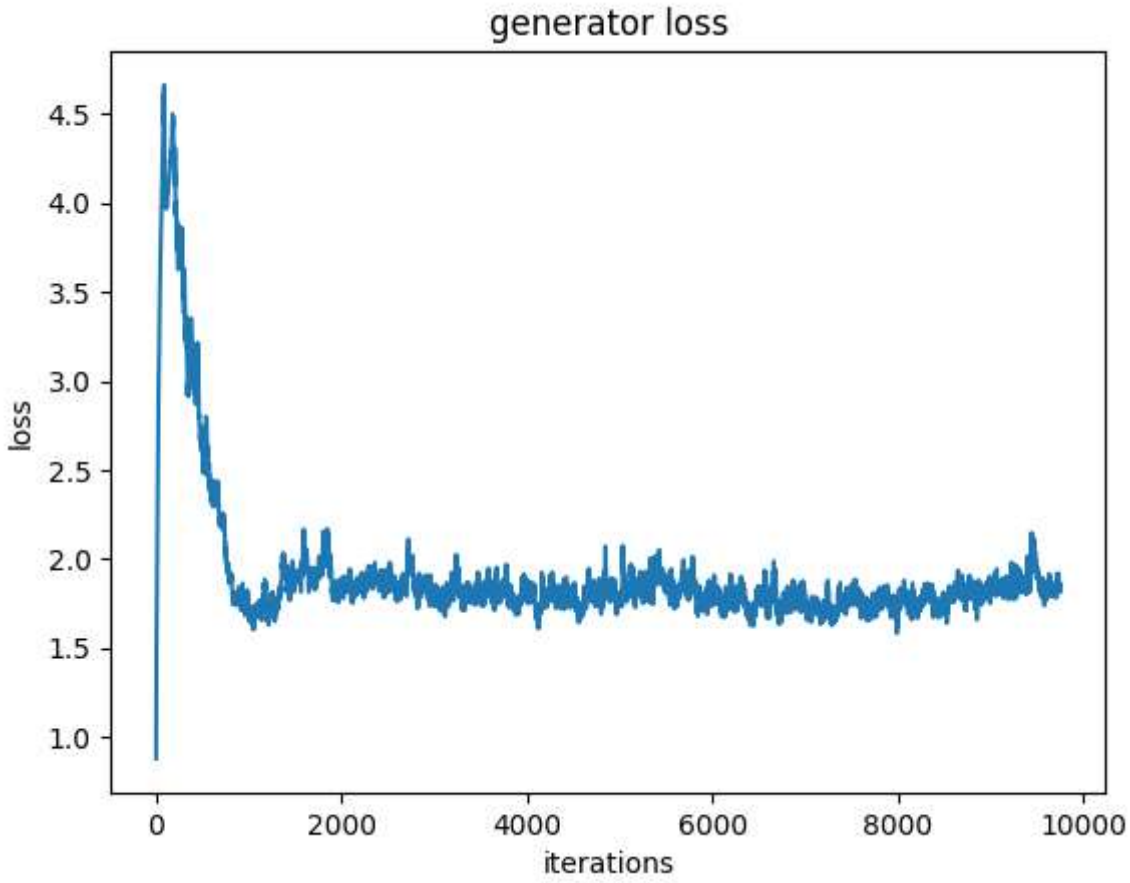
```
Iteration 8200/9750: dis loss = 0.6928, gen loss = 2.0903
Iteration 8300/9750: dis loss = 0.5625, gen loss = 1.2250
Iteration 8400/9750: dis loss = 0.8917, gen loss = 2.9418
Iteration 8500/9750: dis loss = 0.8411, gen loss = 2.5410
Iteration 8600/9750: dis loss = 0.8506, gen loss = 3.3851
Iteration 8700/9750: dis loss = 0.6647, gen loss = 1.9377
Iteration 8800/9750: dis loss = 0.5977, gen loss = 1.3303
Iteration 8900/9750: dis loss = 0.7185, gen loss = 2.3143
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
2000
4000
6000
8000


generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
2000
4000
6000
8000

```
Iteration 9000/9750: dis loss = 0.8186, gen loss = 1.2216
Iteration 9100/9750: dis loss = 0.8327, gen loss = 1.9684
Iteration 9200/9750: dis loss = 0.6694, gen loss = 2.2542
Iteration 9300/9750: dis loss = 0.6159, gen loss = 1.6024
Iteration 9400/9750: dis loss = 1.0299, gen loss = 0.7974
Iteration 9500/9750: dis loss = 0.7199, gen loss = 1.2209
Iteration 9600/9750: dis loss = 0.4325, gen loss = 2.0841
Iteration 9700/9750: dis loss = 0.5072, gen loss = 1.7121
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
2000
4000
6000
8000
10000


generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
2000
4000
6000
8000
10000

```
... Done!
```

Problem 2-2: The Batch Normalization dilemma (4 pts)

Here are two questions related to the use of Batch Normalization in GANs. Q1 below will not be graded and the answer is provided. But you should attempt to solve it before looking at the answer.

Q2 will be graded.

Q1: We made separate batches for real samples and fake samples when training the discriminator. Is this just an arbitrary design decision made by the inventor that later becomes the common practice, or is it critical to the correctness of the algorithm? **[0 pt]**

Answer to Q1: When we are training the generator, the input batch to the discriminator will always consist of only fake samples. If we separate real and fake batches when training the discriminator, then the fake samples are normalized in the same way when we are training the discriminator and when we are training the generator. If we mix real and fake samples in the same batch when training the discriminator, then the fake samples are not normalized in the same way when we train the two networks, which causes the generator to fail to learn the correct distribution.

Q2: Look at the construction of the discriminator carefully. You will find that between dis_conv1 and dis_lrelu1 there is no batch normalization. This is not a mistake. What could go wrong if there were a batch normalization layer there? Why do you think that omitting this batch normalization layer solves the problem practically if not theoretically? **[3 pt]**

Please provide your answer to Q2:

If a batch normalization layer were added between the first convolutional layer and the first LeakyReLU activation function in the Discriminator, it could lead to issues such as instability in the training process and poor quality of generated samples.

- Batch normalization works by normalizing the activations of a given layer by subtracting the mean and dividing by the standard deviation of the activations over a batch of inputs. It is designed to standardize the input to a layer so that the gradients during backpropagation don't explode or vanish. So normalization between dis_conv1 and dis_lrelu1 can remove important information about the original input distribution, which can hurt model performance if done incorrectly. It could result in a loss of important information about these low-level features, causing the model to overfit to the training data, leading to poor generalization performance on new data.

- In the case of a GAN, adding batch normalization between the first convolutional layer and the first LeakyReLU activation function in the Discriminator can lead to over-regularization,

which can negatively impact the Discriminator's ability to distinguish between real and fake samples. This can cause the Generator to receive insufficient feedback during training, which can result in poor-quality generated samples. In the Discriminator, the first convolutional layer is responsible for capturing low-level features of the input image, such as edges and corners. By omitting the batch normalization layer between the first convolutional layer and the first LeakyReLU activation function, the Discriminator is allowed to capture the low-level features of the input image more accurately, which can help it to better distinguish between real and fake samples. This approach may not be theoretically optimal, but it can work well in practice, as shown in many successful GAN implementations.

- Furthermore, batch normalization can cause the training process to become unstable, leading to issues such as mode collapse and oscillation. This is because batch normalization can introduce additional noise into the training process, which can make it more difficult for the model to converge.

- Batch normalization is used to normalize the activations of a layer, making it easier to train the model by reducing the internal covariate shift problem. However, it is important to use batch normalization judiciously to avoid over-normalization and loss of important information in the model.

Therefore, it is generally recommended to avoid adding batch normalization between the first convolutional layer and the first LeakyReLU activation function in the Discriminator of a GAN, to ensure that the model can learn the correct distribution of the data and generate high-quality samples.

Takeaway from this problem: **always excercise extreme caution when using batch normalization in your network!**

For further info (optional): you can read this paper to find out more about why Batch Normalization might be bad for your GANs: On the Effects of Batch and Weight Normalization in Generative Adversarial Networks

---

Problem 2-3: What about other normalization methods for GAN? (4 pts)

Spectral norm is a way of stabilizing the GAN training of discriminator. Please add the embedded spectral norm function in Pytorch to the Discriminator class below in order to test its effects. (see link: https://pytorch.org/docs/stable/generated/torch.nn.utils.spectral_norm.html)

In [8]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
```

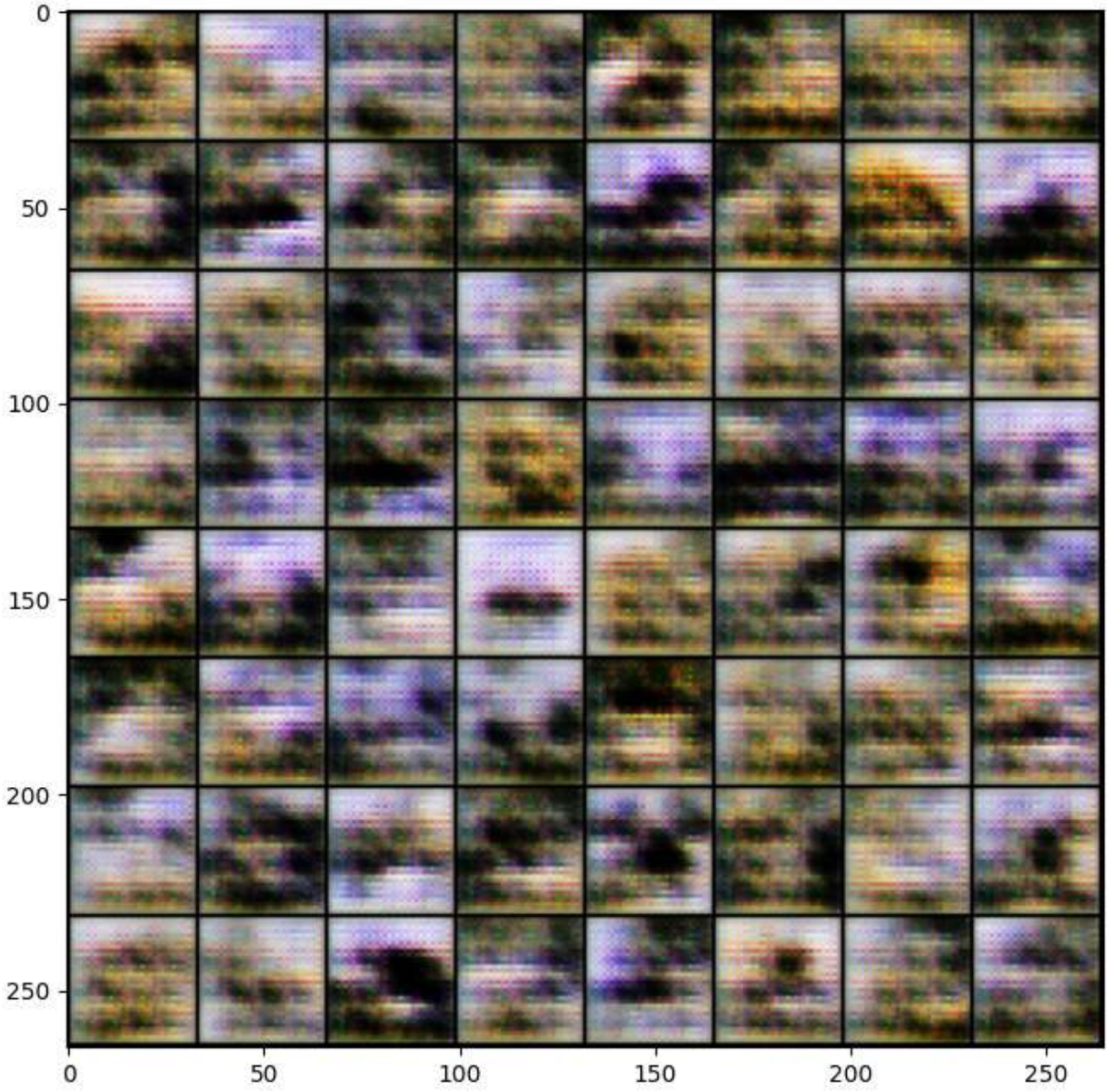
```
        ##########################################################################
        # Prob 2-3:
        # adding spectral norm to the discriminator
        ##########################################################################
        self.downsample = nn.Sequential(
            nn.utils.spectral_norm(nn.Conv2d(in_channels=3, out_channels=32, kernel_si
            nn.LeakyReLU(),
            nn.utils.spectral_norm(nn.Conv2d(in_channels=32, out_channels=64, kernel_s
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.utils.spectral_norm(nn.Conv2d(in_channels=64, out_channels=128, kernel_
            nn.BatchNorm2d(128),
            nn.LeakyReLU(),
        )
        ##########################################################################
        # END OF YOUR CODE
        ##########################################################################
        self.fc = nn.Linear(4 * 4 * 128, 1)
    def forward(self, input):
        downsampled_image = self.downsample(input)
        reshaped_for_fc = downsampled_image.reshape((-1, 4 * 4 * 128))
        classification_probs = self.fc(reshaped_for_fc)
        return classification_probs
```

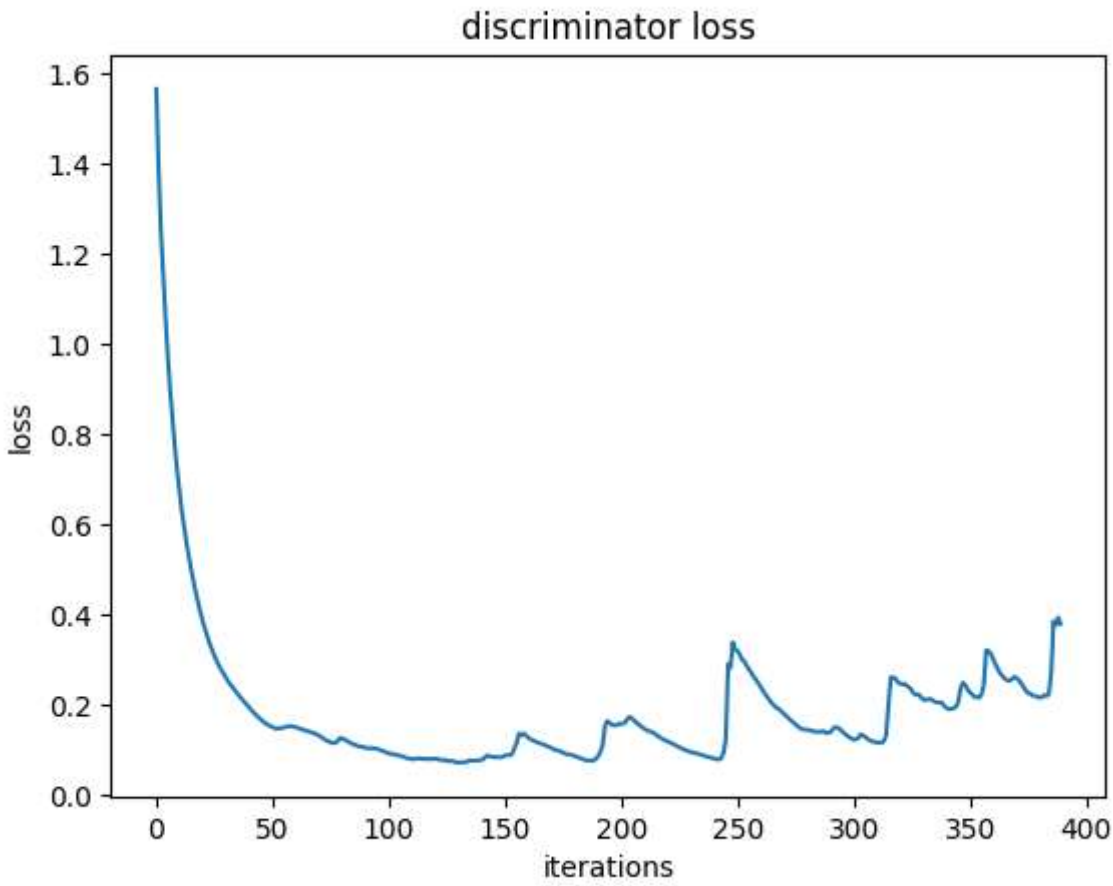
After adding the spectral norm to the discriminator, redo the training block below to see the effects.

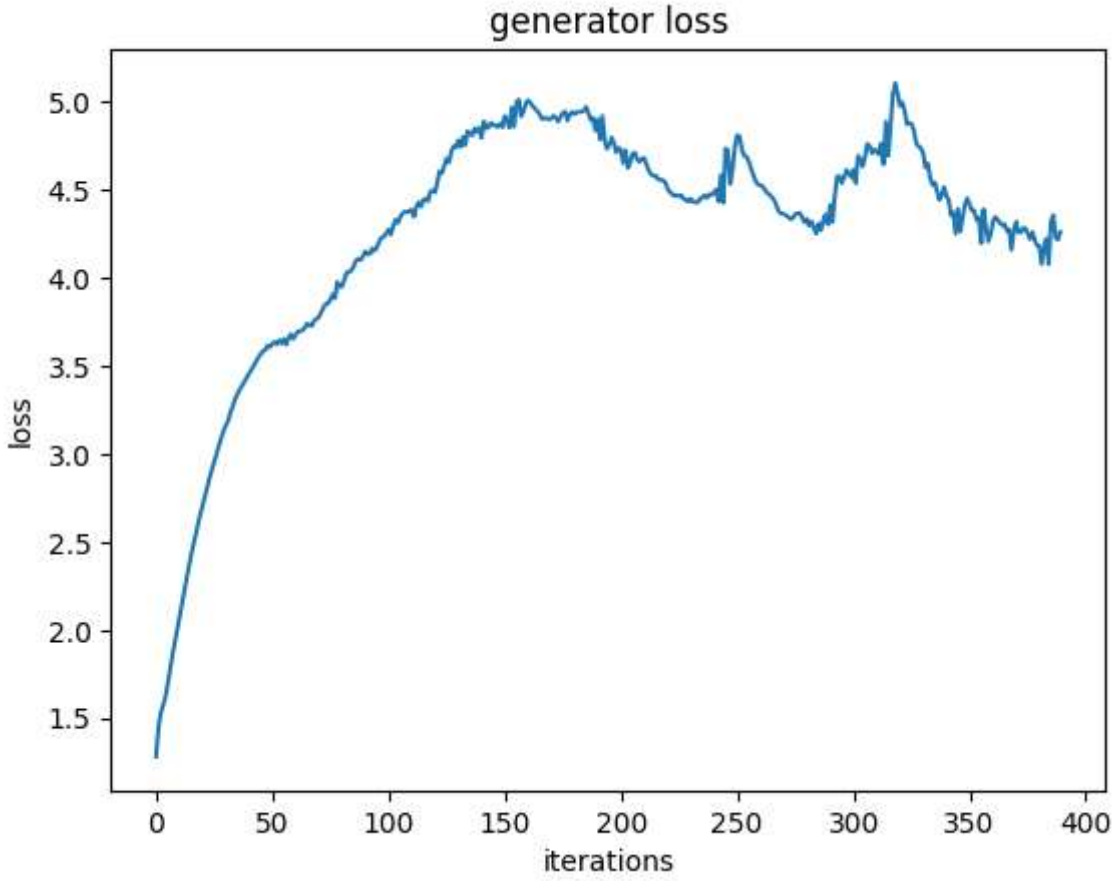
In [9]:
```
set_seed(42)

dcgan = DCGAN()
dcgan.train(train_samples)
torch.save(dcgan.state_dict(), "dcgan.pt")
```

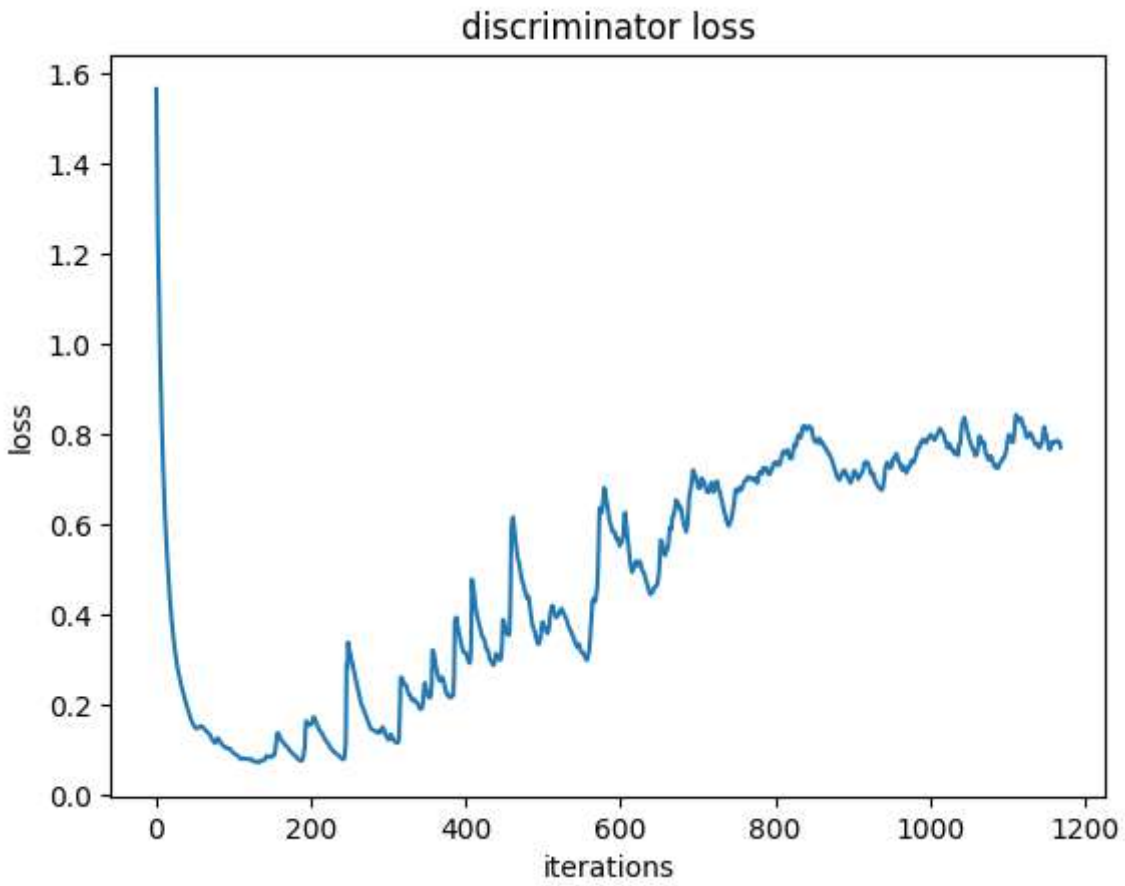
```
Start training ...
Iteration 100/9750: dis loss = 0.0603, gen loss = 4.6211
Iteration 200/9750: dis loss = 0.2156, gen loss = 5.2160
Iteration 300/9750: dis loss = 0.0561, gen loss = 4.1756
```

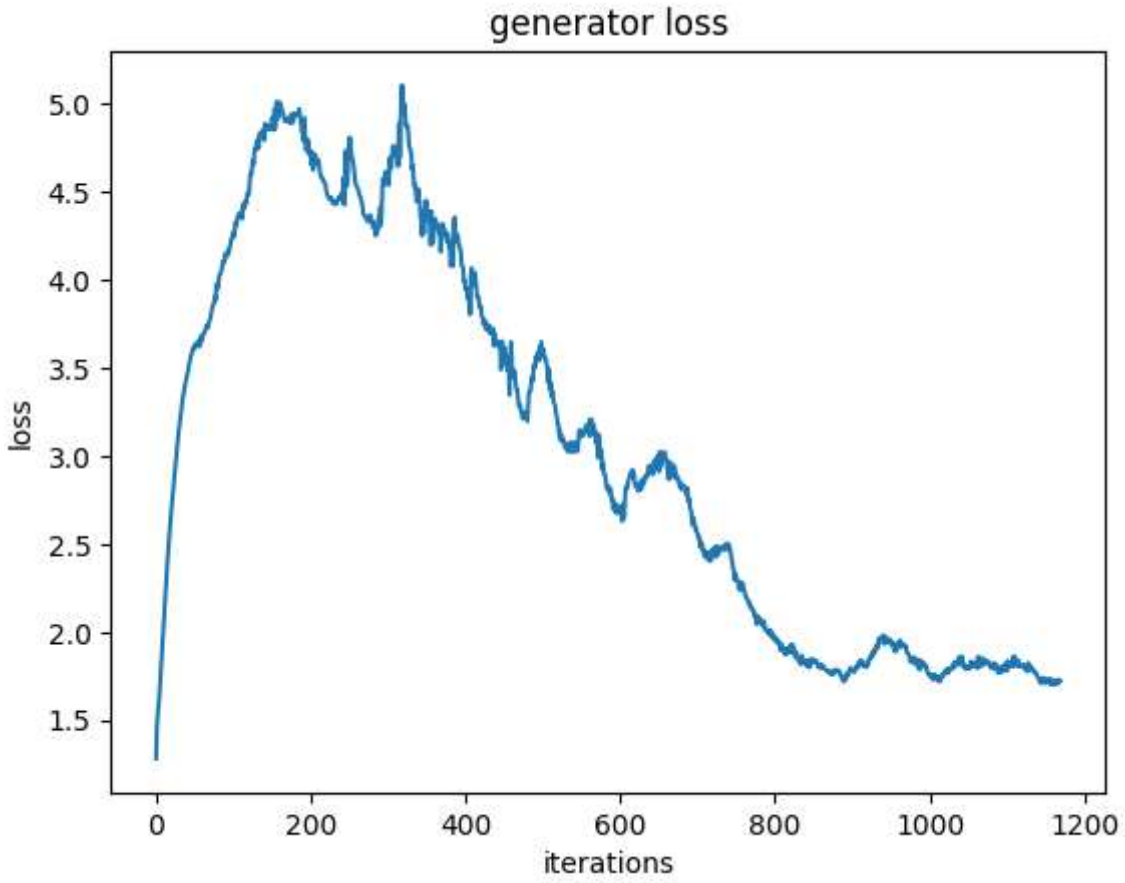

0
50
100
150
200
250
0
50
100
150
200
250


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
50
100
150
200
250
300
350
400

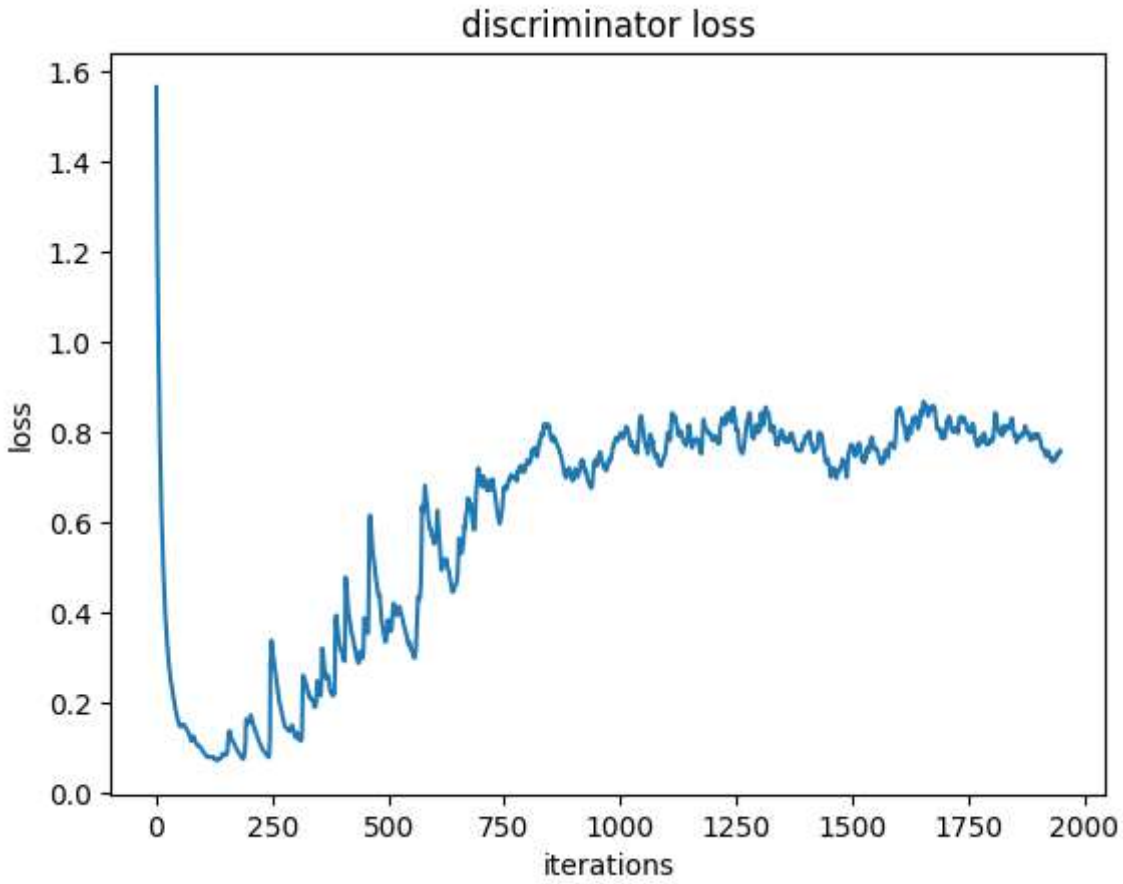

generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
50
100
150
200
250
300
350
400

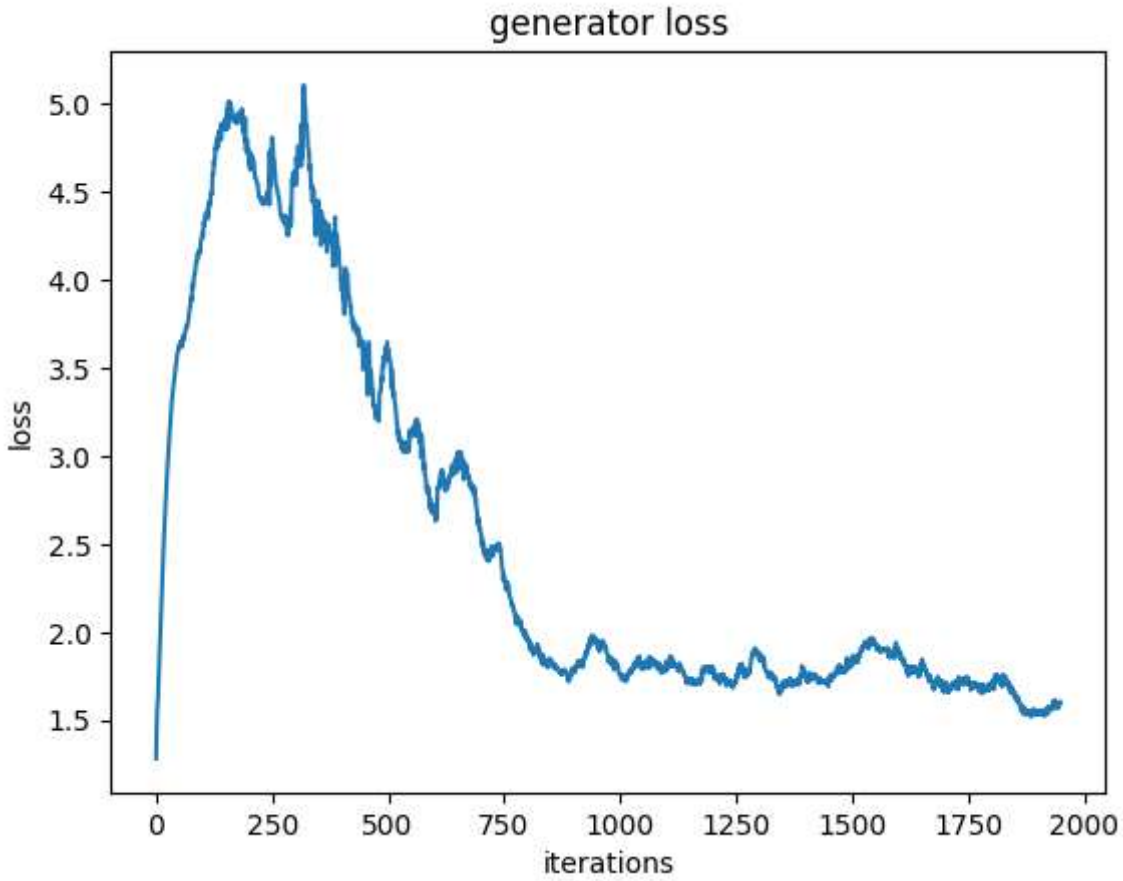
```
Iteration 400/9750: dis loss = 0.2989, gen loss = 4.0949
Iteration 500/9750: dis loss = 0.6924, gen loss = 2.4864
Iteration 600/9750: dis loss = 0.3739, gen loss = 2.5307
Iteration 700/9750: dis loss = 0.5333, gen loss = 2.1007
Iteration 800/9750: dis loss = 0.7506, gen loss = 2.2634
Iteration 900/9750: dis loss = 0.7875, gen loss = 1.4889
Iteration 1000/9750: dis loss = 0.8589, gen loss = 1.7490
Iteration 1100/9750: dis loss = 0.9146, gen loss = 2.5723
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
200
400
600
800
1000
1200

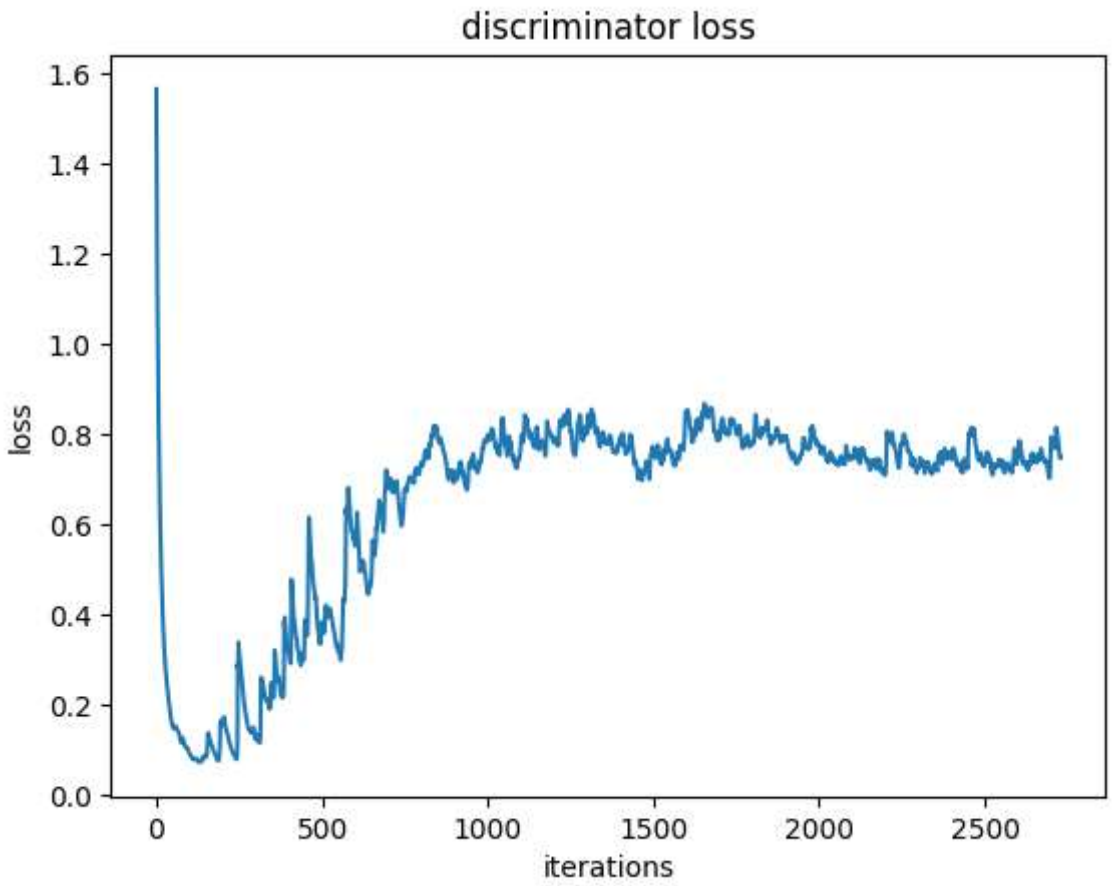

generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
200
400
600
800
1000
1200

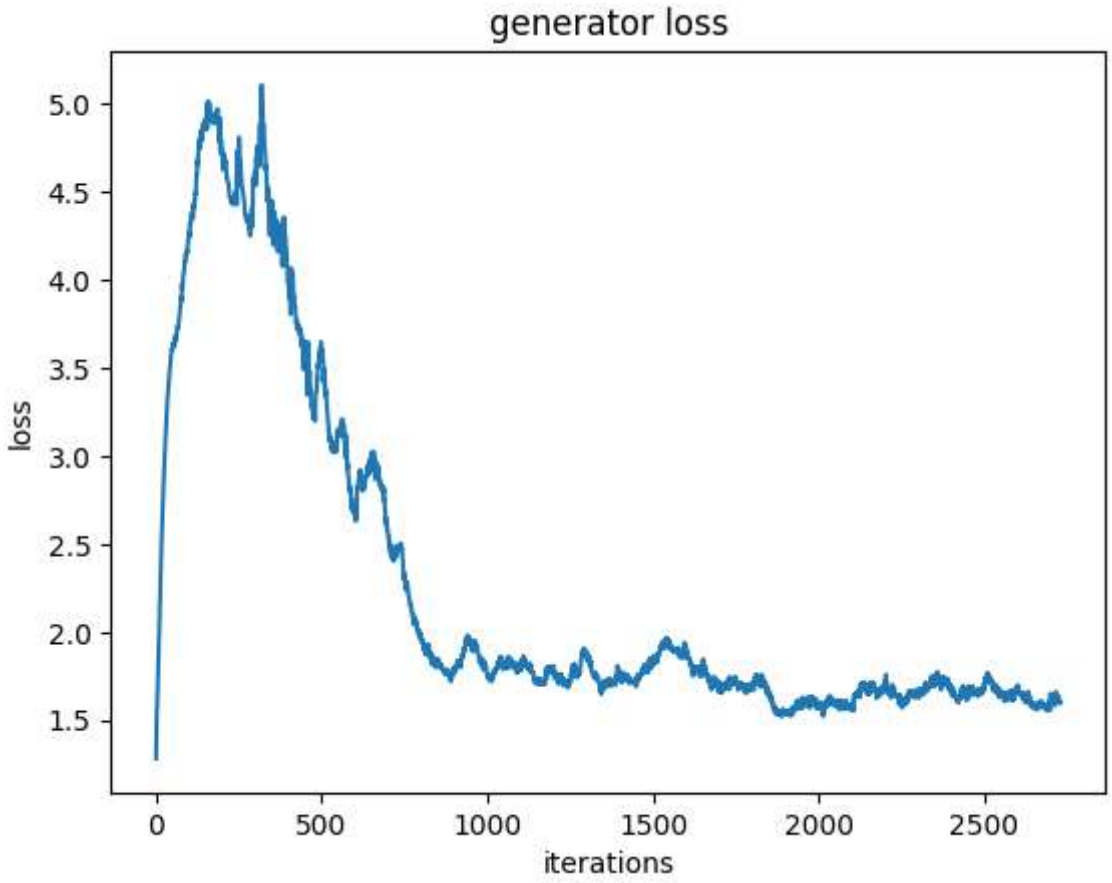
```
Iteration 1200/9750: dis loss = 0.8280, gen loss = 2.2547
Iteration 1300/9750: dis loss = 0.8006, gen loss = 1.6334
Iteration 1400/9750: dis loss = 0.8247, gen loss = 1.6119
Iteration 1500/9750: dis loss = 0.6841, gen loss = 1.6429
Iteration 1600/9750: dis loss = 0.7496, gen loss = 1.9720
Iteration 1700/9750: dis loss = 0.7146, gen loss = 1.0213
Iteration 1800/9750: dis loss = 0.8284, gen loss = 1.8854
Iteration 1900/9750: dis loss = 0.7669, gen loss = 1.8147
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
250
500
750
1000
1250
1500
1750
2000

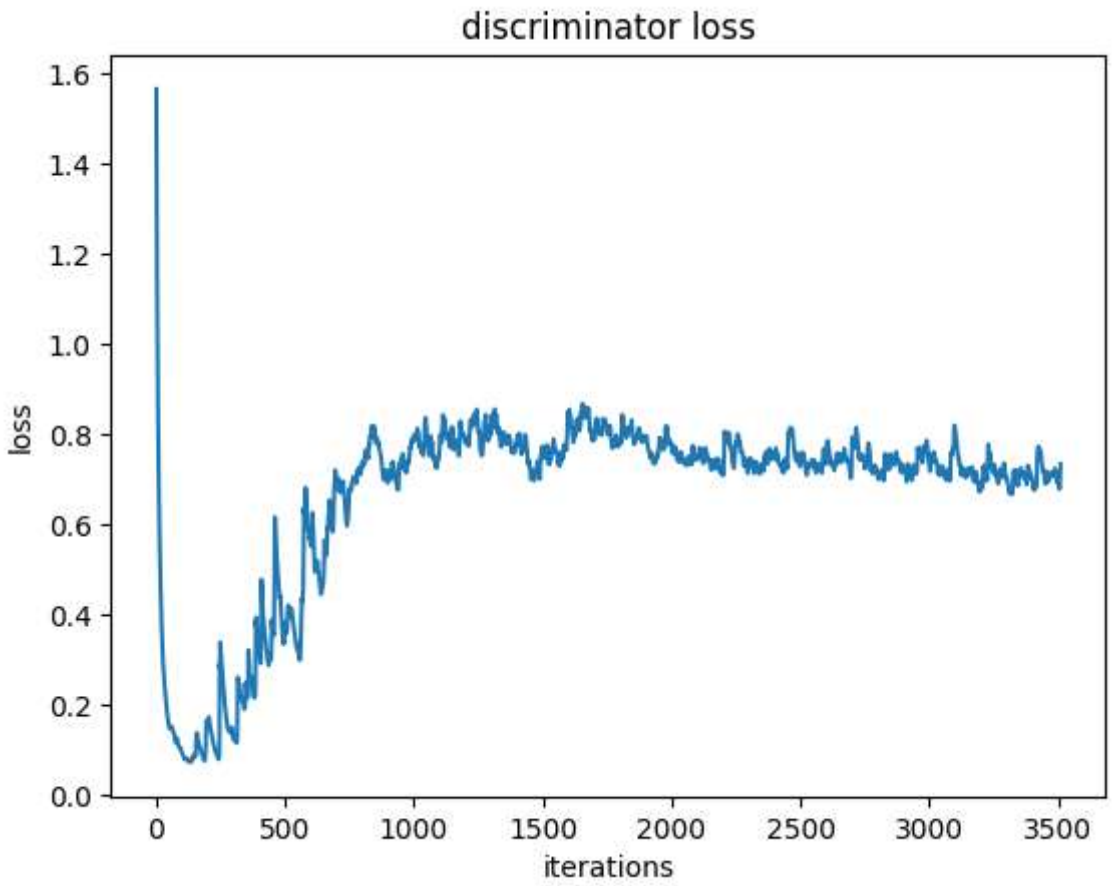

generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
250
500
750
1000
1250
1500
1750
2000

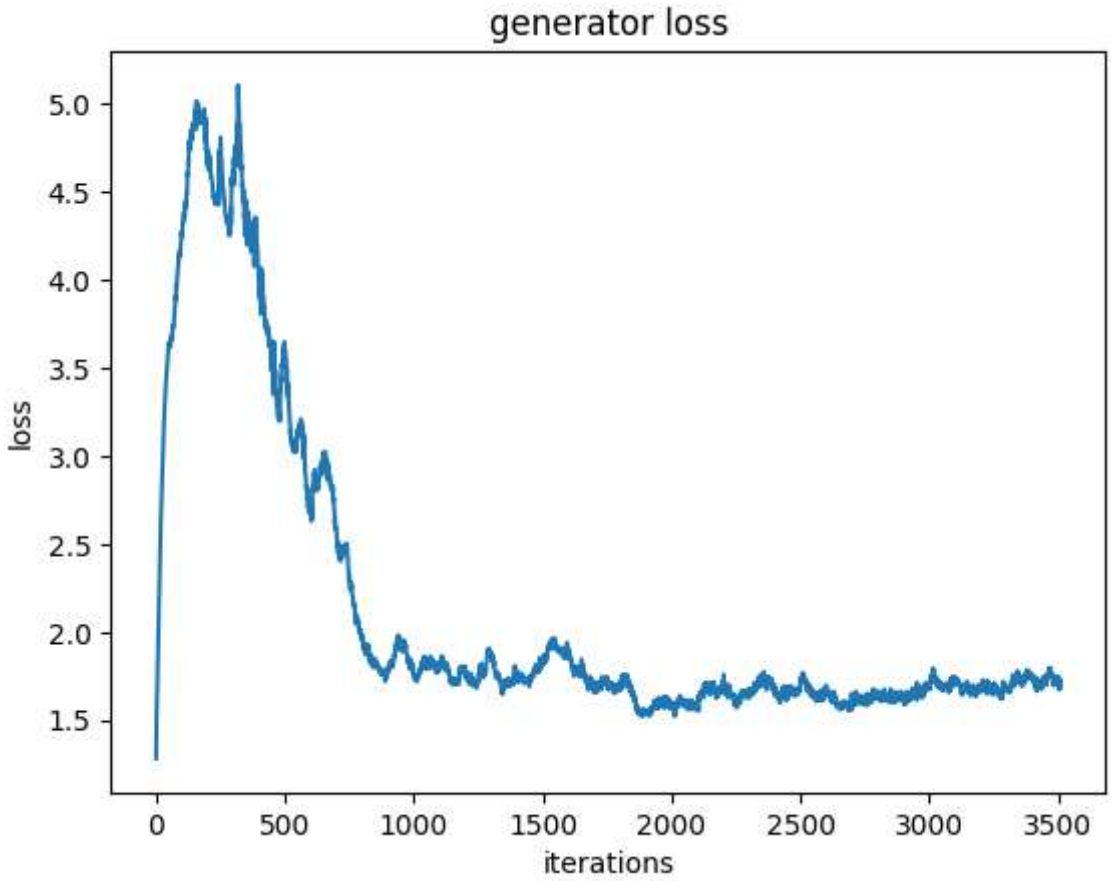
```
Iteration 2000/9750: dis loss = 0.7229, gen loss = 1.8599
Iteration 2100/9750: dis loss = 0.8604, gen loss = 2.0591
Iteration 2200/9750: dis loss = 0.7026, gen loss = 1.8088
Iteration 2300/9750: dis loss = 0.6944, gen loss = 1.5798
Iteration 2400/9750: dis loss = 0.5947, gen loss = 1.9899
Iteration 2500/9750: dis loss = 0.8181, gen loss = 2.5769
Iteration 2600/9750: dis loss = 0.8371, gen loss = 1.2123
Iteration 2700/9750: dis loss = 1.3747, gen loss = 3.4524
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
500
1000
1500
2000
2500

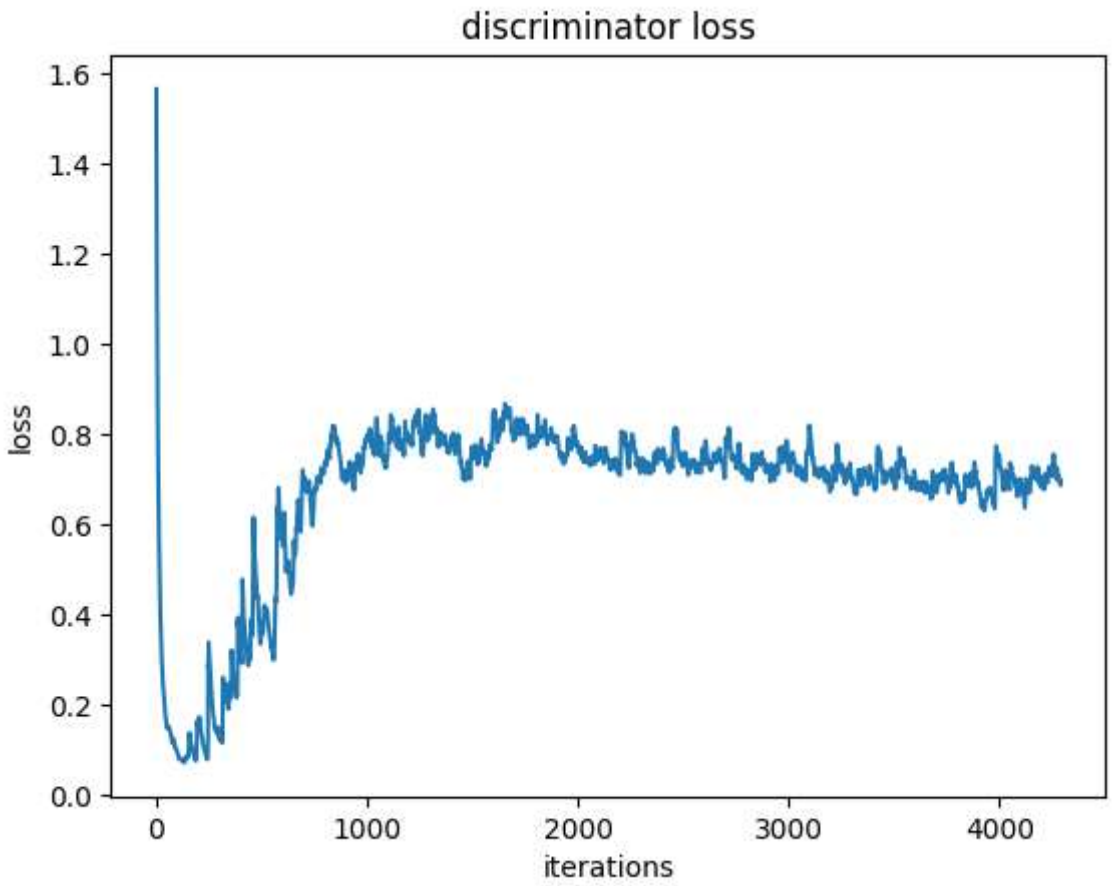

generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
500
1000
1500
2000
2500

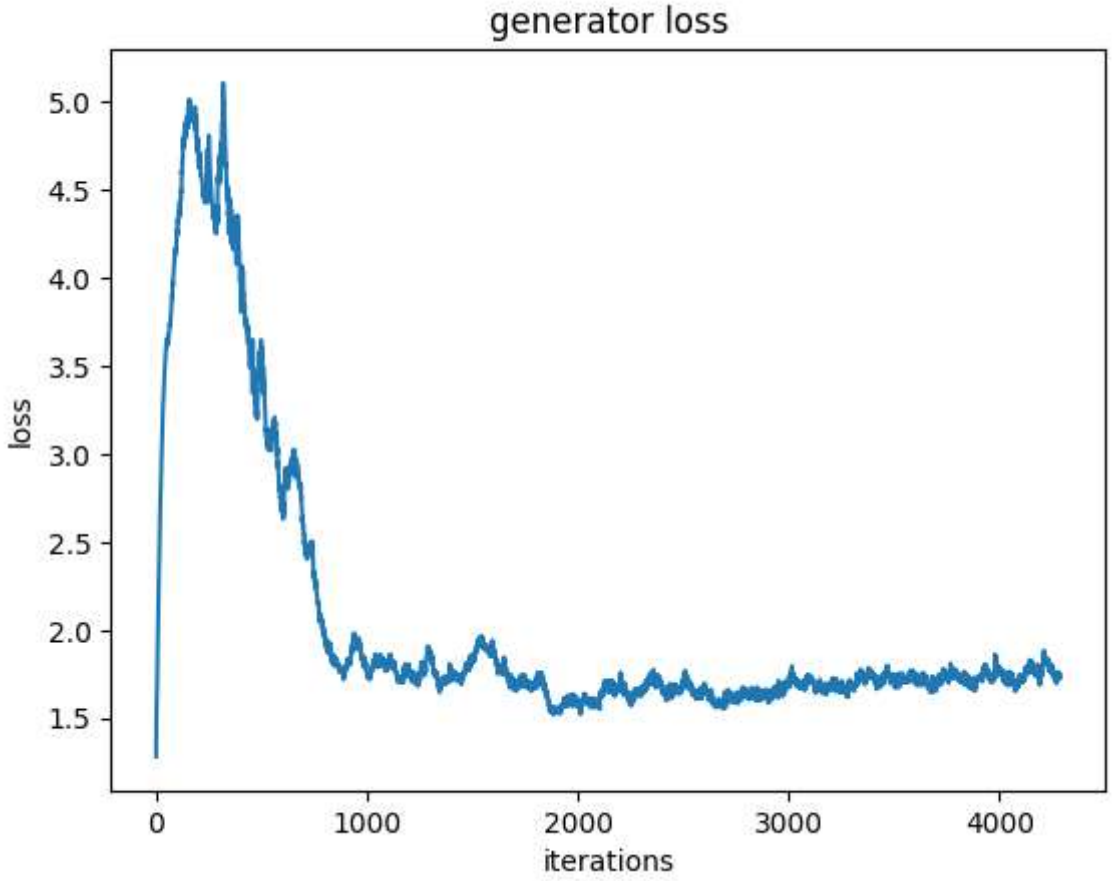
```
Iteration 2800/9750: dis loss = 0.6554, gen loss = 1.4294
Iteration 2900/9750: dis loss = 0.6266, gen loss = 1.8264
Iteration 3000/9750: dis loss = 0.6140, gen loss = 1.7790
Iteration 3100/9750: dis loss = 0.7364, gen loss = 1.9271
Iteration 3200/9750: dis loss = 0.8299, gen loss = 2.1296
Iteration 3300/9750: dis loss = 0.7239, gen loss = 1.8173
Iteration 3400/9750: dis loss = 0.6981, gen loss = 1.5748
Iteration 3500/9750: dis loss = 0.5749, gen loss = 1.5557
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
500
1000
1500
2000
2500
3000
3500

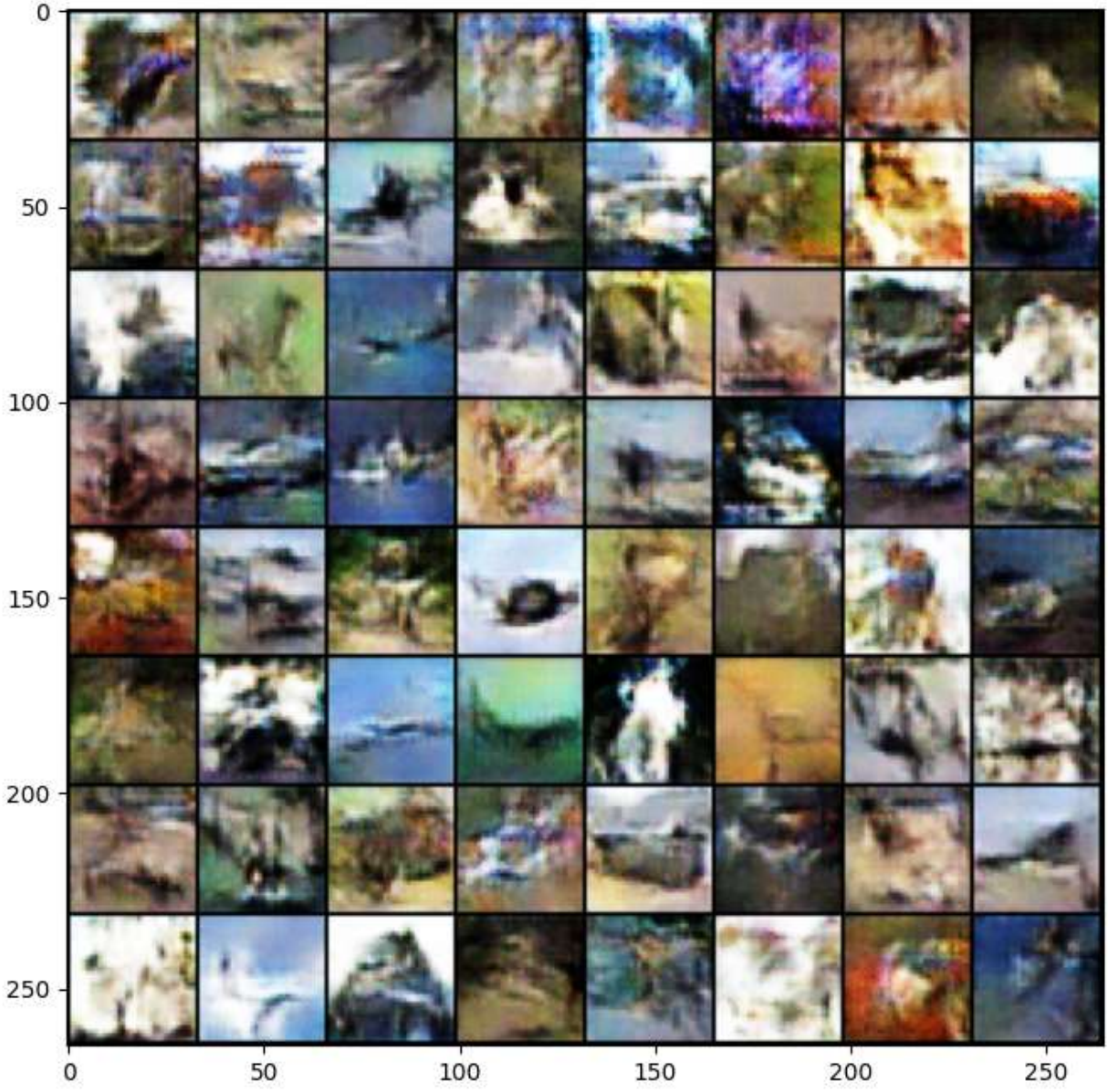

generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
500
1000
1500
2000
2500
3000
3500

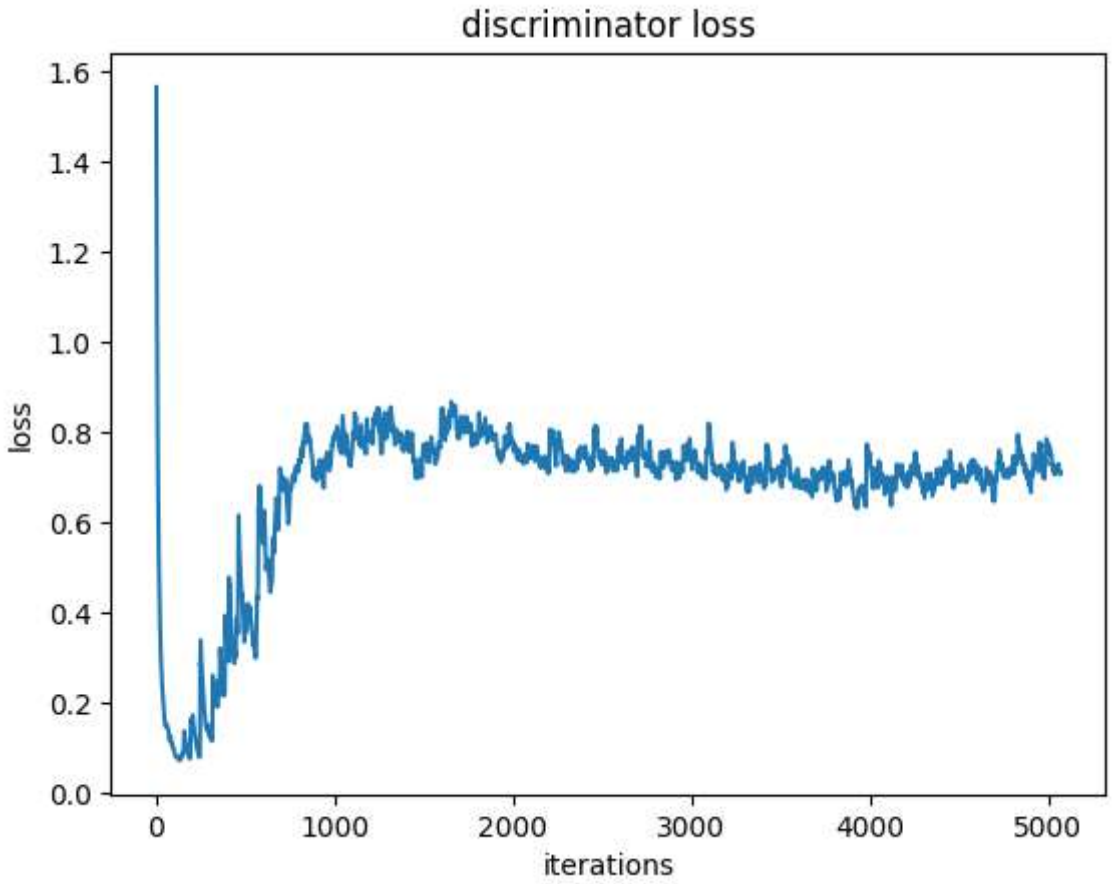
```
Iteration 3600/9750: dis loss = 0.6949, gen loss = 1.5382
Iteration 3700/9750: dis loss = 0.6582, gen loss = 2.2350
Iteration 3800/9750: dis loss = 0.7313, gen loss = 1.2533
Iteration 3900/9750: dis loss = 0.6379, gen loss = 1.3522
Iteration 4000/9750: dis loss = 0.6207, gen loss = 1.7190
Iteration 4100/9750: dis loss = 0.5392, gen loss = 1.7330
Iteration 4200/9750: dis loss = 0.8965, gen loss = 0.9065
```

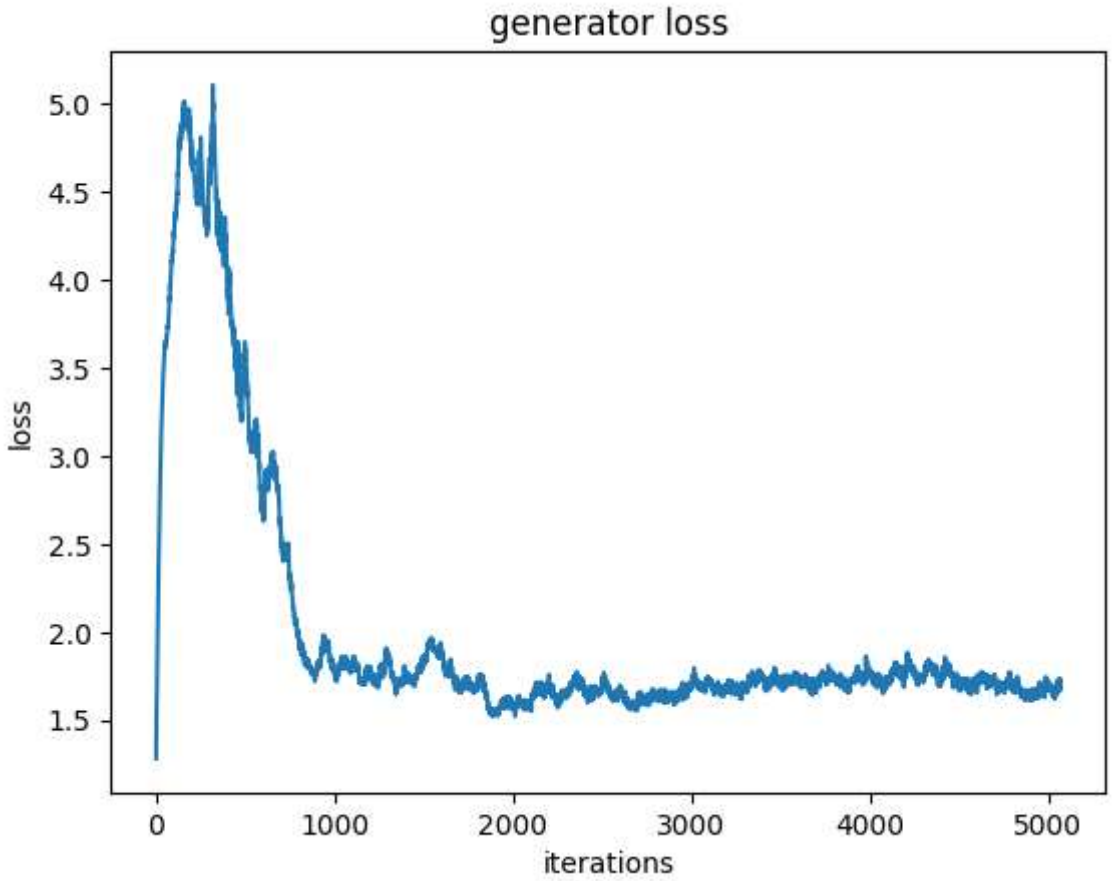

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000


generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
1000
2000
3000
4000

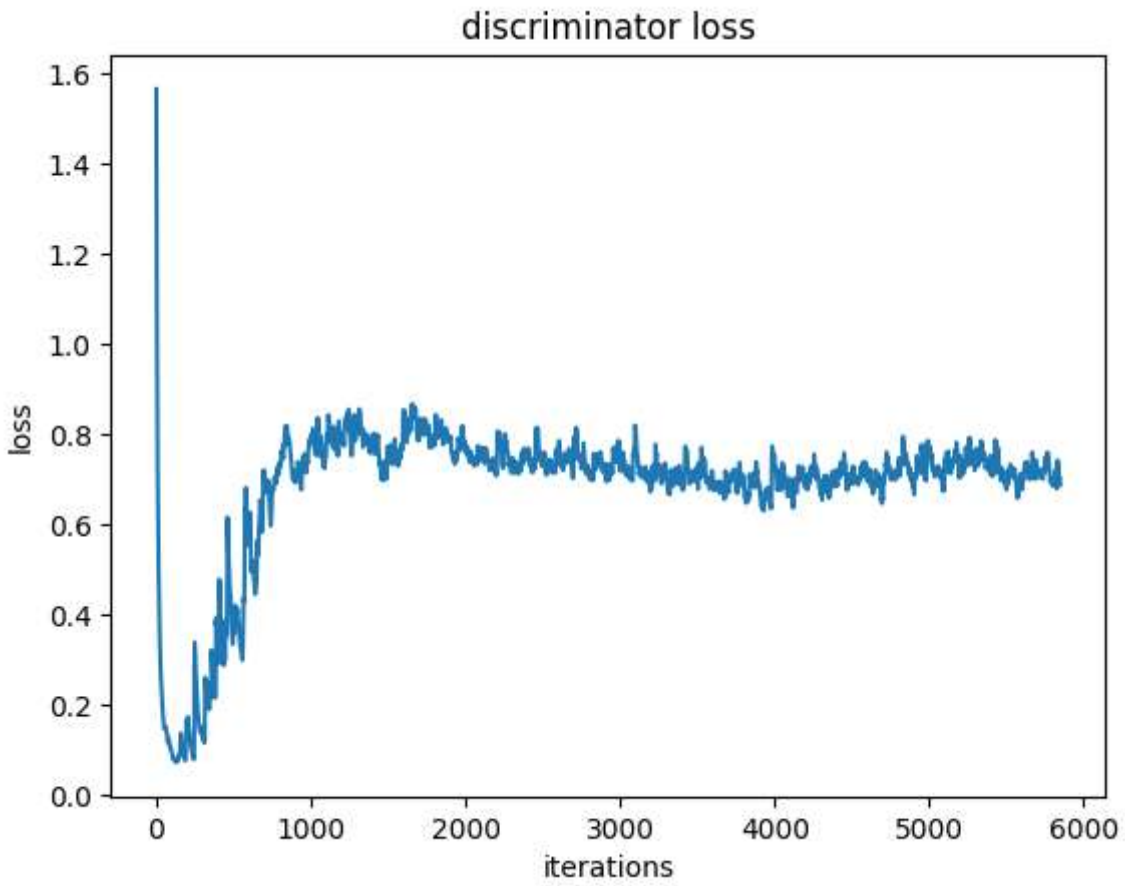
```
Iteration 4300/9750: dis loss = 0.6378, gen loss = 1.3550
Iteration 4400/9750: dis loss = 0.6667, gen loss = 1.0795
Iteration 4500/9750: dis loss = 0.6748, gen loss = 2.3545
Iteration 4600/9750: dis loss = 0.5311, gen loss = 1.6973
Iteration 4700/9750: dis loss = 0.7481, gen loss = 0.9717
Iteration 4800/9750: dis loss = 0.6972, gen loss = 1.2648
Iteration 4900/9750: dis loss = 0.5793, gen loss = 1.4643
Iteration 5000/9750: dis loss = 0.7255, gen loss = 1.9976
```

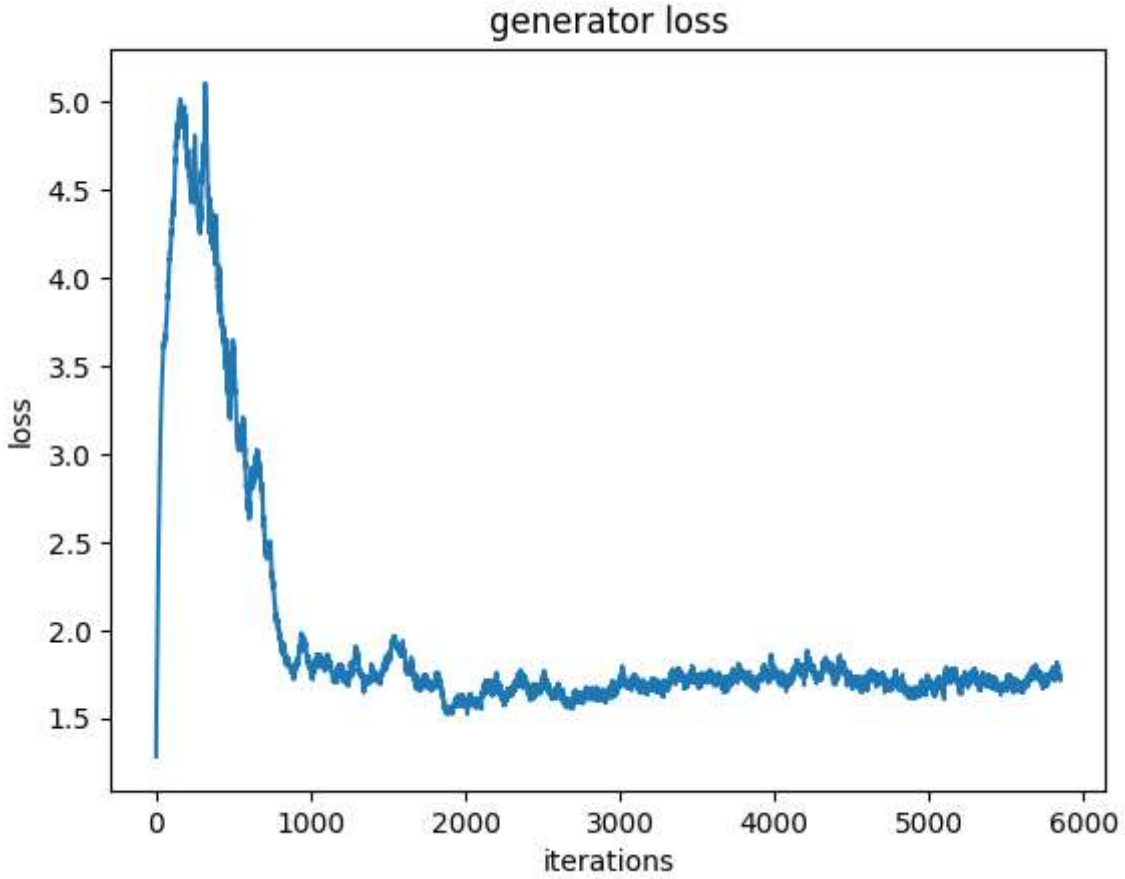



discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000

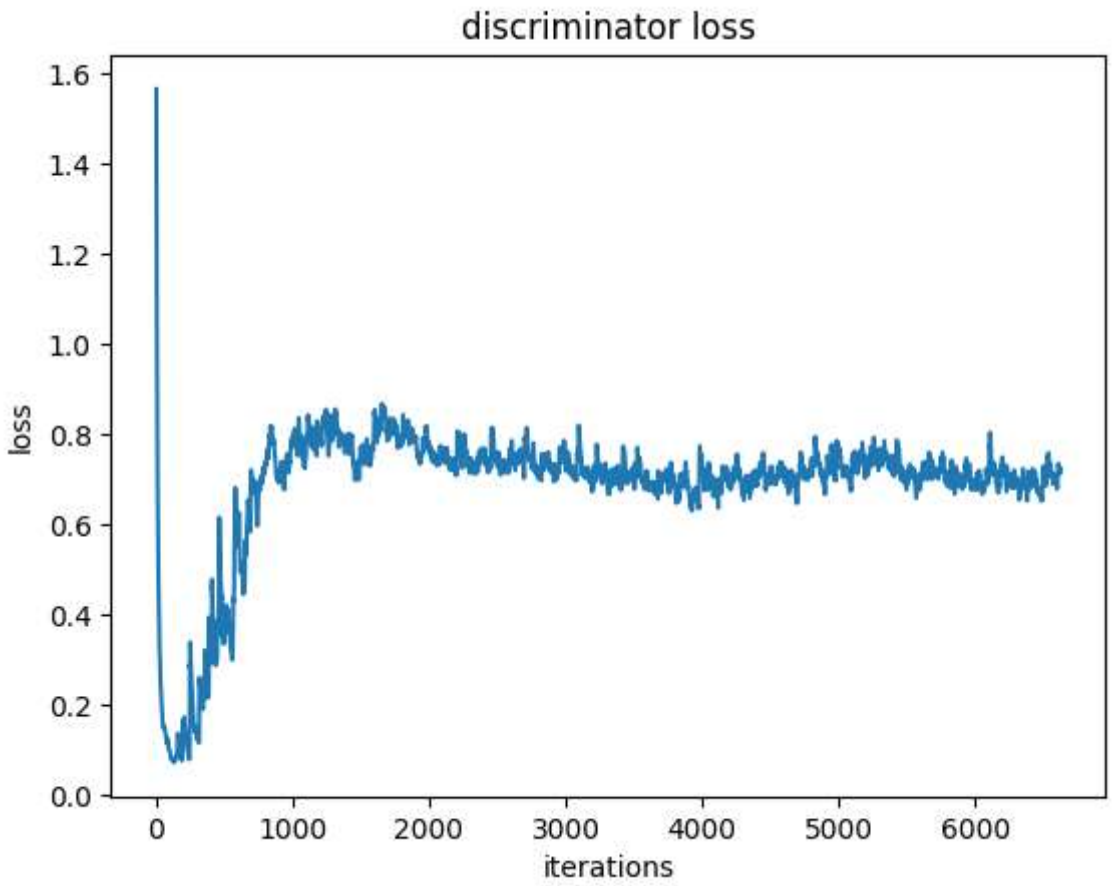

generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
1000
2000
3000
4000
5000

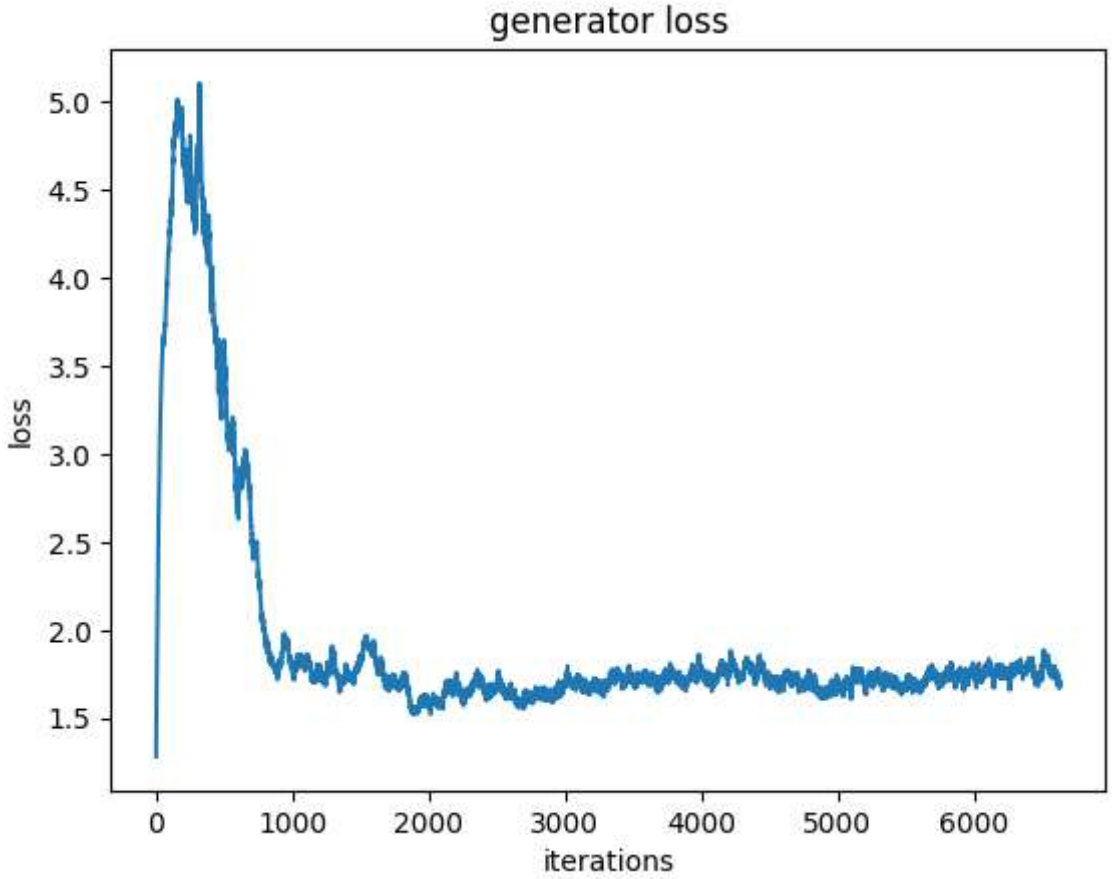
```
Iteration 5100/9750: dis loss = 1.2231, gen loss = 0.9550
Iteration 5200/9750: dis loss = 0.9997, gen loss = 2.1250
Iteration 5300/9750: dis loss = 0.7964, gen loss = 2.5447
Iteration 5400/9750: dis loss = 0.8131, gen loss = 1.4824
Iteration 5500/9750: dis loss = 0.7564, gen loss = 2.1571
Iteration 5600/9750: dis loss = 0.7382, gen loss = 1.1180
Iteration 5700/9750: dis loss = 0.6451, gen loss = 1.5930
Iteration 5800/9750: dis loss = 0.5862, gen loss = 1.2355
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000
6000

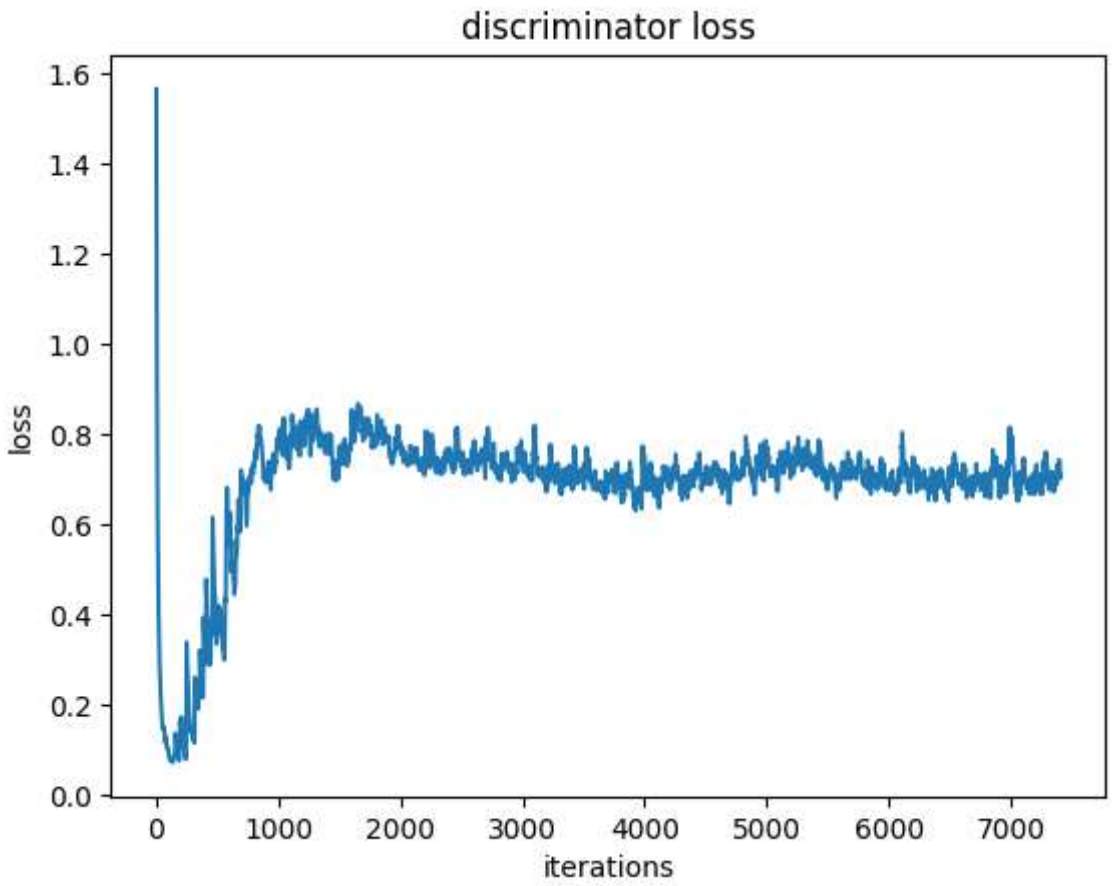

generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
1000
2000
3000
4000
5000
6000

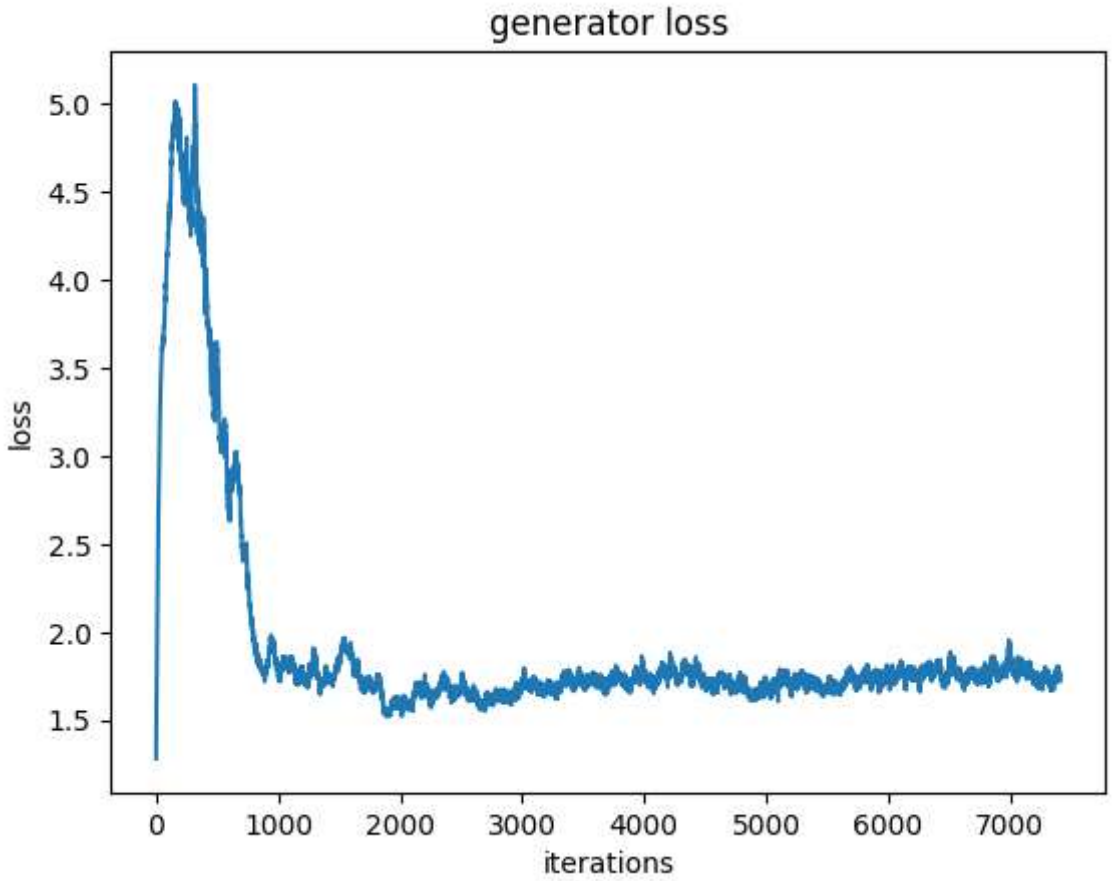
```
Iteration 5900/9750: dis loss = 0.6688, gen loss = 1.6024
Iteration 6000/9750: dis loss = 0.5988, gen loss = 2.0438
Iteration 6100/9750: dis loss = 0.6834, gen loss = 1.5149
Iteration 6200/9750: dis loss = 0.7582, gen loss = 1.4124
Iteration 6300/9750: dis loss = 0.5512, gen loss = 2.4683
Iteration 6400/9750: dis loss = 0.5380, gen loss = 1.9463
Iteration 6500/9750: dis loss = 0.6164, gen loss = 3.0055
Iteration 6600/9750: dis loss = 0.7524, gen loss = 1.3332
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000
6000

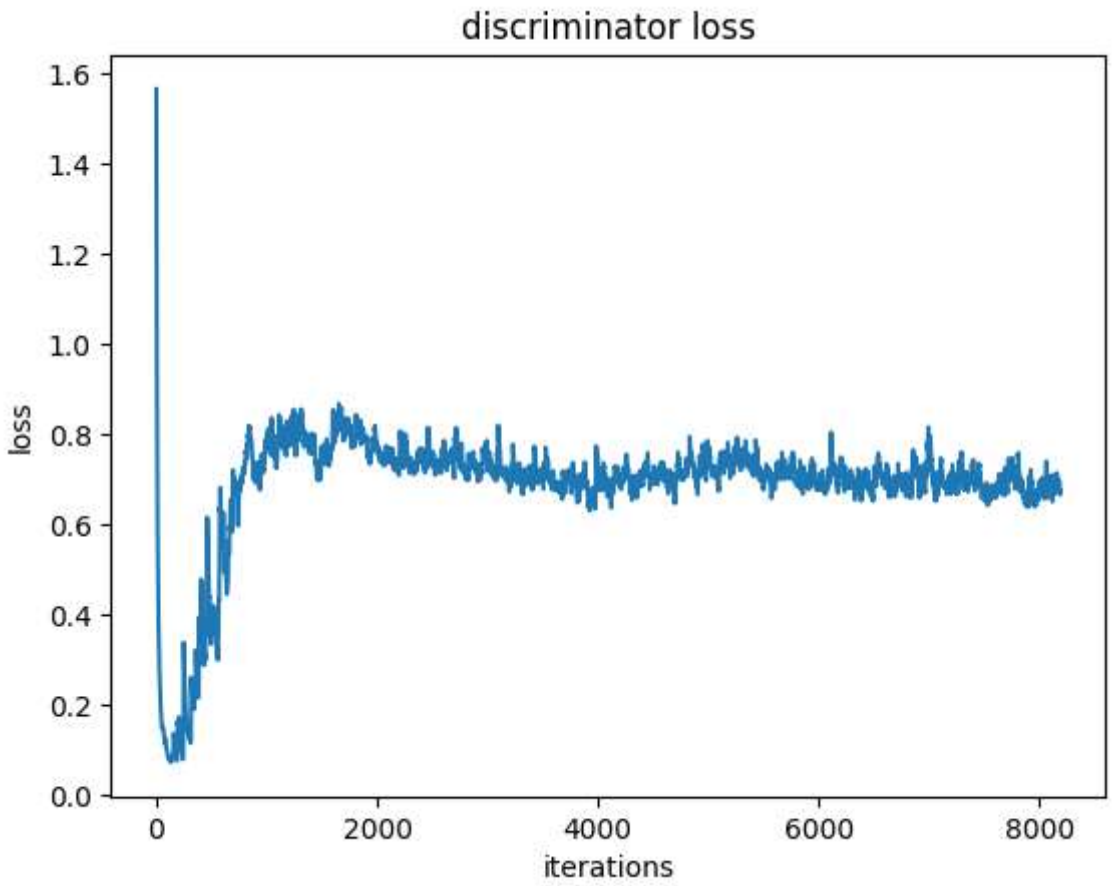

generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
1000
2000
3000
4000
5000
6000

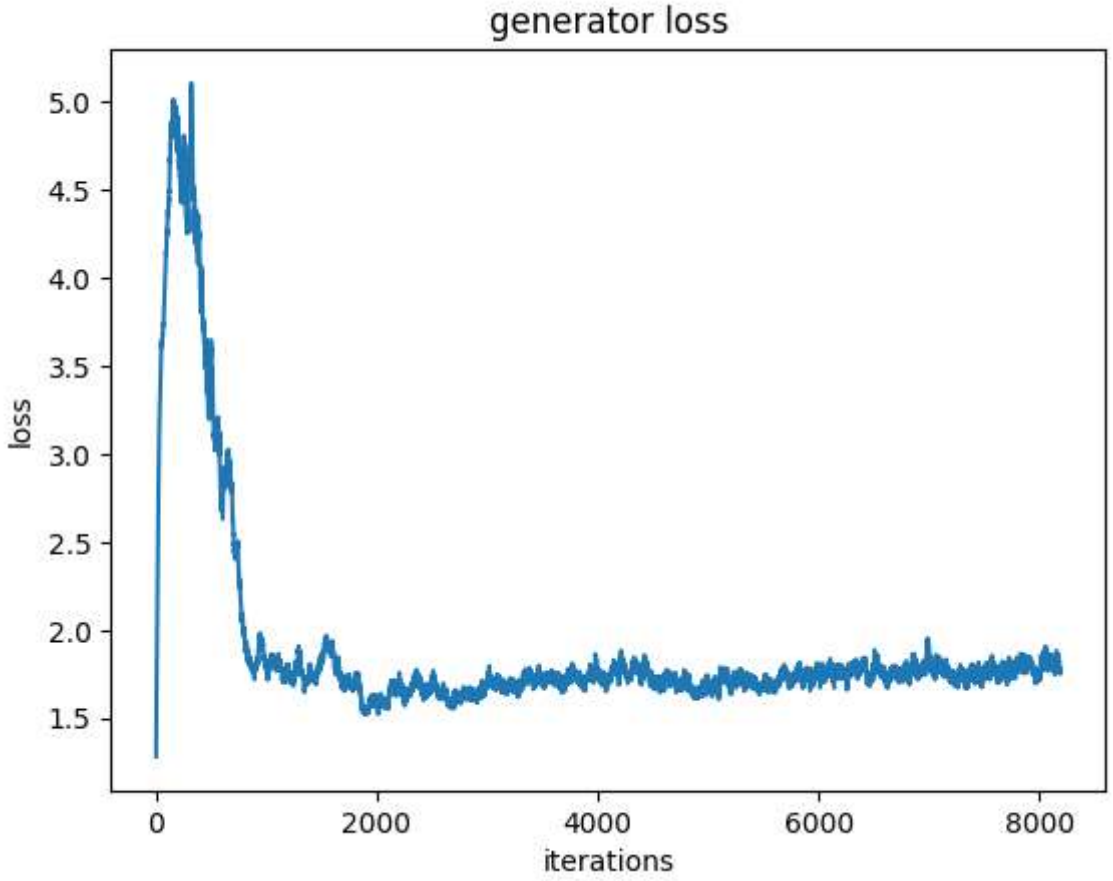
```
Iteration 6700/9750: dis loss = 0.7795, gen loss = 1.0982
Iteration 6800/9750: dis loss = 0.7228, gen loss = 1.7502
Iteration 6900/9750: dis loss = 0.5807, gen loss = 2.1134
Iteration 7000/9750: dis loss = 0.7470, gen loss = 1.4230
Iteration 7100/9750: dis loss = 0.6749, gen loss = 2.4124
Iteration 7200/9750: dis loss = 0.5519, gen loss = 1.8472
Iteration 7300/9750: dis loss = 0.7241, gen loss = 2.2588
Iteration 7400/9750: dis loss = 0.6555, gen loss = 2.4363
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000
6000
7000

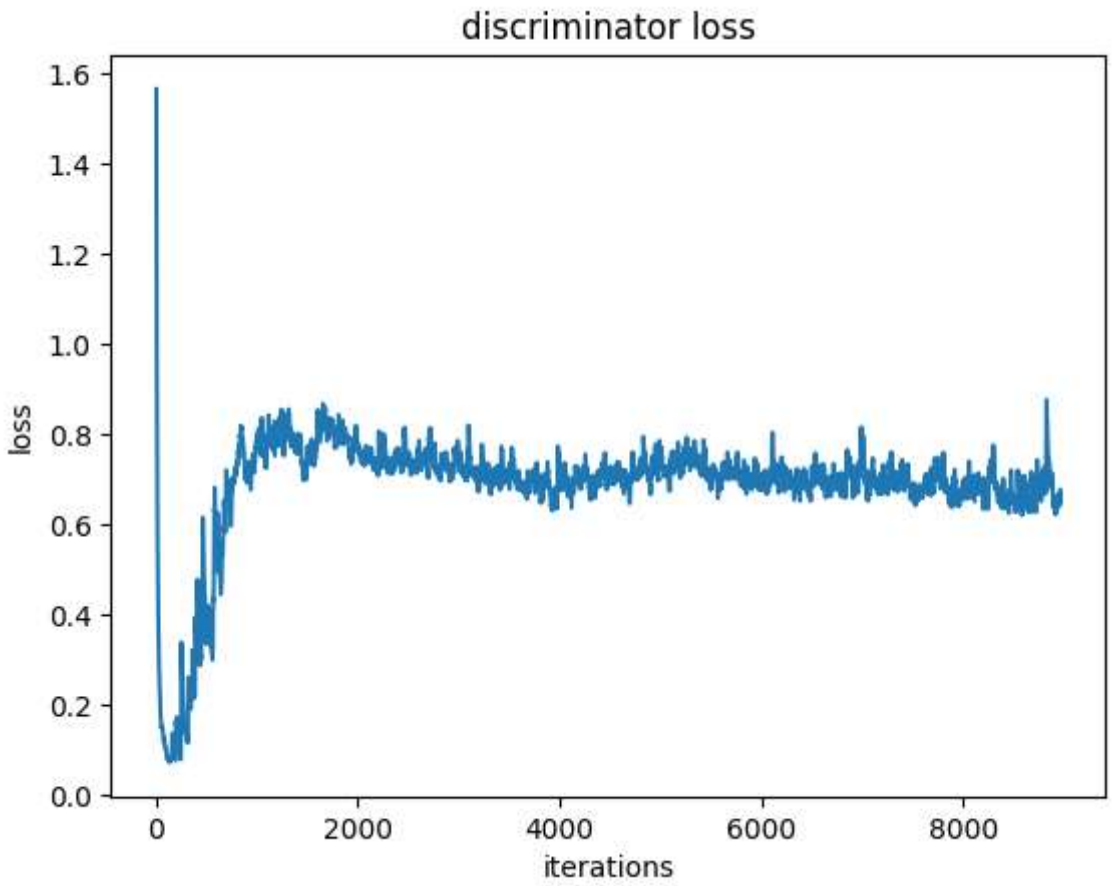

generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
1000
2000
3000
4000
5000
6000
7000

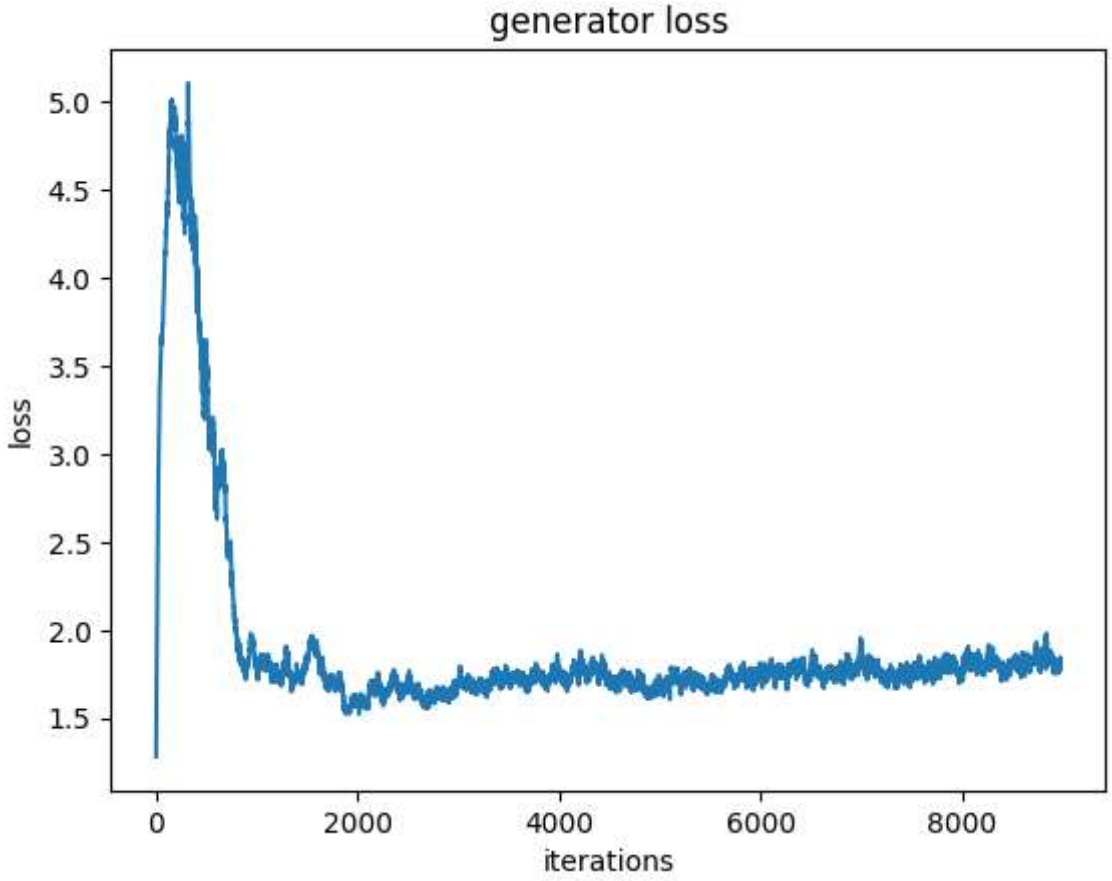
```
Iteration 7500/9750: dis loss = 0.7052, gen loss = 2.1340
Iteration 7600/9750: dis loss = 0.7260, gen loss = 1.4441
Iteration 7700/9750: dis loss = 0.5568, gen loss = 1.9328
Iteration 7800/9750: dis loss = 0.7092, gen loss = 1.1186
Iteration 7900/9750: dis loss = 0.6707, gen loss = 1.2925
Iteration 8000/9750: dis loss = 0.5270, gen loss = 1.8693
Iteration 8100/9750: dis loss = 0.8788, gen loss = 2.2649
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
2000
4000
6000
8000

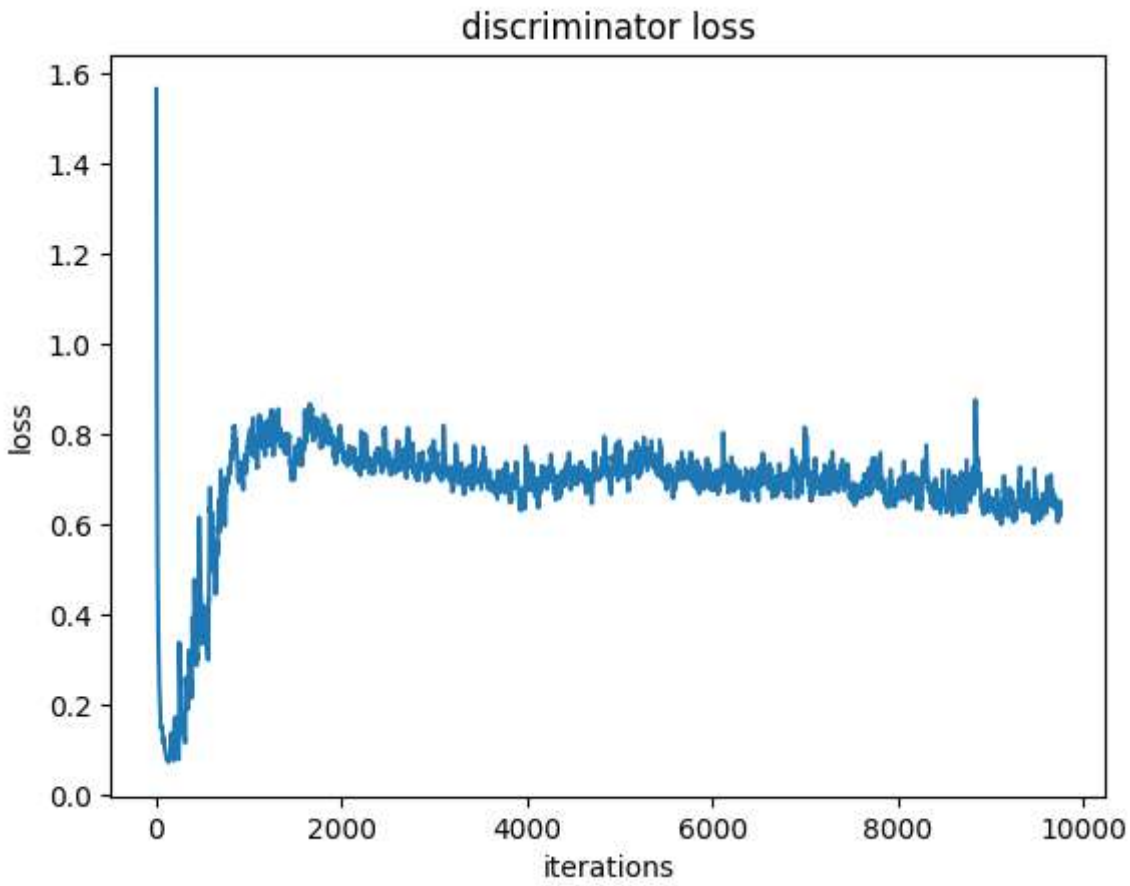

generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
2000
4000
6000
8000

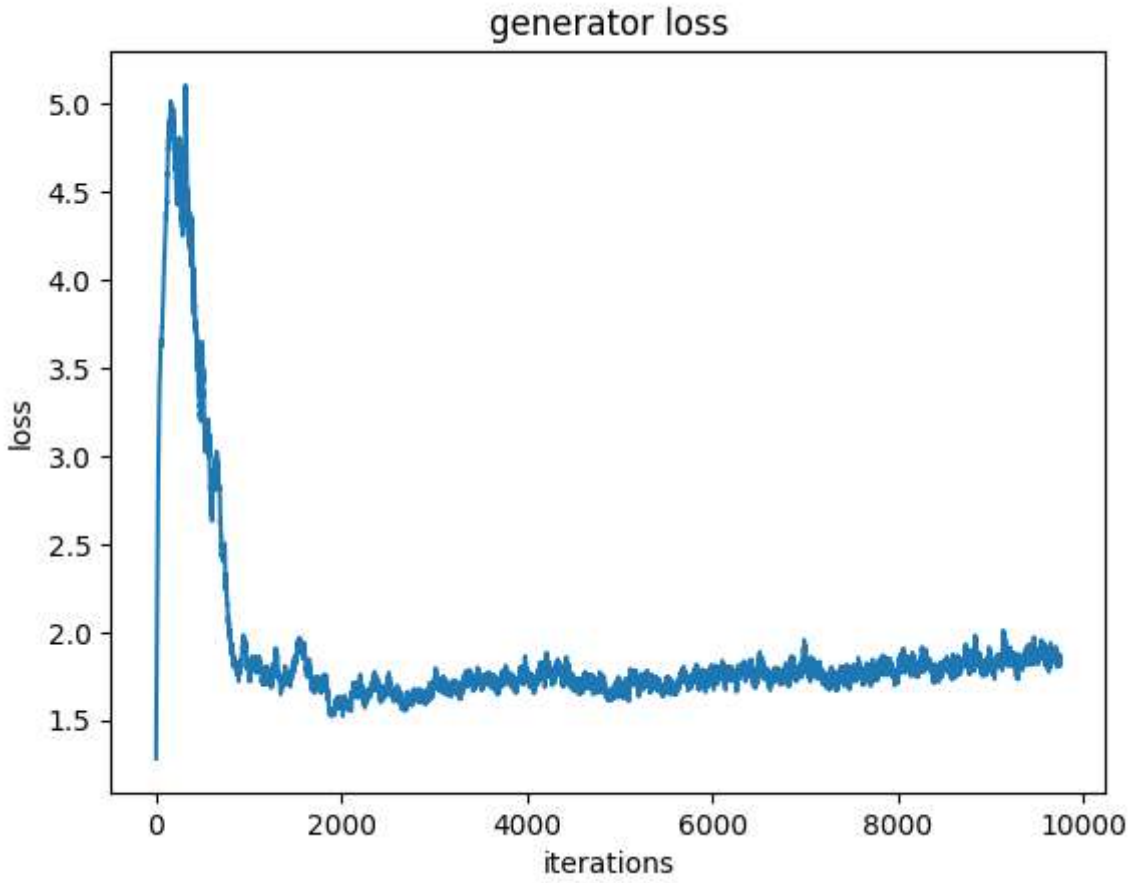
```
Iteration 8200/9750: dis loss = 0.4766, gen loss = 1.9727
Iteration 8300/9750: dis loss = 0.9242, gen loss = 1.1056
Iteration 8400/9750: dis loss = 1.1095, gen loss = 3.1770
Iteration 8500/9750: dis loss = 0.6358, gen loss = 1.3653
Iteration 8600/9750: dis loss = 0.6160, gen loss = 1.0132
Iteration 8700/9750: dis loss = 0.7675, gen loss = 1.2143
Iteration 8800/9750: dis loss = 0.9081, gen loss = 1.0101
Iteration 8900/9750: dis loss = 0.6849, gen loss = 2.0770
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
2000
4000
6000
8000


generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
2000
4000
6000
8000

```
Iteration 9000/9750: dis loss = 0.6168, gen loss = 2.0719
Iteration 9100/9750: dis loss = 0.5323, gen loss = 1.4643
Iteration 9200/9750: dis loss = 0.6545, gen loss = 1.5621
Iteration 9300/9750: dis loss = 0.8368, gen loss = 2.1523
Iteration 9400/9750: dis loss = 0.6427, gen loss = 2.0262
Iteration 9500/9750: dis loss = 0.6225, gen loss = 2.0802
Iteration 9600/9750: dis loss = 1.0068, gen loss = 3.3358
Iteration 9700/9750: dis loss = 0.5965, gen loss = 1.2874
```


discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
2000
4000
6000
8000
10000


generator loss
loss
iterations
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
2000
4000
6000
8000
10000

```
... Done!
```

Problem 2-4: Activation Maximization (12 pts)

Activation Maximization is a visualization technique to see what a particular neuron has learned, by finding the input that maximizes the activation of that neuron. Here we use methods similar to Synthesizing the preferred inputs for neurons in neural networks via deep generator networks.

In short, what we want to do is to find the samples that the discriminator considers most real, among all possible outputs of the generator, which is to say, we want to find the codes (i.e. a point in the input space of the generator) from which the generated images, if labelled as real, would minimize the classification loss of the discriminator:

$$\min_z L(D_\theta(G_\phi(z)), 1)$$

Compare this to the objective when we were training the generator:

$$\min_\phi \mathrm{E}_{z\sim q(z)}[L(D_\theta(G_\phi(z)), 1)]$$

The function to minimize is the same, with the difference being that when training the network we fix a set of input data and find the optimal model parameters, while in activation maximization we fix the model parameters and find the optimal input.

So, similar to the training, we use gradient descent to solve for the optimal input. Starting from a random code (latent vector) drawn from a standard normal distribution, we perform a fixed step of Adam optimization algorithm on the code (latent vector).

The batch normalization layers should work in evaluation mode.

We provide the code for this part, as a reference for solving the next part. You may want to go back to the code above and check the `actmax` function and figure out what it's doing:

In [10]:
```
set_seed(241)

dcgan = DCGAN()
dcgan.load_state_dict(torch.load("dcgan.pt", map_location=device))

actmax_results = dcgan.actmax(np.random.normal(size=(64, dcgan.code_size)))
fig = plt.figure(figsize = (8, 8))
ax1 = plt.subplot(111)
ax1.imshow(make_grid(actmax_results, padding=1, normalize=True).numpy().transpose((1,
plt.show()
```

The output should have less variety than those generated from random code, but look realisitic.

A similar technique can be used to reconstruct a test sample, that is, to find the code that most closely approximates the test sample. To achieve this, we only need to change the loss function from discriminator's loss to the squared L2-distance between the generated image and the target image:

$$\min_z ||G_\phi(z) - x||_2^2$$

This time, we always start from a zero vector.

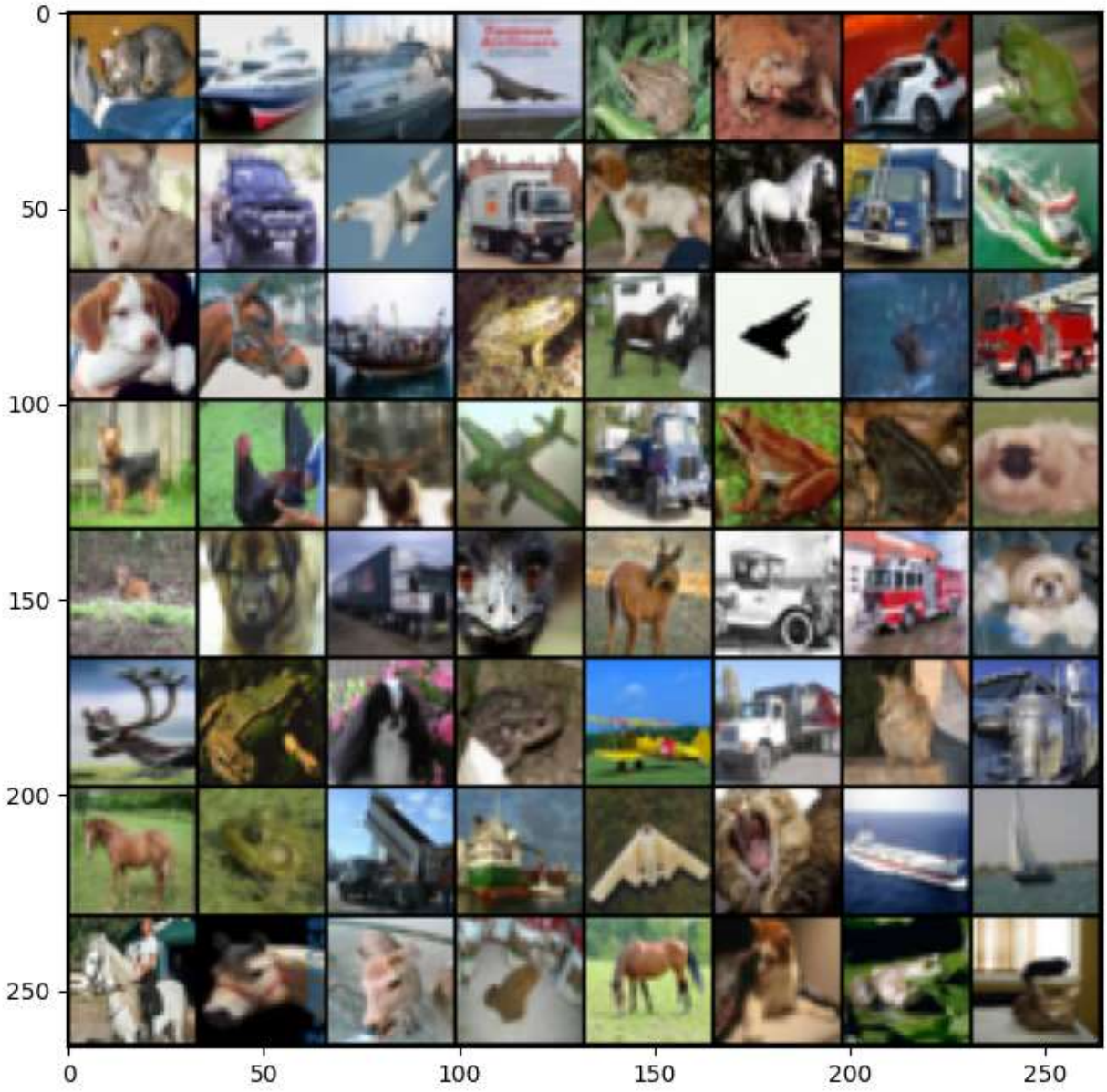
For this part, you need to complete code blocks marked with "Prob 2-4" above. Then run the following block.

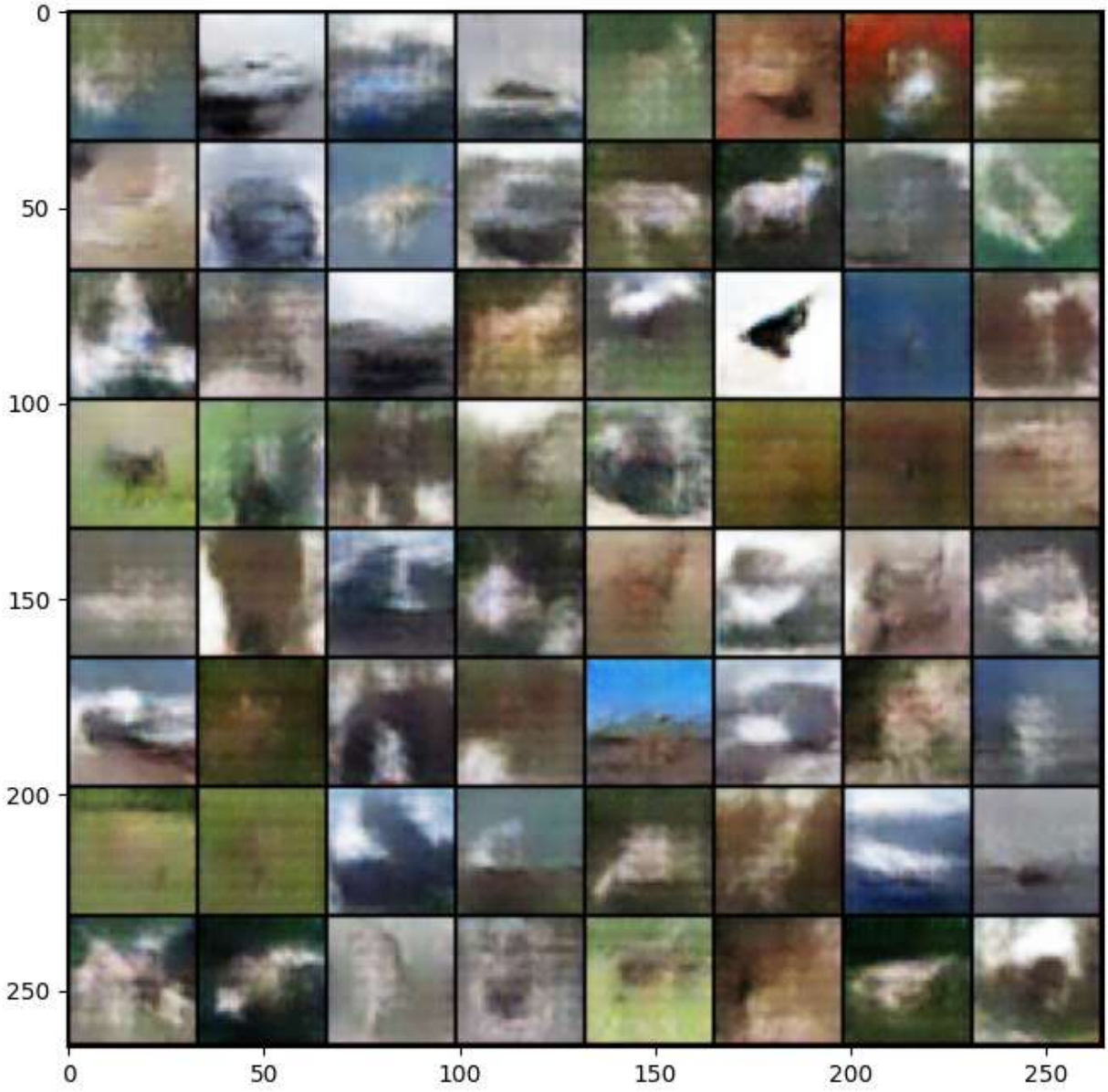
You need to achieve a reconstruction loss < 0.145. Do NOT modify anything outside of the blocks marked for you to fill in.

In [11]:
```
dcgan = DCGAN()
dcgan.load_state_dict(torch.load("dcgan.pt", map_location=device))

avg_loss, reconstructions = dcgan.reconstruct(test_samples[0:64])
print('average reconstruction loss = {0:.4f}'.format(avg_loss))
fig = plt.figure(figsize = (8, 8))
ax1 = plt.subplot(111)
ax1.imshow(make_grid(torch.from_numpy(test_samples[0:64]), padding=1).numpy().transpos
plt.show()
fig = plt.figure(figsize = (8, 8))
ax1 = plt.subplot(111)
ax1.imshow(make_grid(reconstructions, padding=1, normalize=True).numpy().transpose((1,
plt.show()
#Self-Note Result matches approved post: https://piazza.com/class/lcpa44ep1pk5aj/post/
```

```
average reconstruction loss = 0.0146
```





Submission Instruction

See the pinned Piazza post for detailed instruction.