Problem 2 - Generative Adversarial Networks (GAN)

- **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.

```
# 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`.

```
!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
```

```
--2023-04-03 14:49:47--  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  31.7MB/s    in 7.1s

2023-04-03 14:49:55 (22.8 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
```

```
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.

```
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=2, padding=1, bias=True),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.ConvTranspose2d(in_channels=64, out_channels=32, kernel_size=4,
stride=2, padding=1, bias=True),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(),
            nn.ConvTranspose2d(in_channels=32, out_channels=3, kernel_size=4, stride=2,
padding=1, bias=True),
            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
```

```
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=1, bias=True),
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2,
padding=1, bias=True),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2,
padding=1, bias=True),
            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
```

```
# 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(starting_shape=self.code_size).to(device)
        self._discriminator = Discriminator().to(device)

        # To-Do: Apply weight initialization (first implement the weight initialization
        # 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._optimizer_gen = torch.optim.Adam(
```

```
            self._generator.parameters(), lr=self.learning_rate, betas=betas)
        self._optimizer_dis = torch.optim.Adam(
            self._discriminator.parameters(), lr=self.learning_rate, betas=betas)



    # To-Do: Define weight initialization function
    # see link: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
    def _weight_initialization(self):
        for module in self.modules():
            if isinstance(module, nn.Conv2d) or isinstance(module, nn.ConvTranspose2d):
                # init conv with mean=0, stdev=0.02
                nn.init.normal_(module.weight.data, 0.0, 0.02)
            elif isinstance(module, nn.BatchNorm2d):
                # init bn with mean=1. stdev=0.02
                nn.init.normal_(module.weight.data, 1.0, 0.02)
                nn.init.constant_(module.bias.data, 0)


    # To-Do: Define a general classification loss function (sigmoid followed by binary
cross entropy loss)
    def _classification_loss(self, preds, labels):
        return nn.functional.binary_cross_entropy_with_logits(preds, labels)

 ###########################################################################
        #                                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.batch_size]
                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!
(.zero_grad())
                self._discriminator.zero_grad()

                # To-Do: feed real samples to the discriminator
                real_preds = 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_preds, 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
                noises = torch.randn([self.batch_size, self.code_size], device=device)

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

                # 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 to
                # receive gradient info from the Generator
                fake_preds = 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_preds, 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._optimizer_dis.step()


        ########################################################################
                # Prob 2-1: Train the generator
              #

        ########################################################################

                # To-Do: Remember to eliminate all generator gradients first!
        (.zero_grad())
                self._generator.zero_grad()

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

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

                # 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.
                fake_preds = self._discriminator(fake_samples)

                # 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(fake_preds, real_label)

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

                # To-Do: Update the generator weights
                self._optimizer_gen.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_factor)
                plot_gen_s = plot_gen_s * smooth_factor + gen_loss * (1 -
smooth_factor)
                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}'.format(step, max_steps, dis_loss, gen_loss))

            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().detach(),
padding=1, normalize=True).numpy().transpose((1, 2, 0)))
                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, preds, labels):
        return nn.functional.mse_loss(preds, labels)
```

```
    def reconstruct(self, samples):
        recon_code = torch.zeros([samples.shape[0], self.code_size], device=device,
requires_grad=True)
        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 optimizer
        self._optimizer_rec = 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-
4   #

 ##############################################################################
            # To-Do: eliminate the gradients
            self._optimizer_rec.zero_grad()

            # To-Do: feed the reconstruction codes to the generator for generating
reconstructed samples
            # 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_rec.step()


 ##########################################################################
            #                                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,
requires_grad=True)
        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()
```

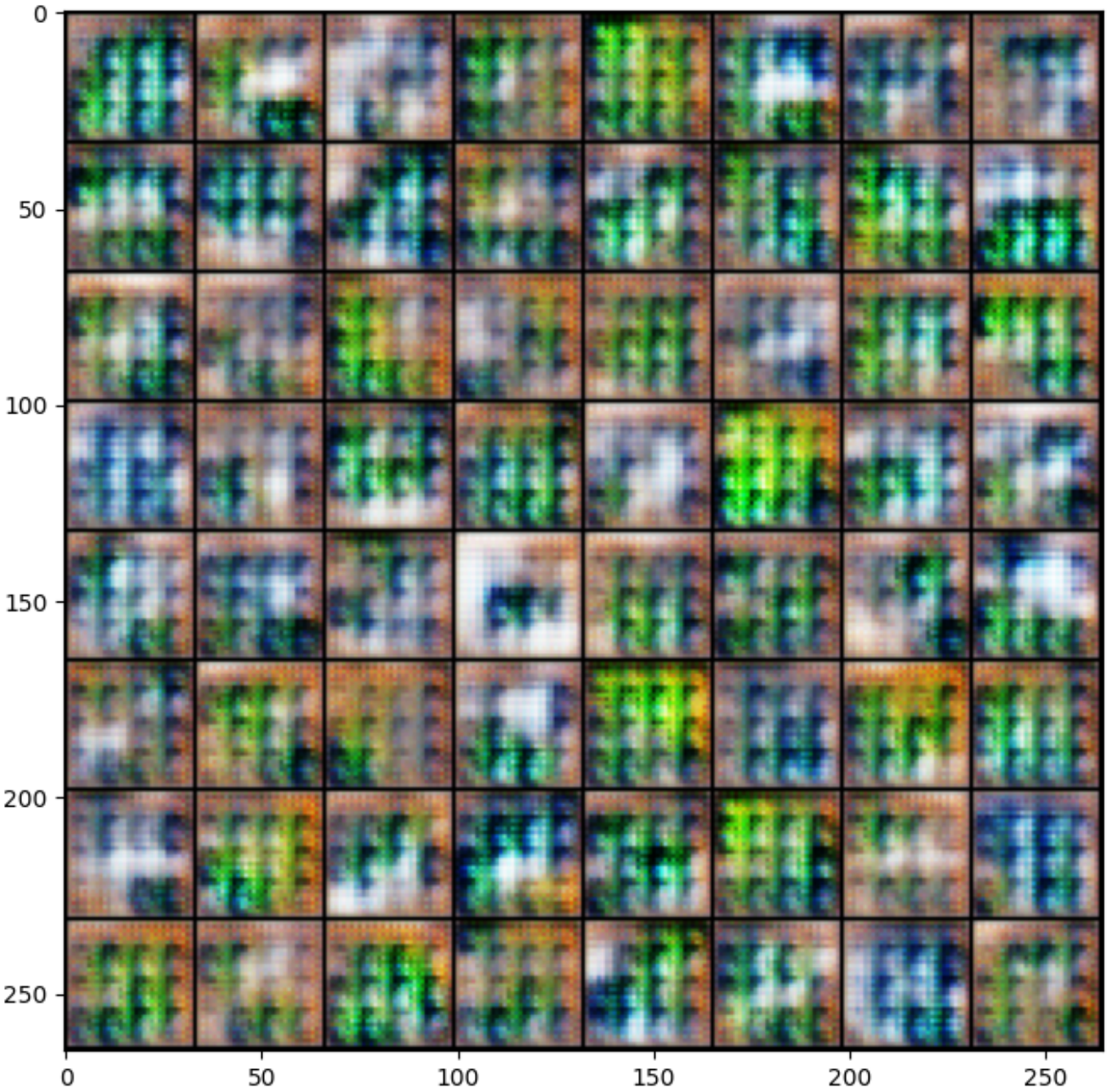
Now let's do the training!

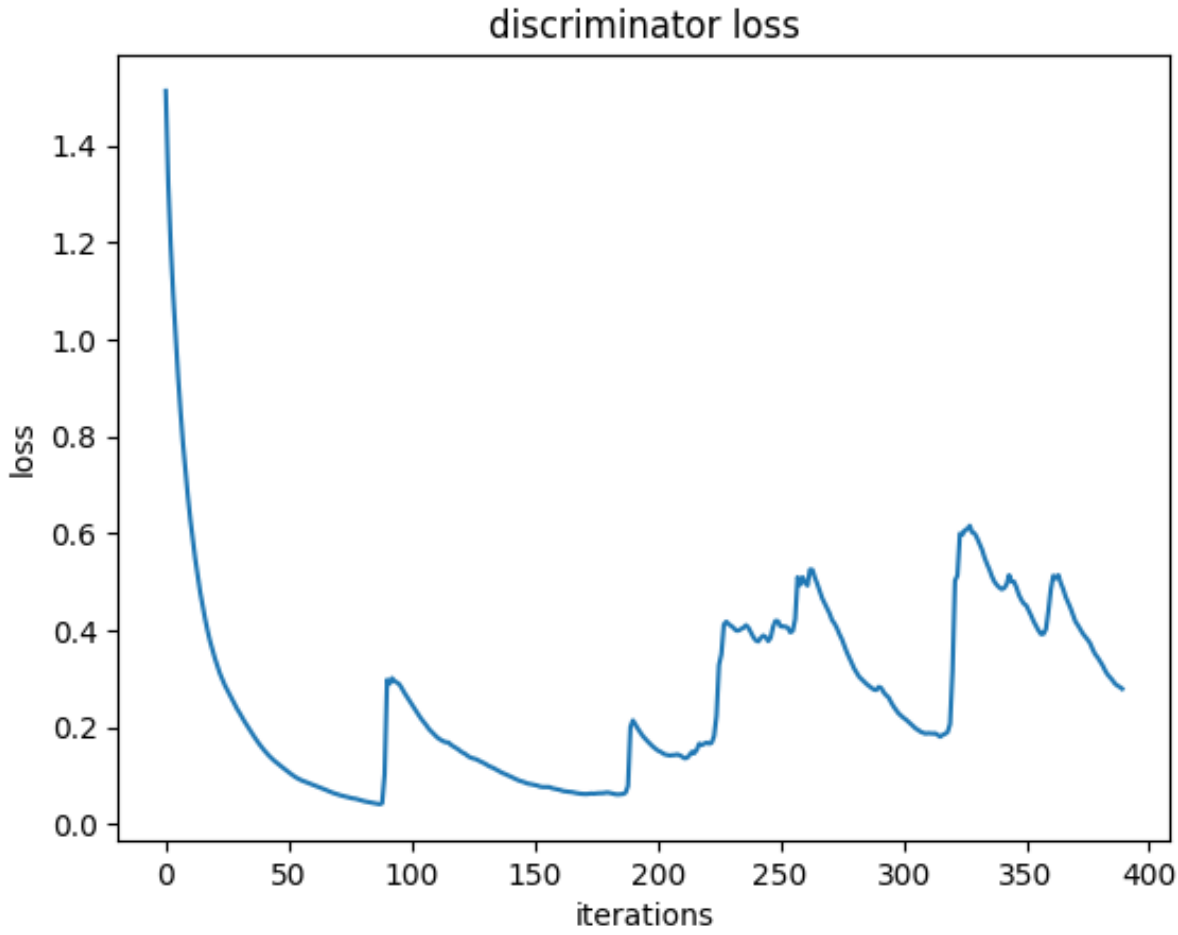
Don't panic if the loss curve goes wild. The two networks are competing for the loss curve to go different directions, so virtually anything can happen. If your code is correct, the generated samples should have a high variety.

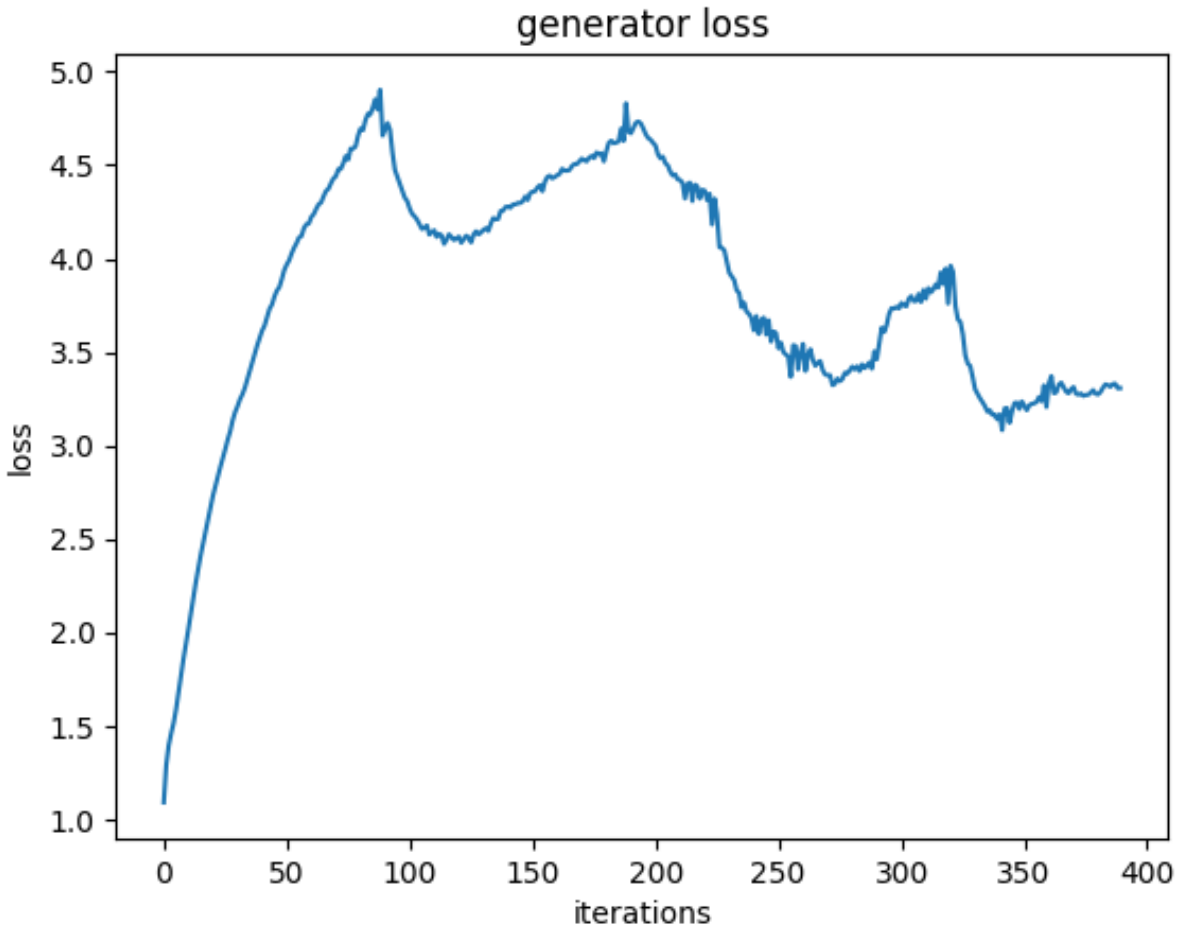
Do NOT change the number of epochs, learning rate, or batch size. If you're using Google Colab, the batch size will not be an issue during training.

```
set_seed(42)

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

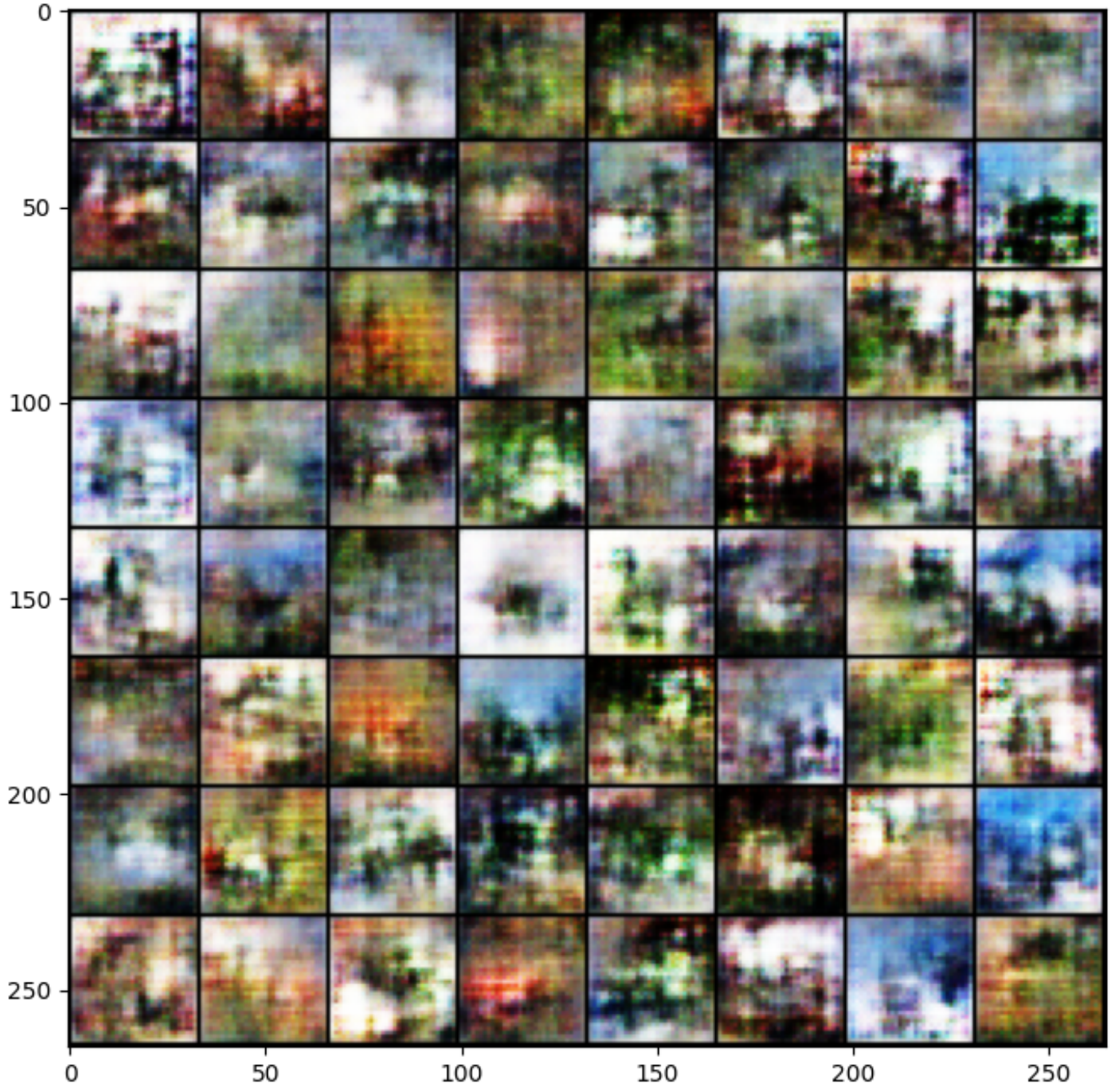
```
Start training ...
Iteration 100/9750: dis loss = 0.0924, gen loss = 3.9709
Iteration 200/9750: dis loss = 0.0729, gen loss = 4.3005
Iteration 300/9750: dis loss = 0.1352, gen loss = 3.5841
```

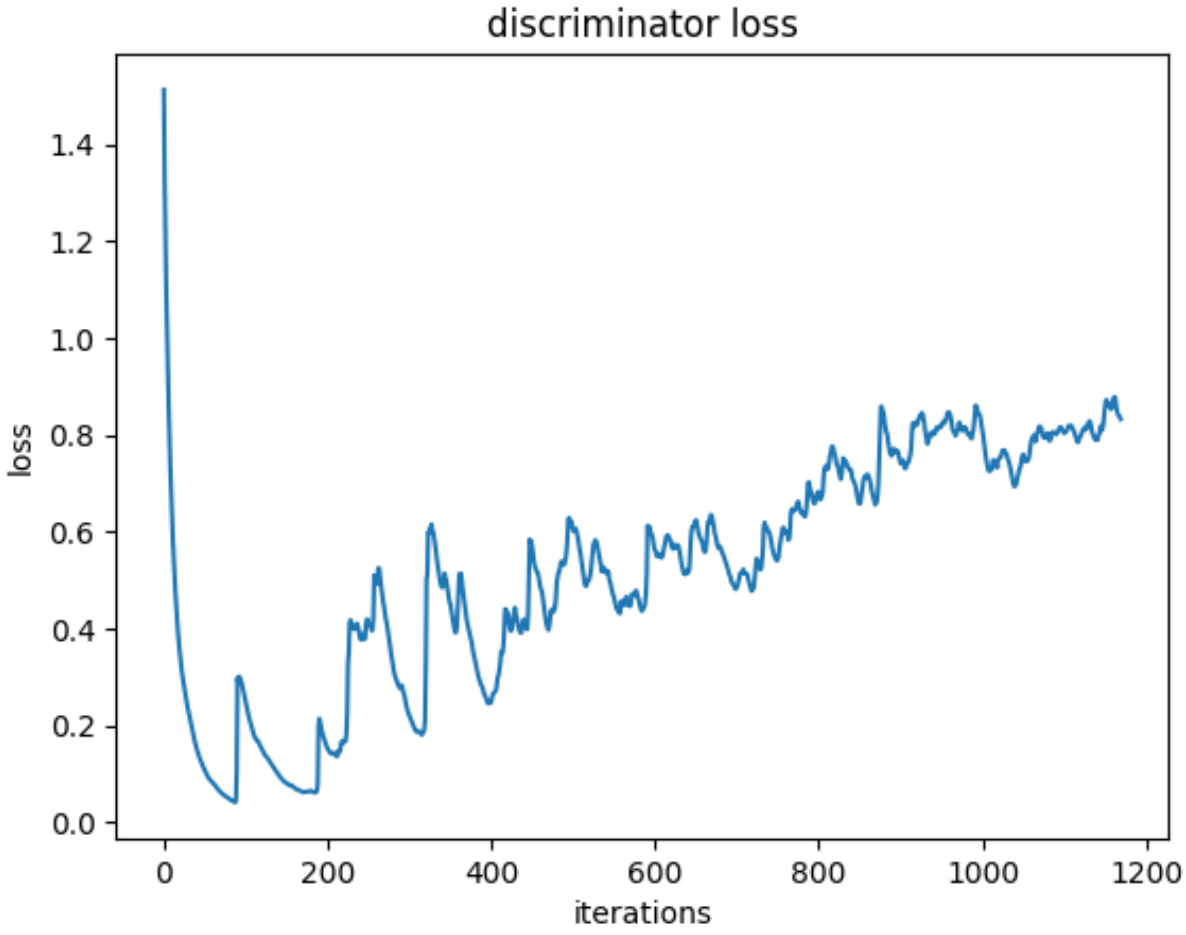


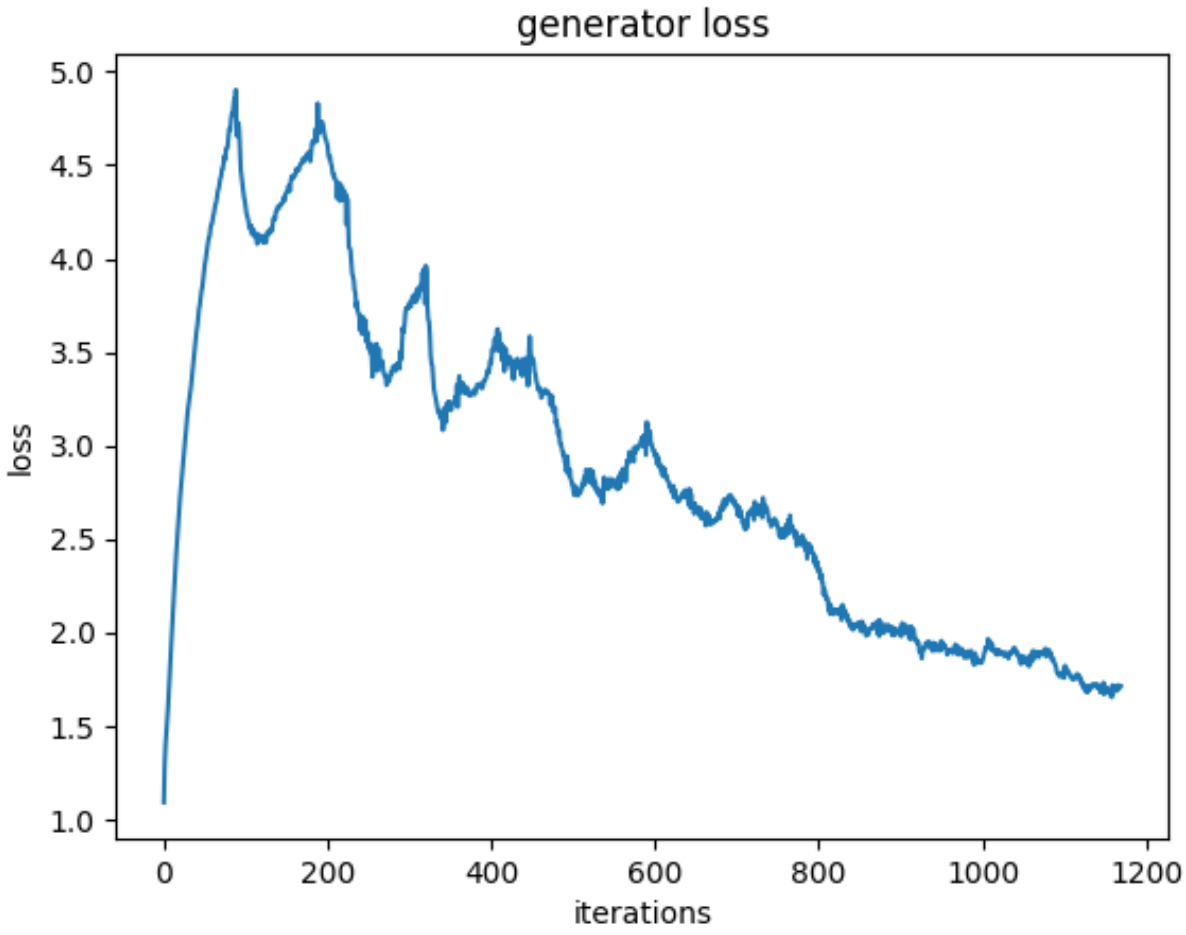

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
50
100
150
200
250
300
350
400




```
Iteration 400/9750: dis loss = 0.1822, gen loss = 4.0839
Iteration 500/9750: dis loss = 0.4507, gen loss = 2.3115
Iteration 600/9750: dis loss = 0.3485, gen loss = 3.0100
Iteration 700/9750: dis loss = 0.4557, gen loss = 2.1347
Iteration 800/9750: dis loss = 0.7396, gen loss = 1.3783
Iteration 900/9750: dis loss = 0.6372, gen loss = 1.8775
Iteration 1000/9750: dis loss = 0.6071, gen loss = 1.5943
Iteration 1100/9750: dis loss = 0.7467, gen loss = 2.1148
```

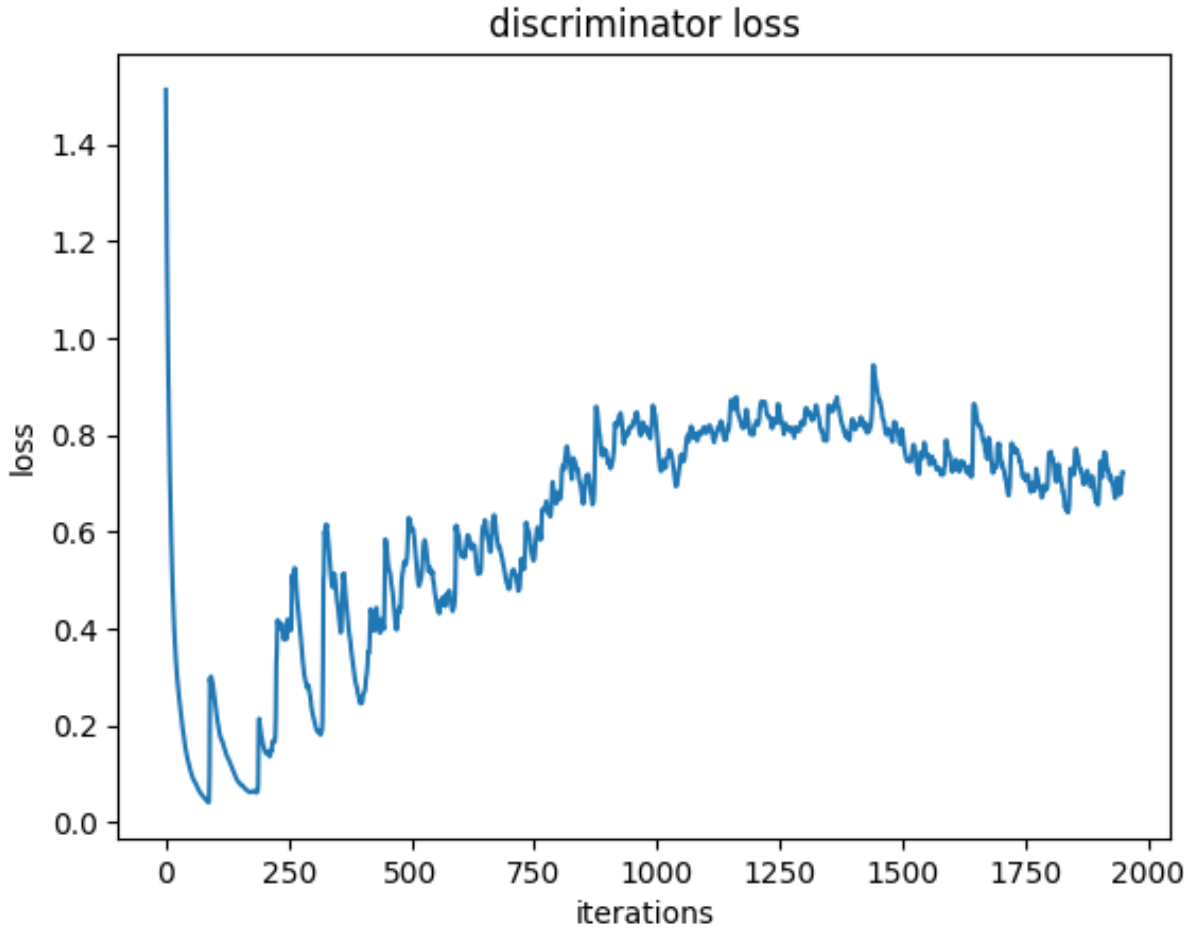

0
50
100
150
200
250
0
50
100
150
200
250

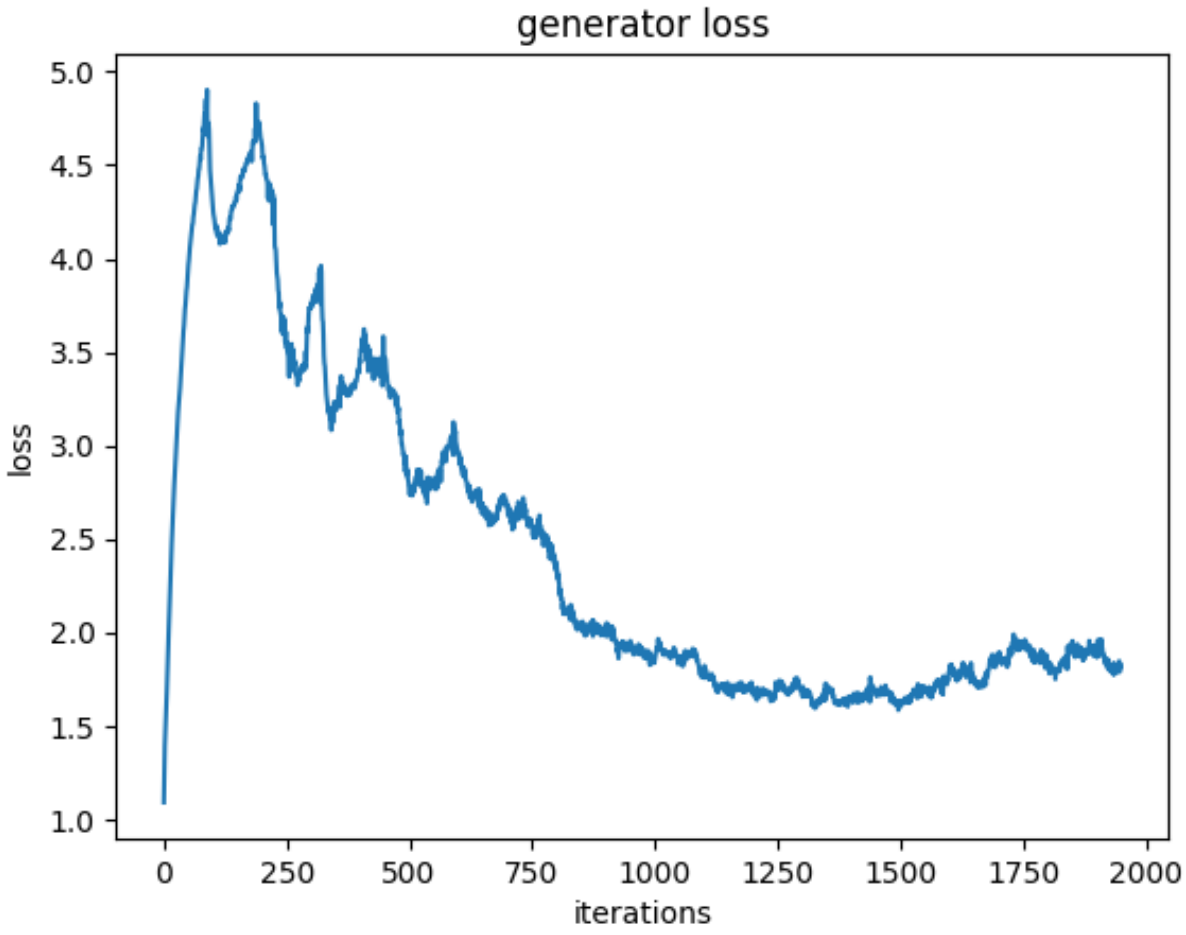

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
200
400
600
800
1000
1200




```
Iteration 1200/9750: dis loss = 1.0768, gen loss = 2.2291
Iteration 1300/9750: dis loss = 0.7528, gen loss = 1.7763
Iteration 1400/9750: dis loss = 0.8526, gen loss = 1.5880
Iteration 1500/9750: dis loss = 0.8896, gen loss = 2.2377
Iteration 1600/9750: dis loss = 0.5000, gen loss = 2.4446
Iteration 1700/9750: dis loss = 0.5523, gen loss = 1.4741
Iteration 1800/9750: dis loss = 1.0443, gen loss = 2.1691
Iteration 1900/9750: dis loss = 0.6483, gen loss = 1.3259
```

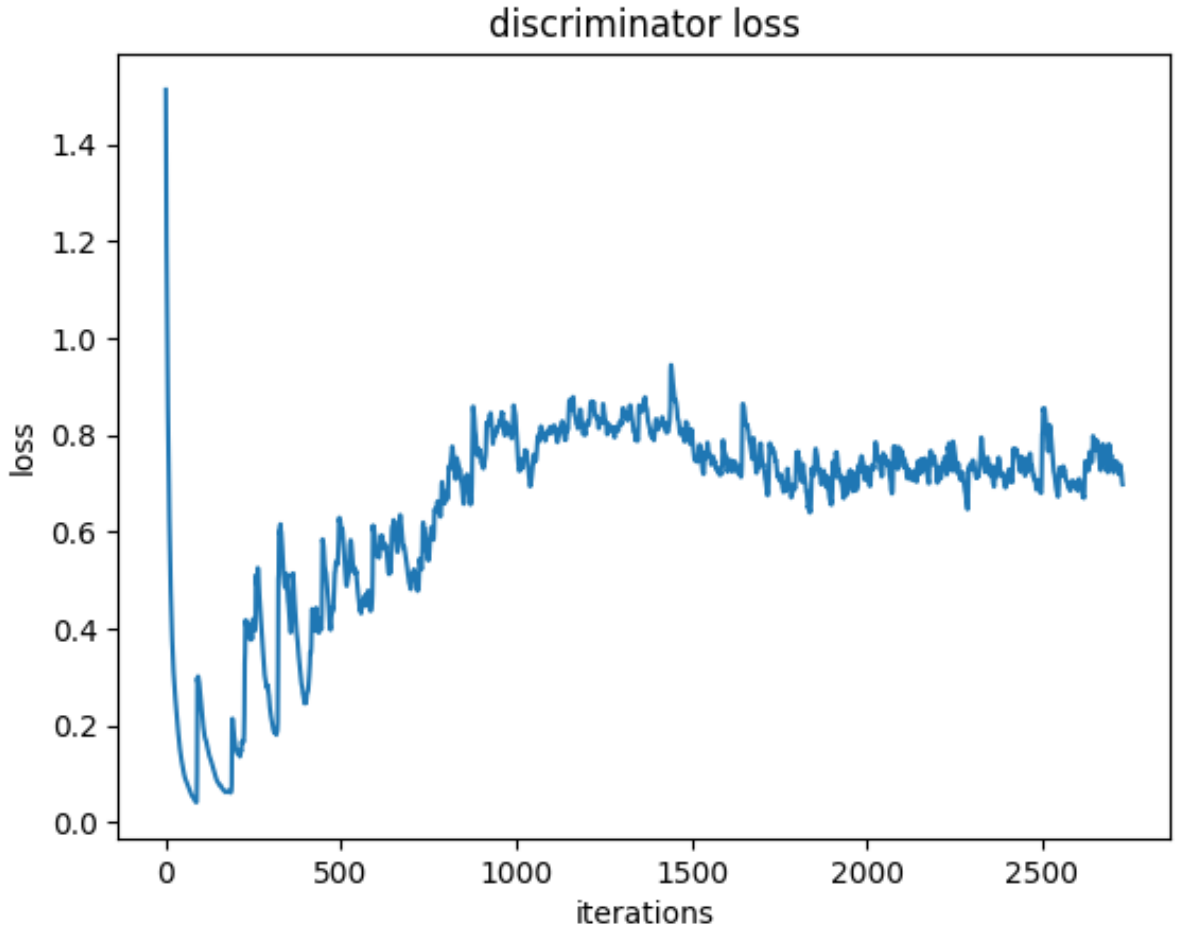

0
50
100
150
200
250
0
50
100
150
200
250

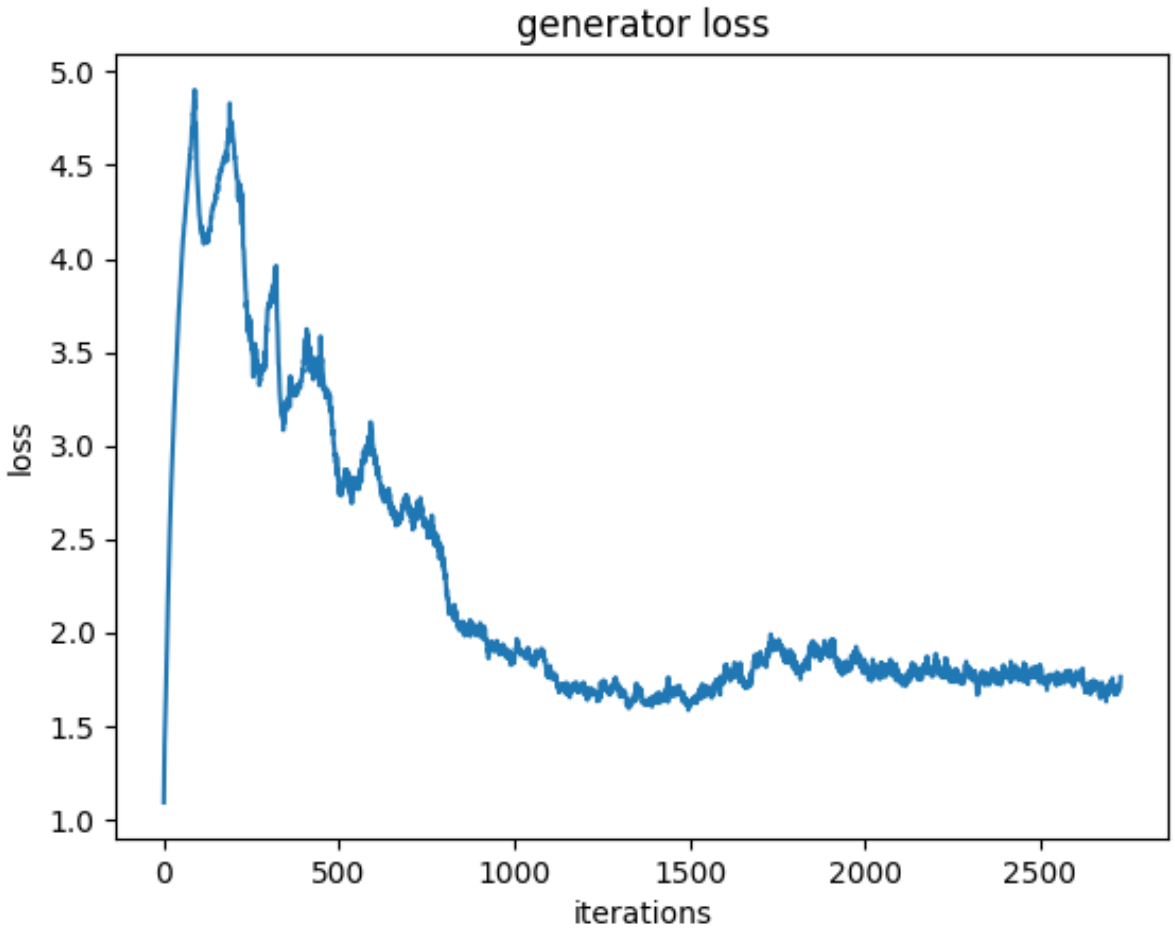

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
250
500
750
1000
1250
1500
1750
2000




```
Iteration 2000/9750: dis loss = 0.7101, gen loss = 2.7094
Iteration 2100/9750: dis loss = 0.8773, gen loss = 2.2992
Iteration 2200/9750: dis loss = 0.5071, gen loss = 1.6112
Iteration 2300/9750: dis loss = 0.7701, gen loss = 2.4130
Iteration 2400/9750: dis loss = 0.7735, gen loss = 1.3677
Iteration 2500/9750: dis loss = 2.2392, gen loss = 3.6571
Iteration 2600/9750: dis loss = 0.5492, gen loss = 2.1036
Iteration 2700/9750: dis loss = 0.6083, gen loss = 2.0433
```

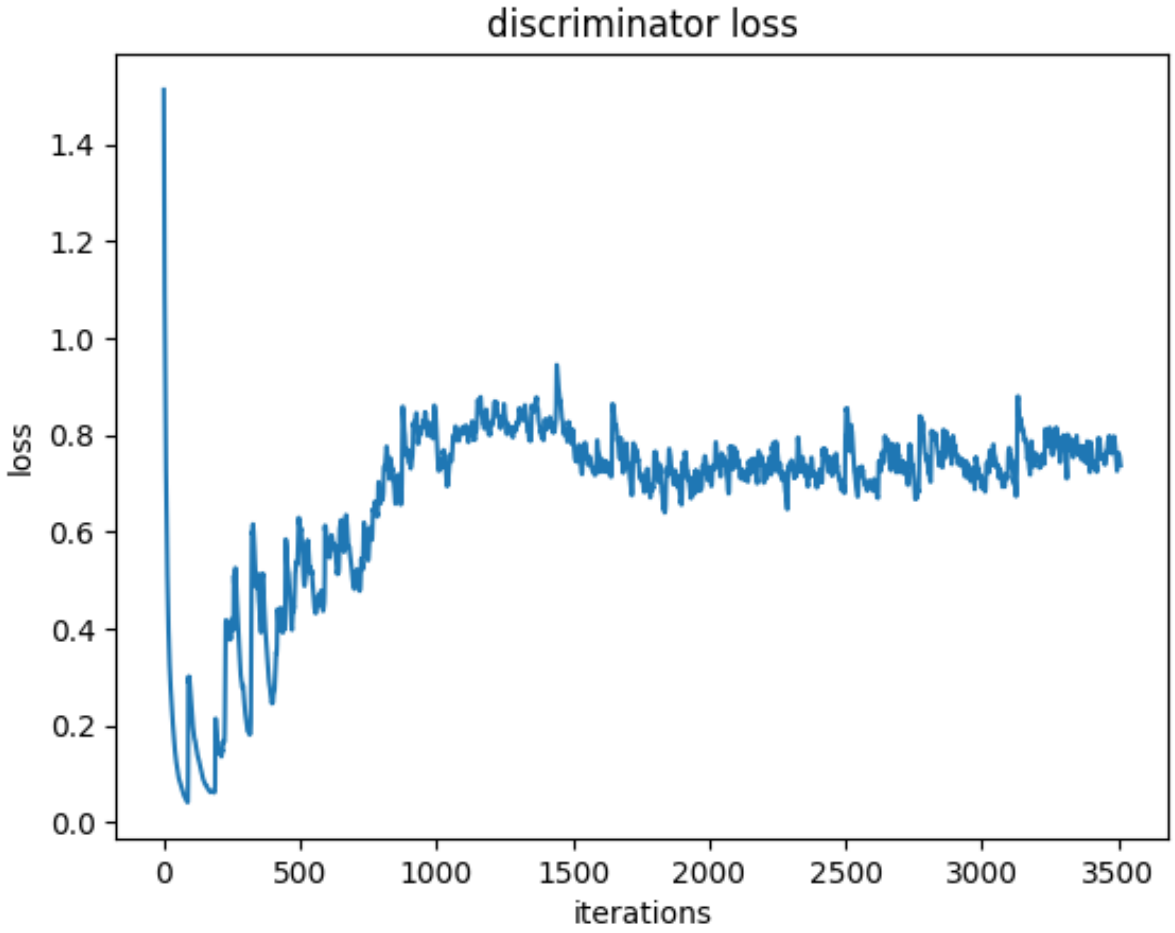

0
50
100
150
200
250
0
50
100
150
200
250

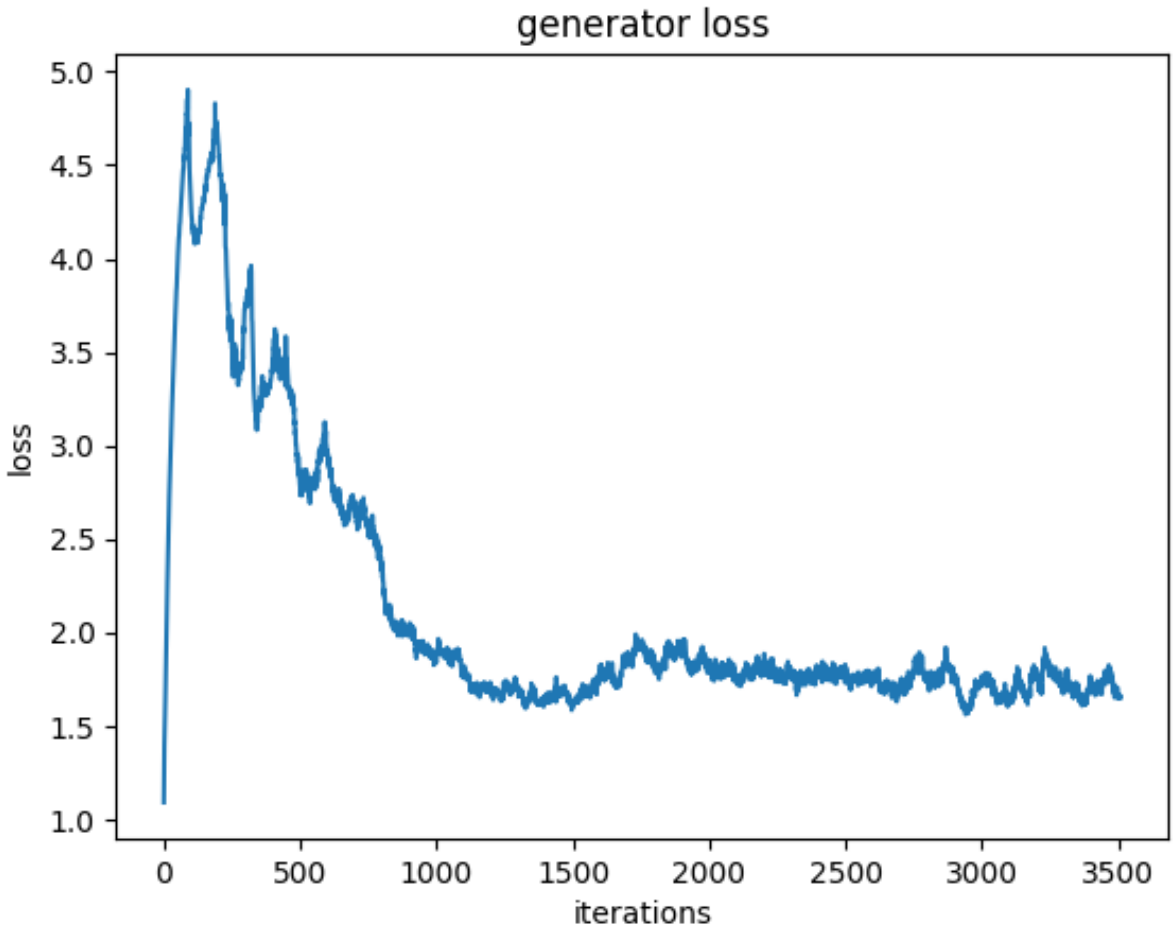

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
500
1000
1500
2000
2500



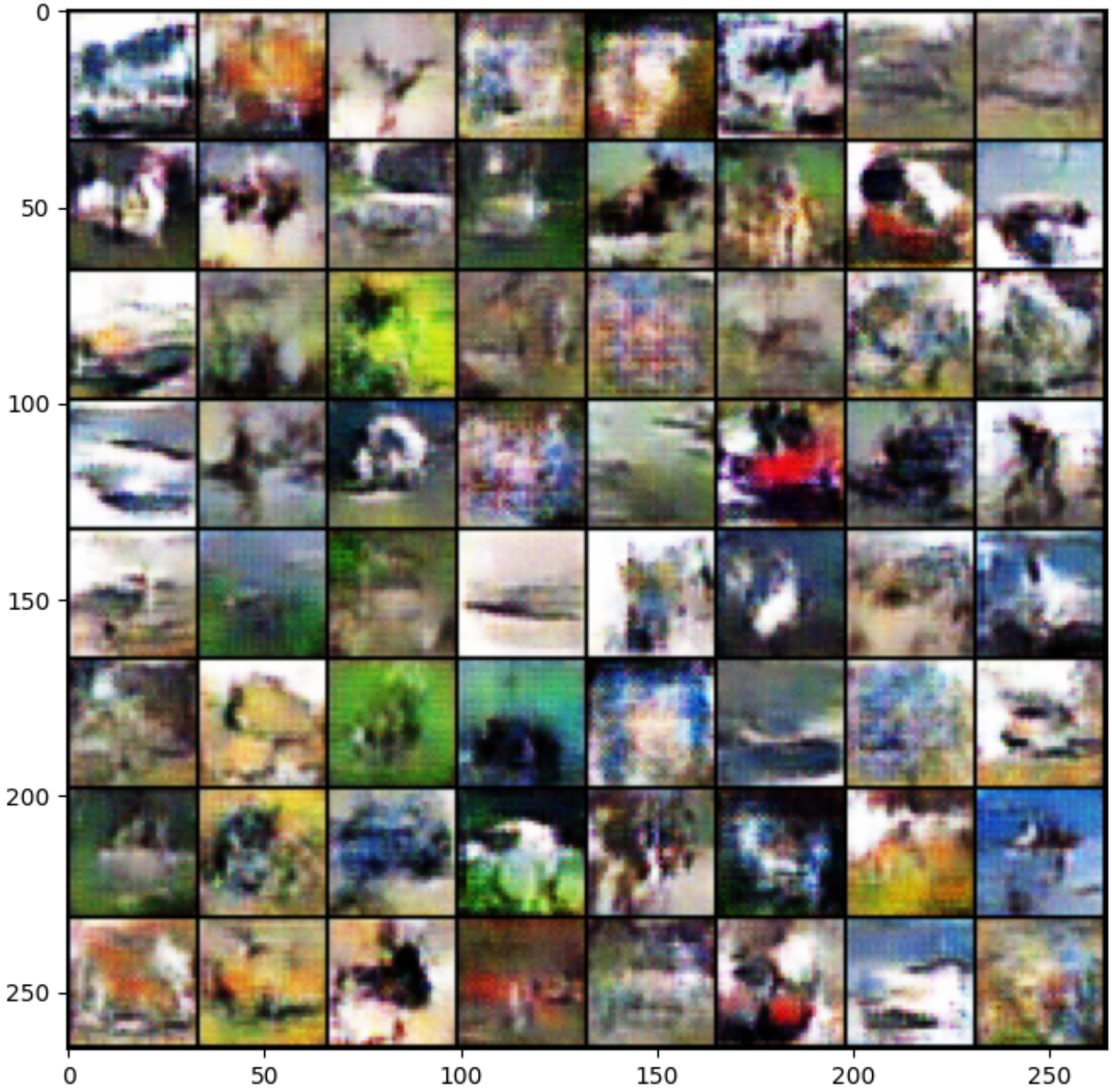

```
Iteration 2800/9750: dis loss = 0.7362, gen loss = 1.8398
Iteration 2900/9750: dis loss = 0.6924, gen loss = 1.3191
Iteration 3000/9750: dis loss = 0.5558, gen loss = 1.9200
Iteration 3100/9750: dis loss = 0.7337, gen loss = 2.0633
Iteration 3200/9750: dis loss = 0.6704, gen loss = 2.6113
Iteration 3300/9750: dis loss = 0.9168, gen loss = 1.0646
Iteration 3400/9750: dis loss = 1.4052, gen loss = 0.7715
Iteration 3500/9750: dis loss = 1.0345, gen loss = 1.9875
```

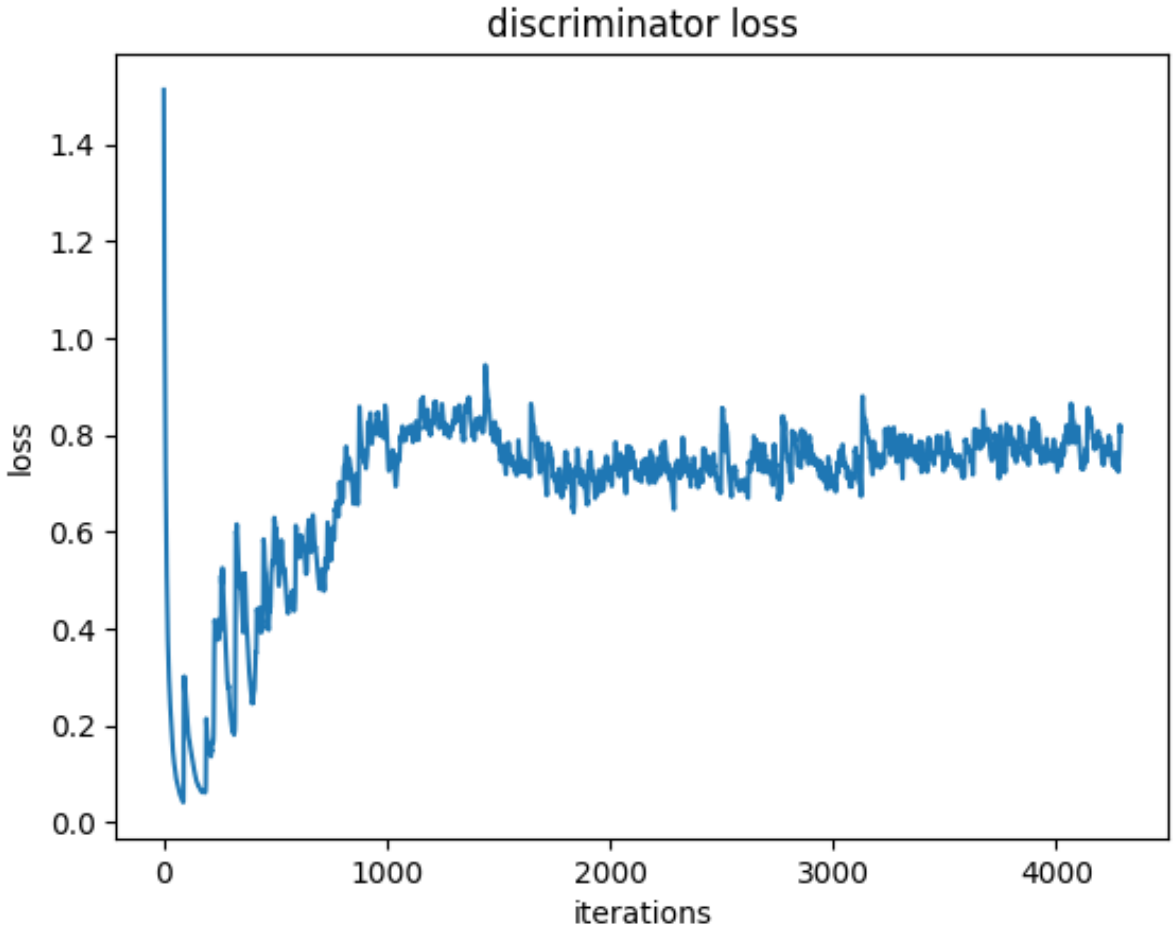

0
50
100
150
200
250
0
50
100
150
200
250

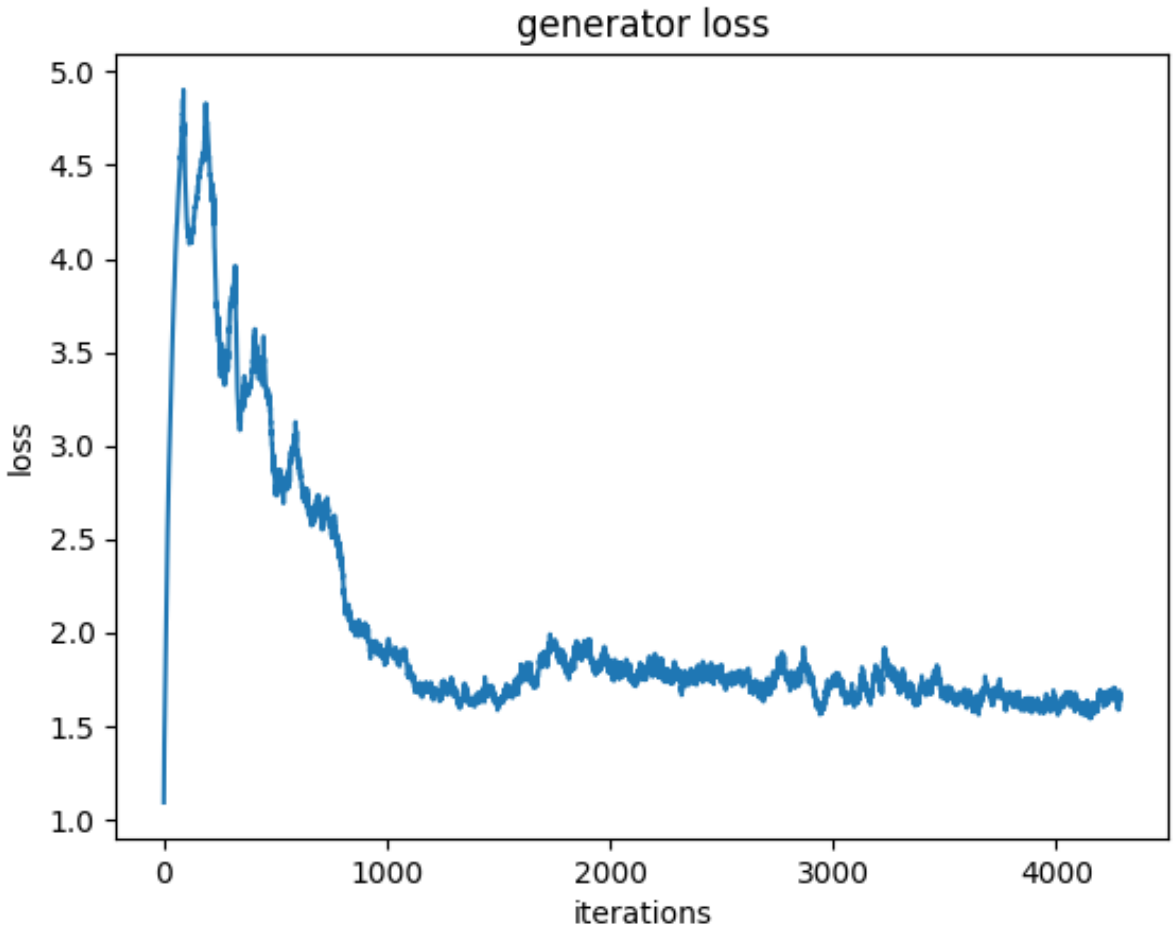

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
500
1000
1500
2000
2500
3000
3500



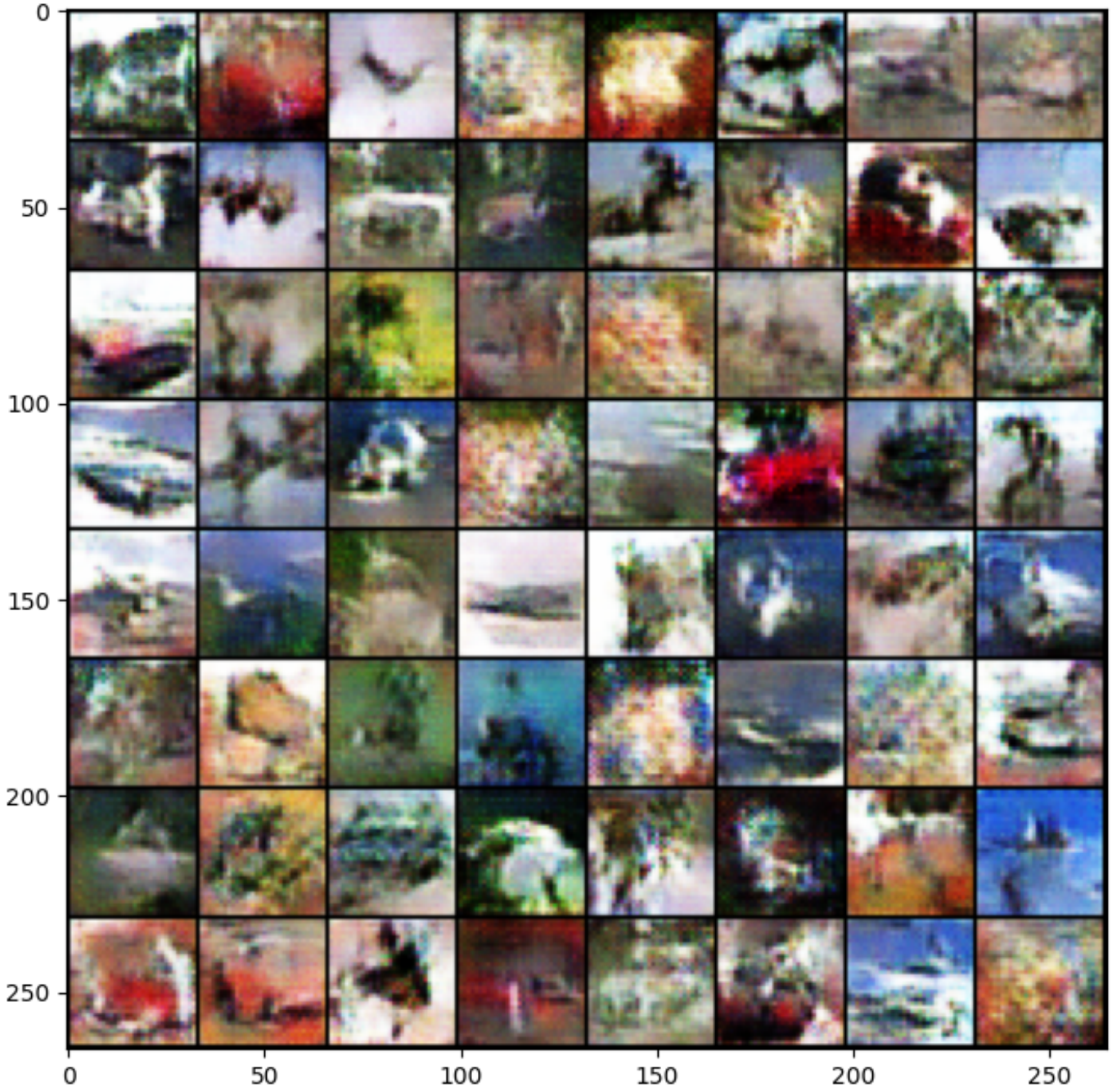

```
Iteration 3600/9750: dis loss = 0.6485, gen loss = 1.6018
Iteration 3700/9750: dis loss = 0.5656, gen loss = 2.0181
Iteration 3800/9750: dis loss = 0.7206, gen loss = 2.3437
Iteration 3900/9750: dis loss = 0.9910, gen loss = 2.5414
Iteration 4000/9750: dis loss = 0.8126, gen loss = 1.4000
Iteration 4100/9750: dis loss = 0.7141, gen loss = 1.7153
Iteration 4200/9750: dis loss = 0.8041, gen loss = 1.6754
```

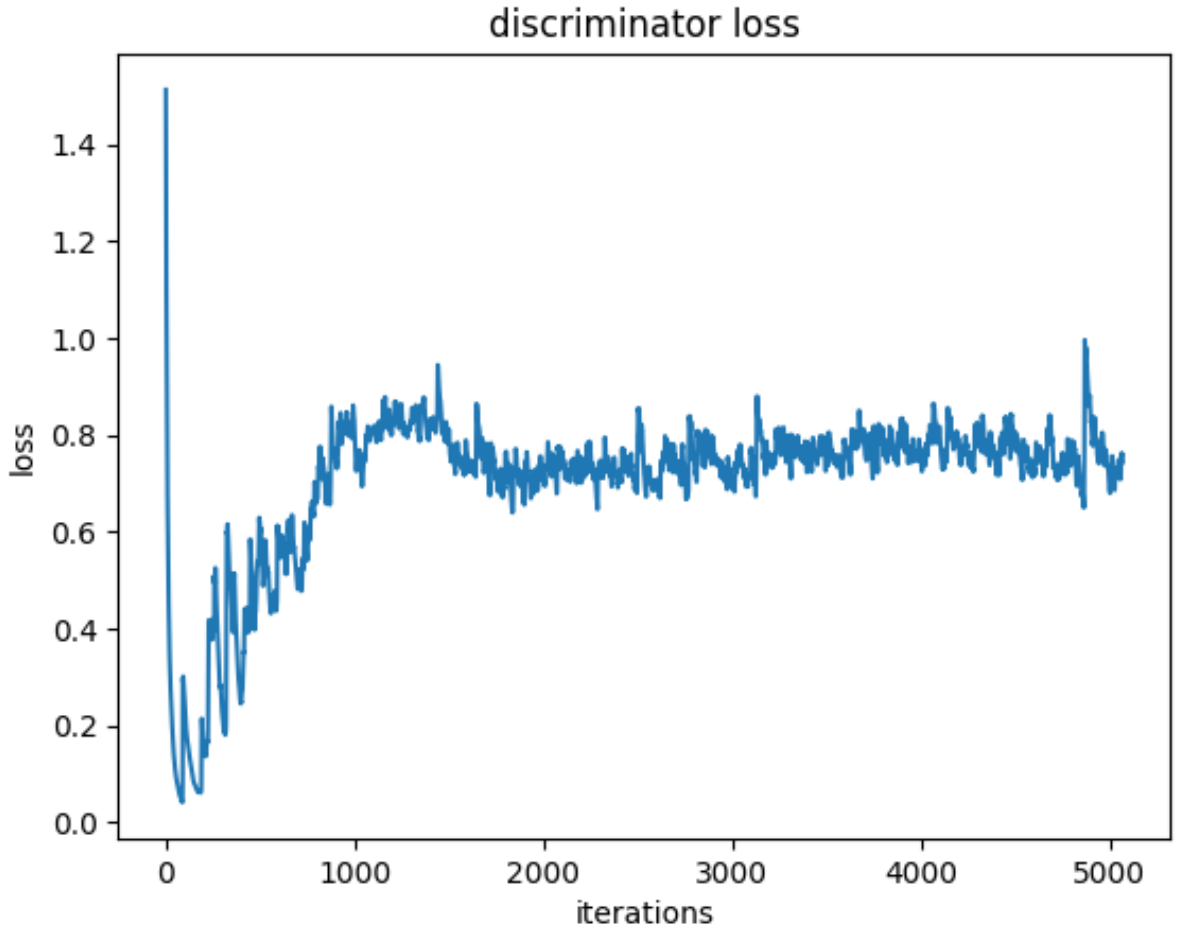

0
50
100
150
200
250
0
50
100
150
200
250

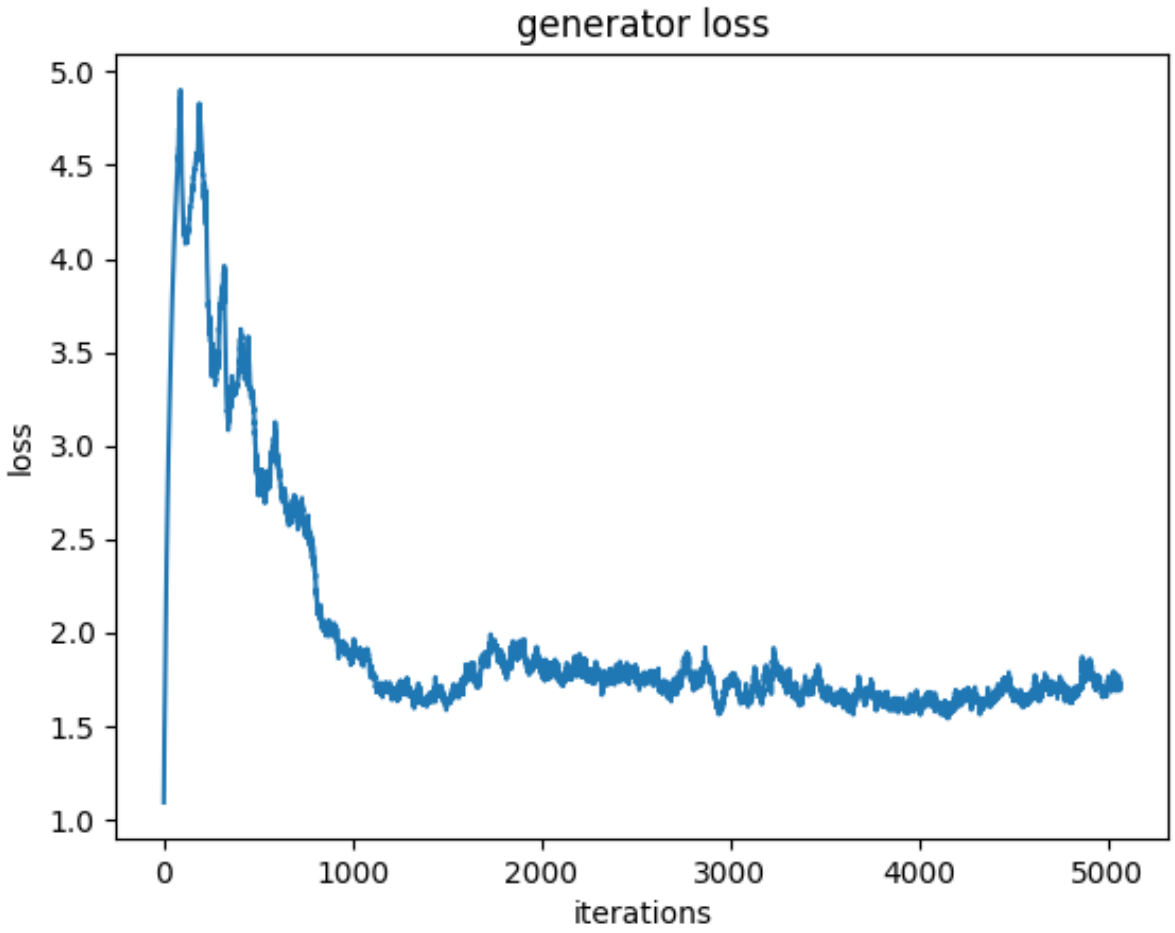

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
1000
2000
3000
4000



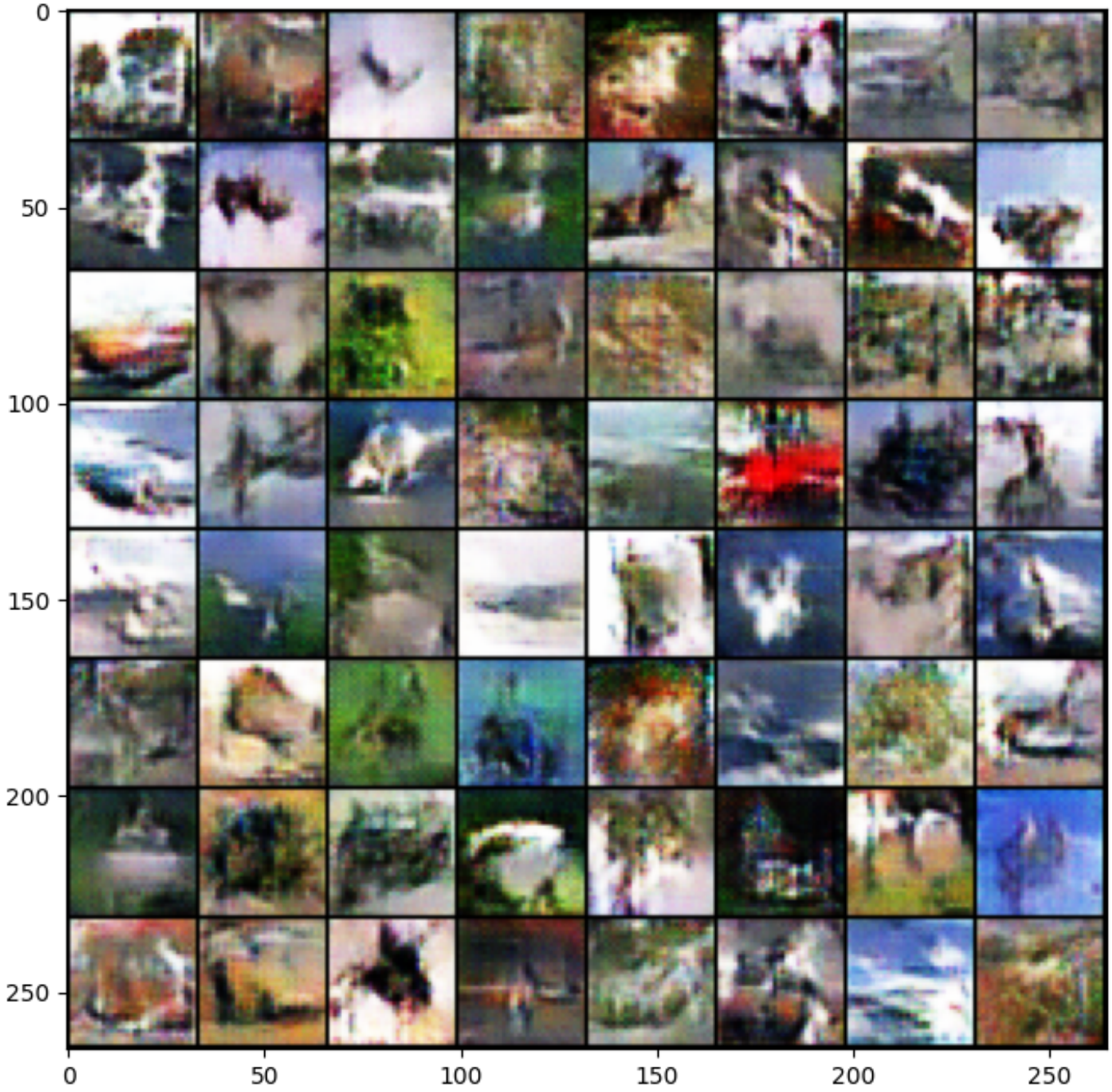

```
Iteration 4300/9750: dis loss = 1.4217, gen loss = 0.6892
Iteration 4400/9750: dis loss = 0.6072, gen loss = 1.9813
Iteration 4500/9750: dis loss = 0.6546, gen loss = 1.4731
Iteration 4600/9750: dis loss = 0.4628, gen loss = 2.3115
Iteration 4700/9750: dis loss = 0.7371, gen loss = 1.9661
Iteration 4800/9750: dis loss = 0.7202, gen loss = 1.9724
Iteration 4900/9750: dis loss = 0.6230, gen loss = 2.4505
Iteration 5000/9750: dis loss = 0.5159, gen loss = 1.7756
```

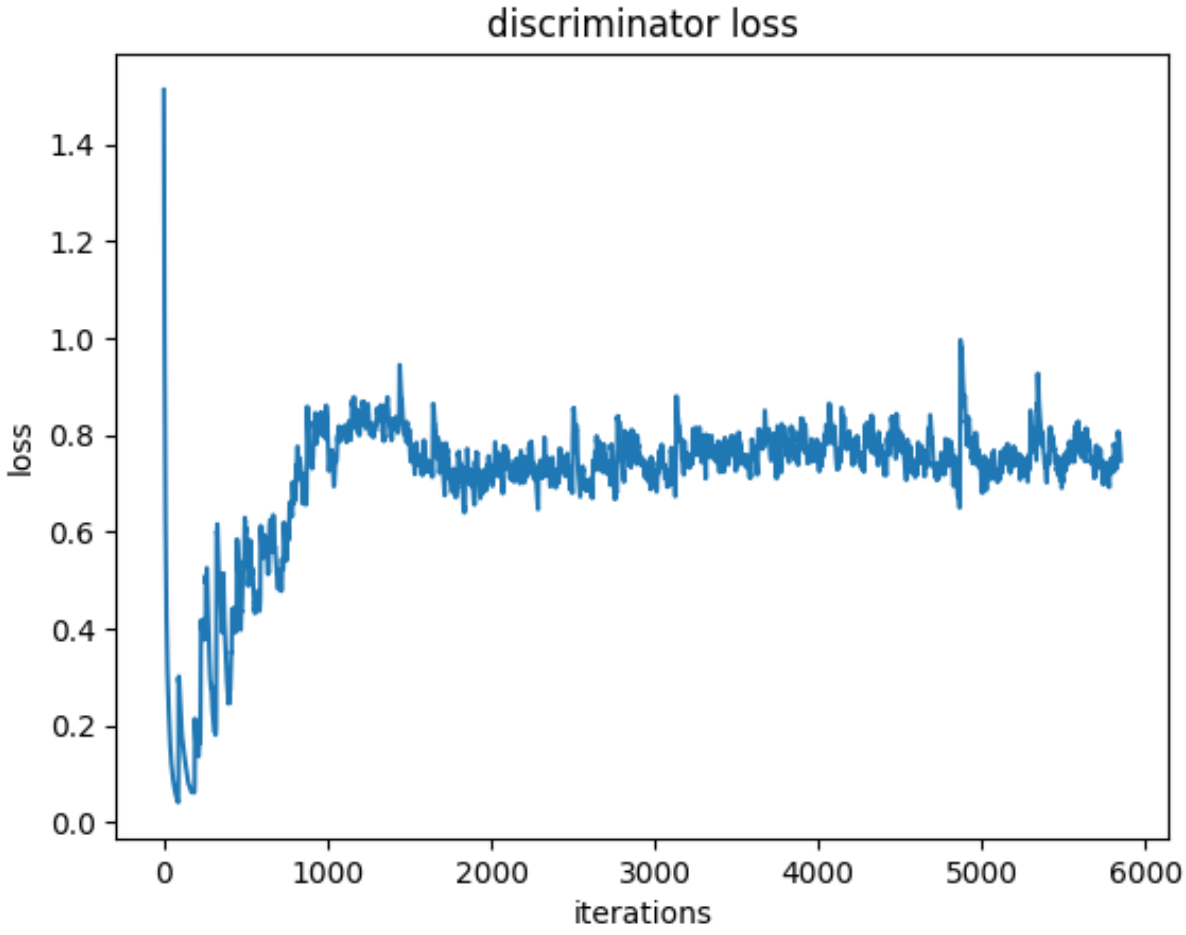

0
50
100
150
200
250
0
50
100
150
200
250

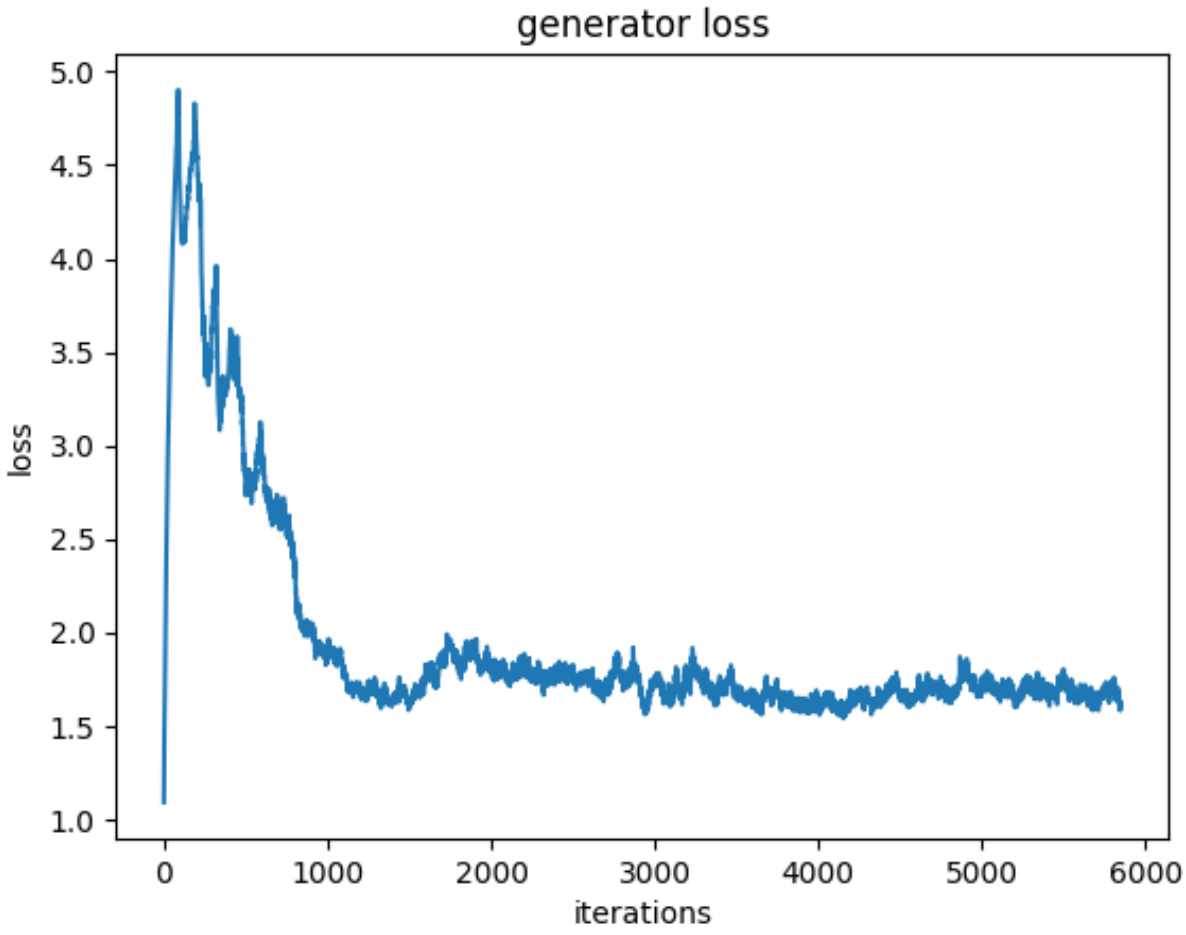

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
1000
2000
3000
4000
5000




```
Iteration 5100/9750: dis loss = 0.7616, gen loss = 1.5360
Iteration 5200/9750: dis loss = 0.5930, gen loss = 2.0320
Iteration 5300/9750: dis loss = 0.7452, gen loss = 0.8170
Iteration 5400/9750: dis loss = 1.0267, gen loss = 0.6306
Iteration 5500/9750: dis loss = 1.0222, gen loss = 0.4792
Iteration 5600/9750: dis loss = 0.6256, gen loss = 1.5588
Iteration 5700/9750: dis loss = 0.7744, gen loss = 2.3363
Iteration 5800/9750: dis loss = 0.7608, gen loss = 1.3266
```

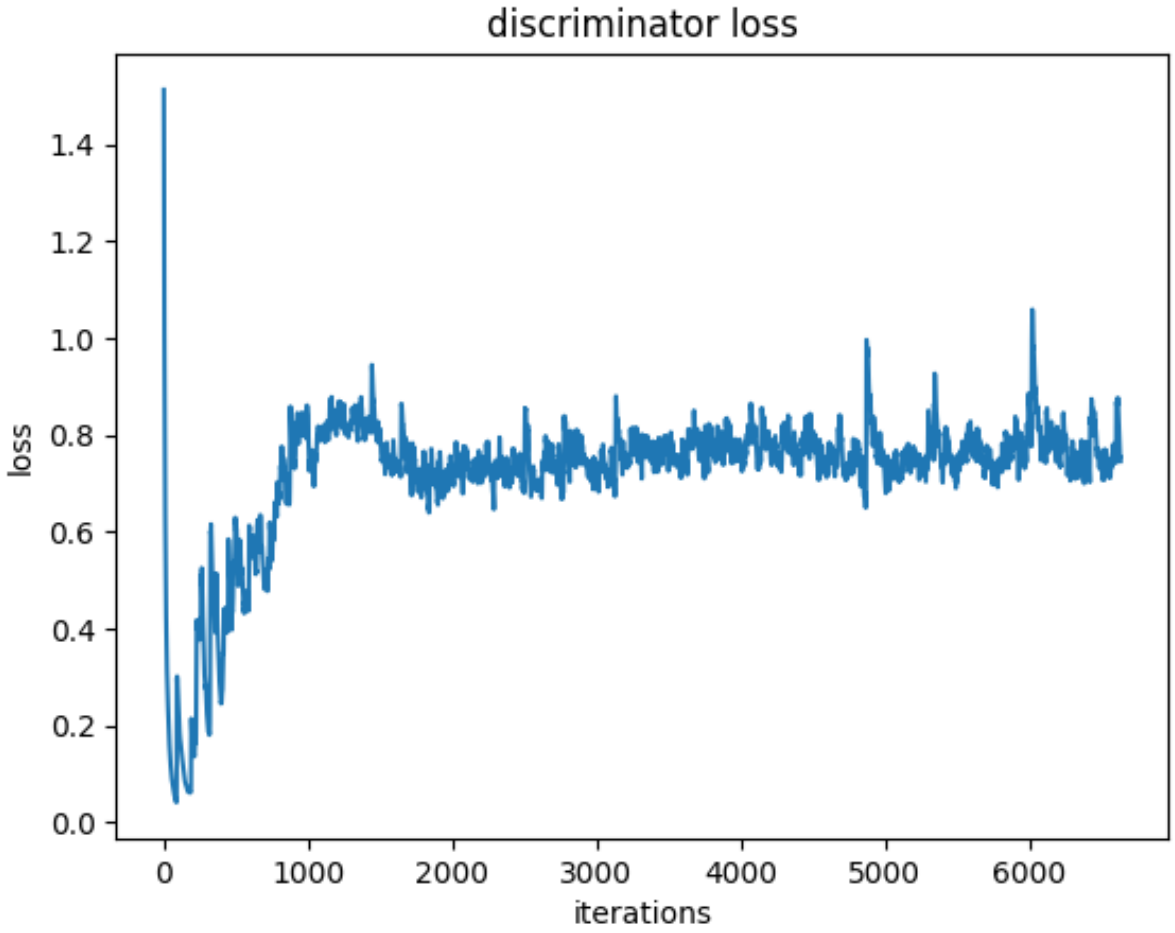

0
50
100
150
200
250
0
50
100
150
200
250

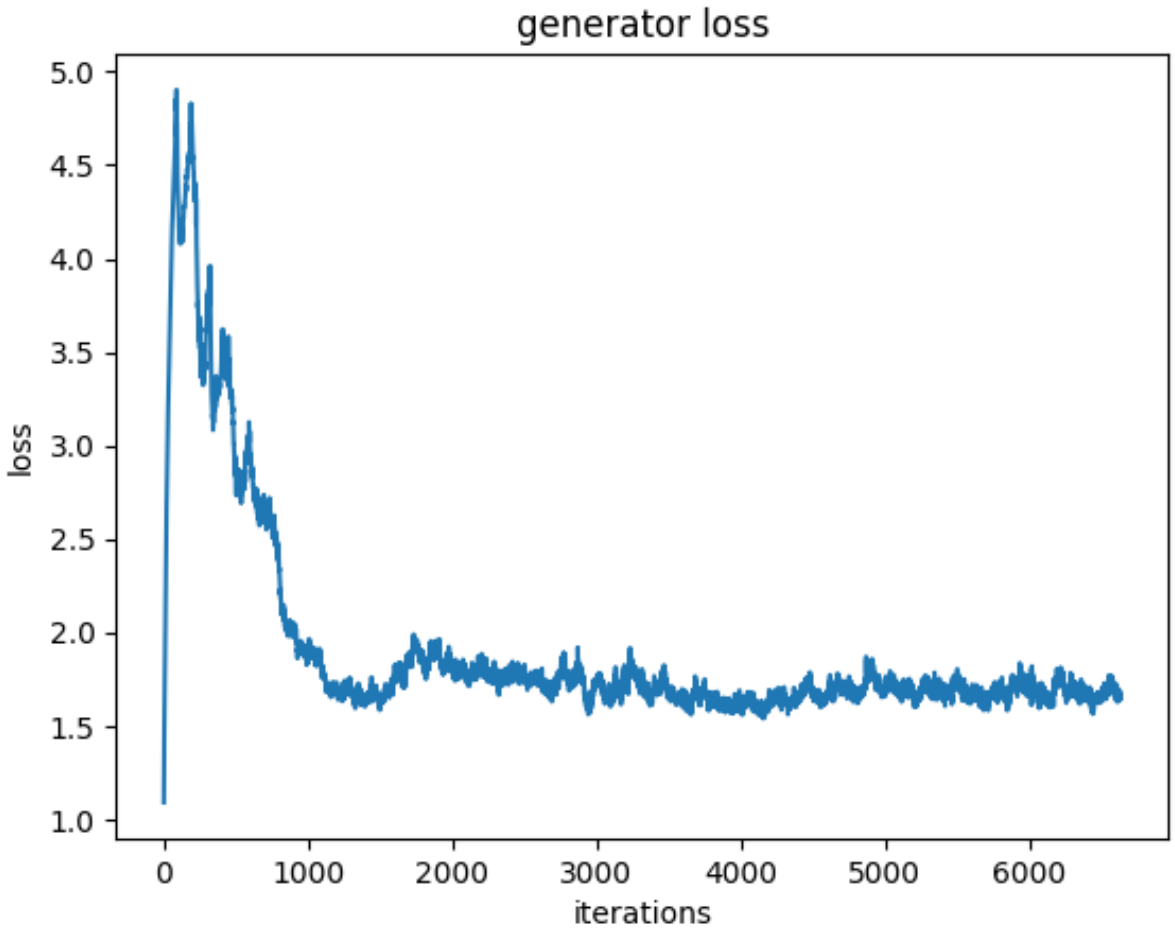

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
1000
2000
3000
4000
5000
6000




```
Iteration 5900/9750: dis loss = 0.8927, gen loss = 1.3969
Iteration 6000/9750: dis loss = 0.5900, gen loss = 1.8974
Iteration 6100/9750: dis loss = 0.5880, gen loss = 2.2768
Iteration 6200/9750: dis loss = 0.9764, gen loss = 1.2754
Iteration 6300/9750: dis loss = 0.6458, gen loss = 1.3111
Iteration 6400/9750: dis loss = 0.8739, gen loss = 2.3438
Iteration 6500/9750: dis loss = 0.6600, gen loss = 1.2879
Iteration 6600/9750: dis loss = 1.8391, gen loss = 0.5187
```

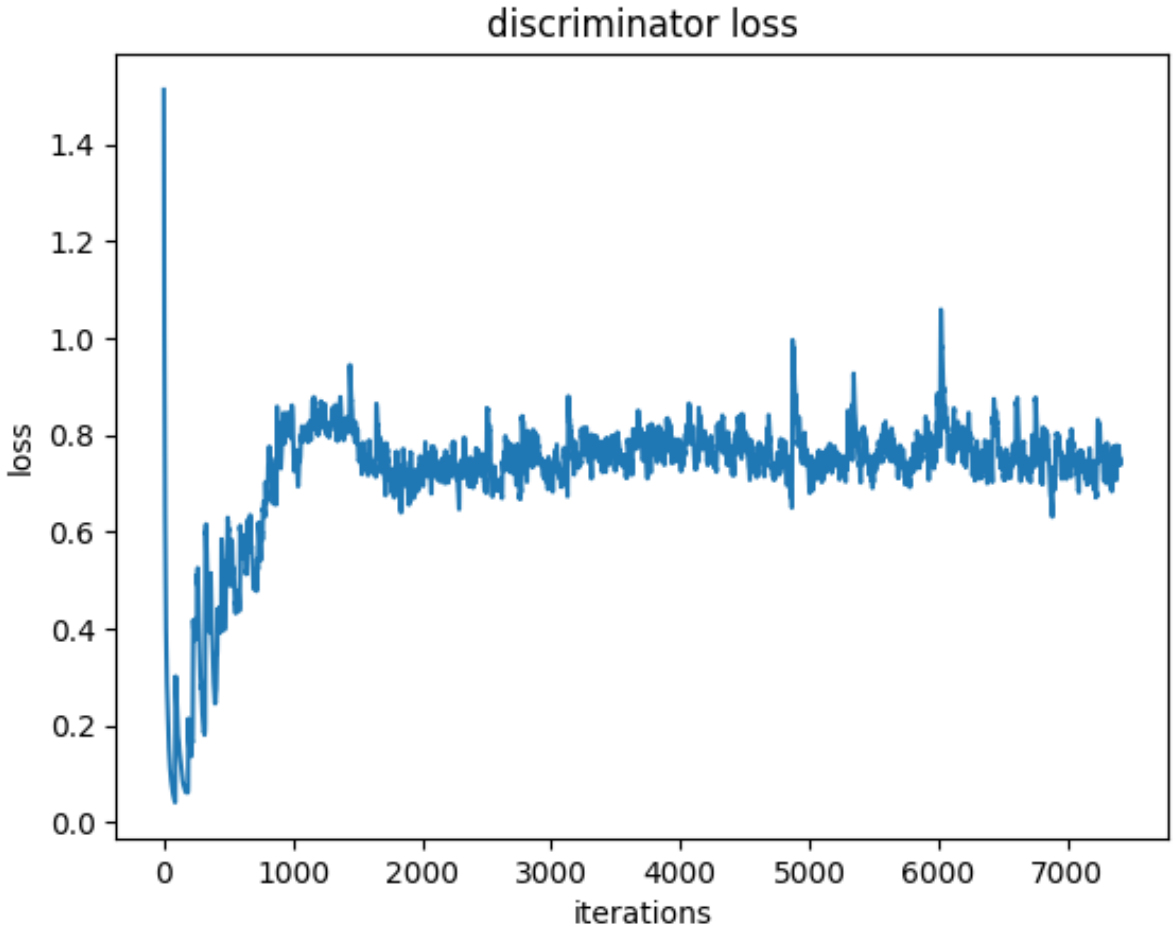

0
50
100
150
200
250
0
50
100
150
200
250

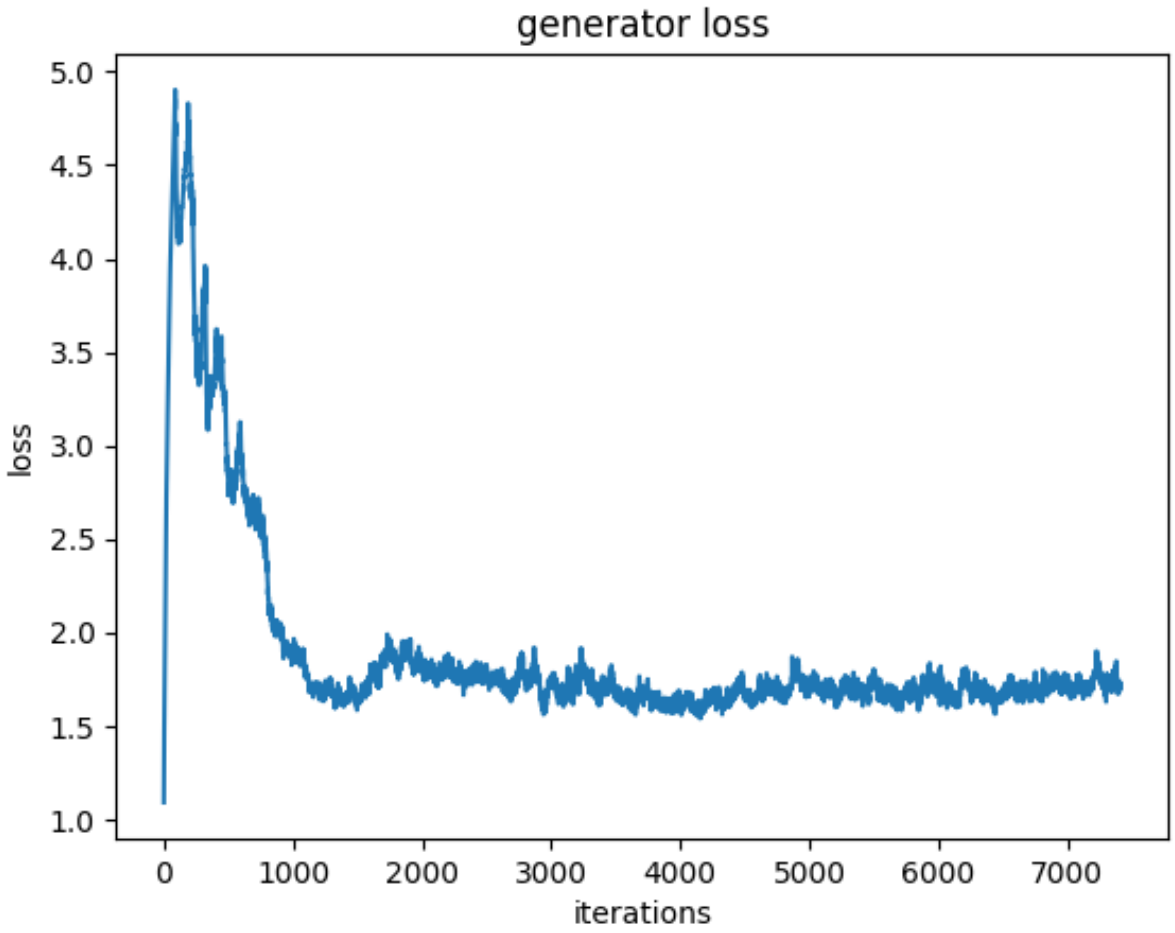

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
1000
2000
3000
4000
5000
6000



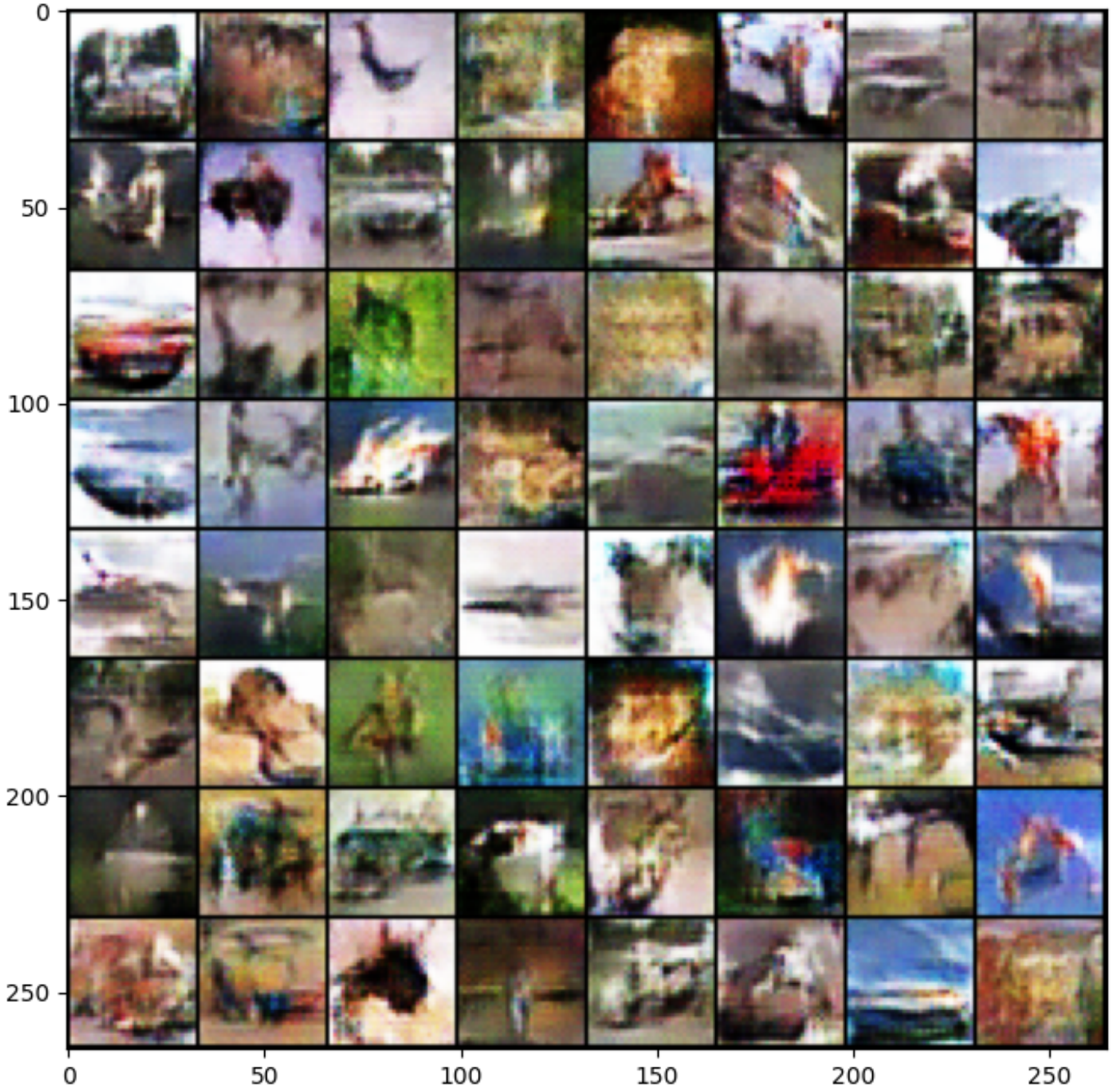

```
Iteration 6700/9750: dis loss = 0.8214, gen loss = 0.7980
Iteration 6800/9750: dis loss = 0.6863, gen loss = 1.5062
Iteration 6900/9750: dis loss = 0.8197, gen loss = 0.9231
Iteration 7000/9750: dis loss = 0.4960, gen loss = 1.8128
Iteration 7100/9750: dis loss = 0.6214, gen loss = 1.7561
Iteration 7200/9750: dis loss = 0.7961, gen loss = 2.1440
Iteration 7300/9750: dis loss = 0.6634, gen loss = 1.7471
Iteration 7400/9750: dis loss = 0.5494, gen loss = 2.3284
```

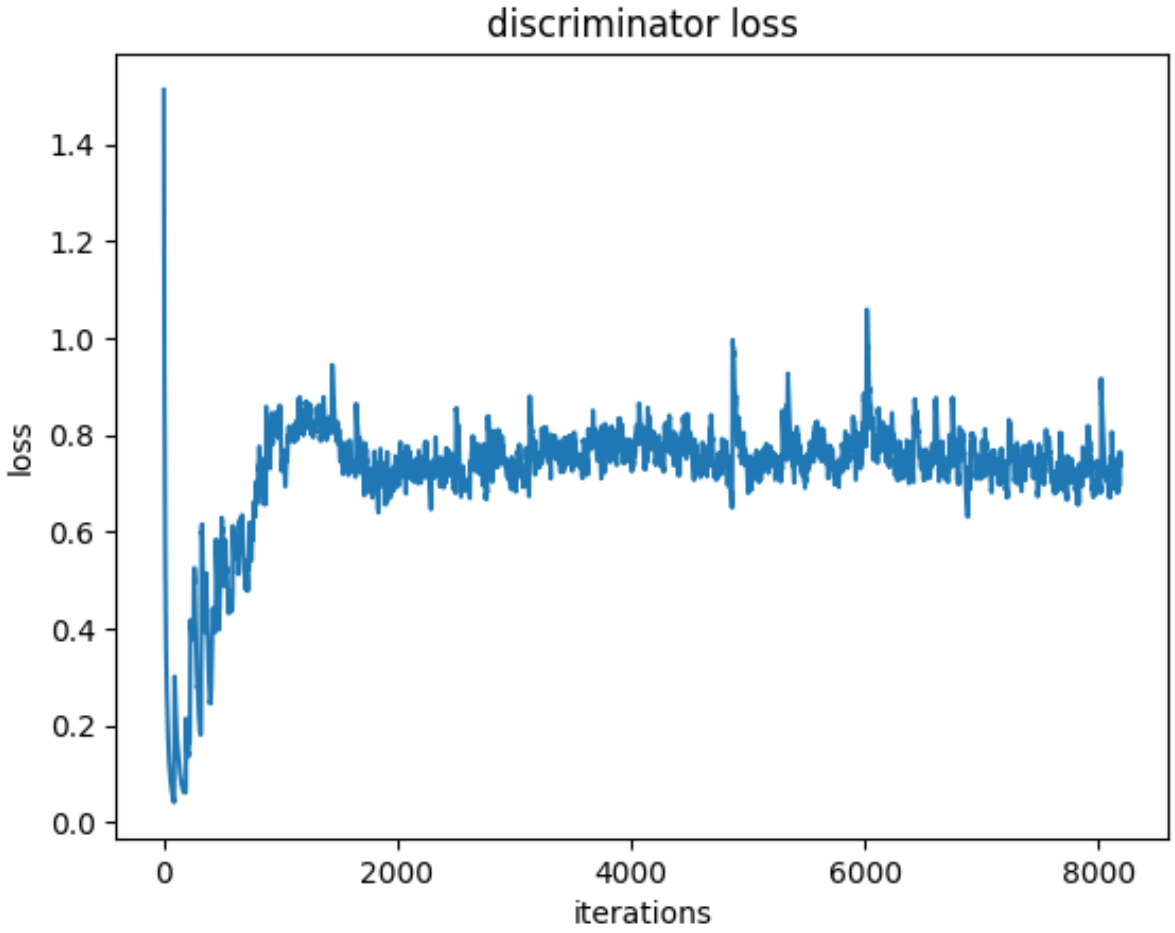

0
50
100
150
200
250
0
50
100
150
200
250

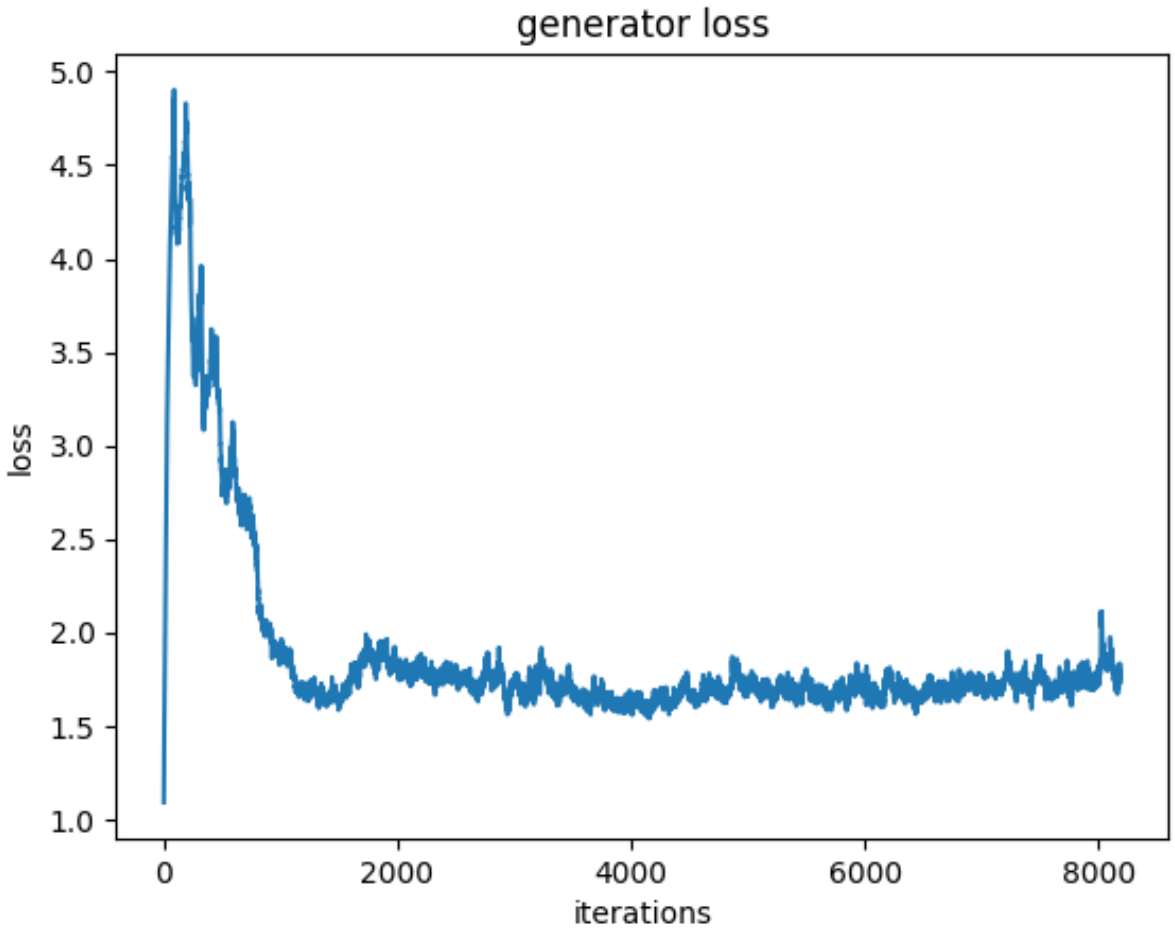

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
1000
2000
3000
4000
5000
6000
7000



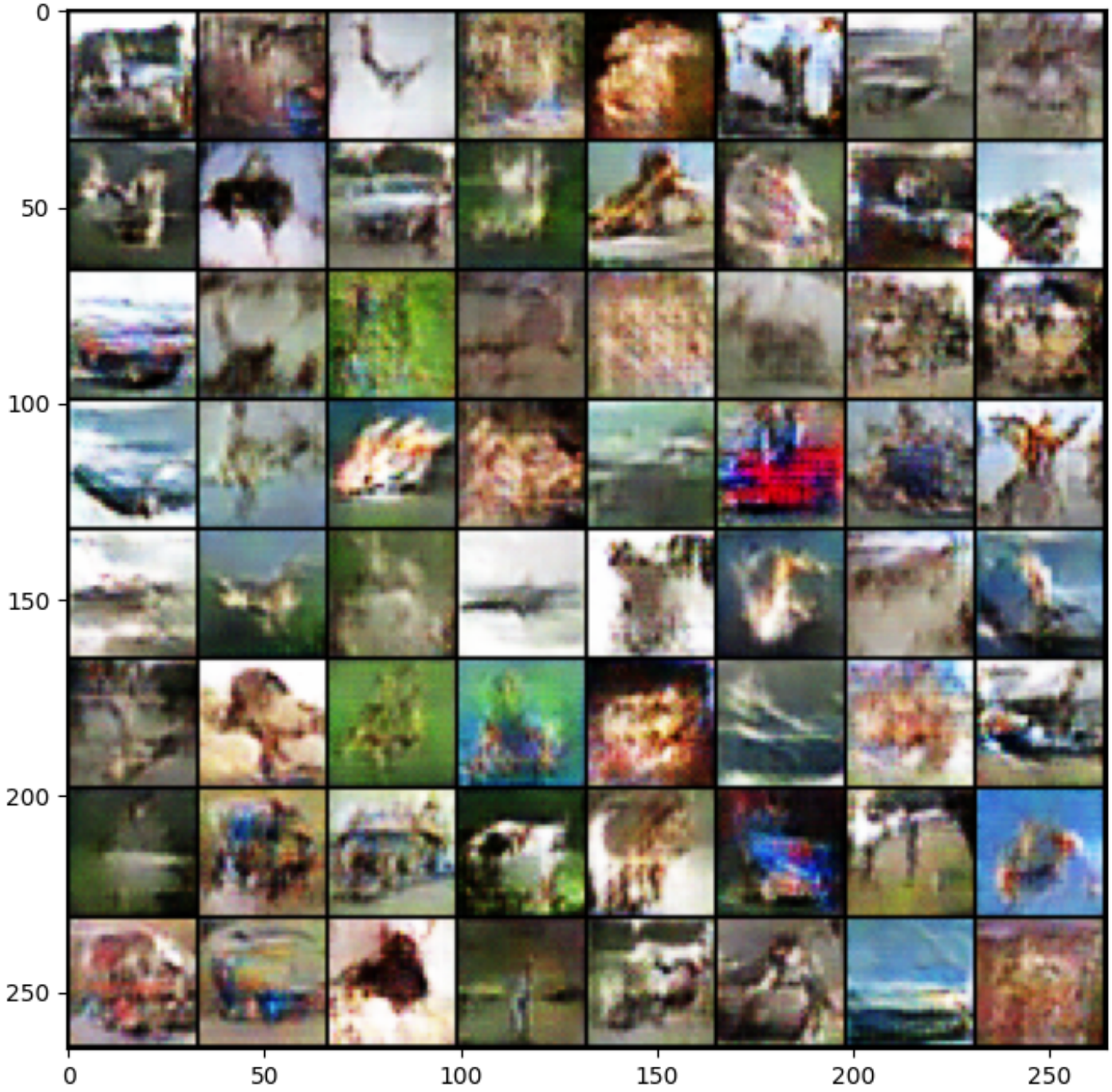

```
Iteration 7500/9750: dis loss = 0.6634, gen loss = 2.5993
Iteration 7600/9750: dis loss = 0.5296, gen loss = 1.0963
Iteration 7700/9750: dis loss = 0.7229, gen loss = 2.3367
Iteration 7800/9750: dis loss = 0.5080, gen loss = 1.4877
Iteration 7900/9750: dis loss = 0.8000, gen loss = 0.8097
Iteration 8000/9750: dis loss = 0.6972, gen loss = 1.2189
Iteration 8100/9750: dis loss = 0.7354, gen loss = 1.2541
```

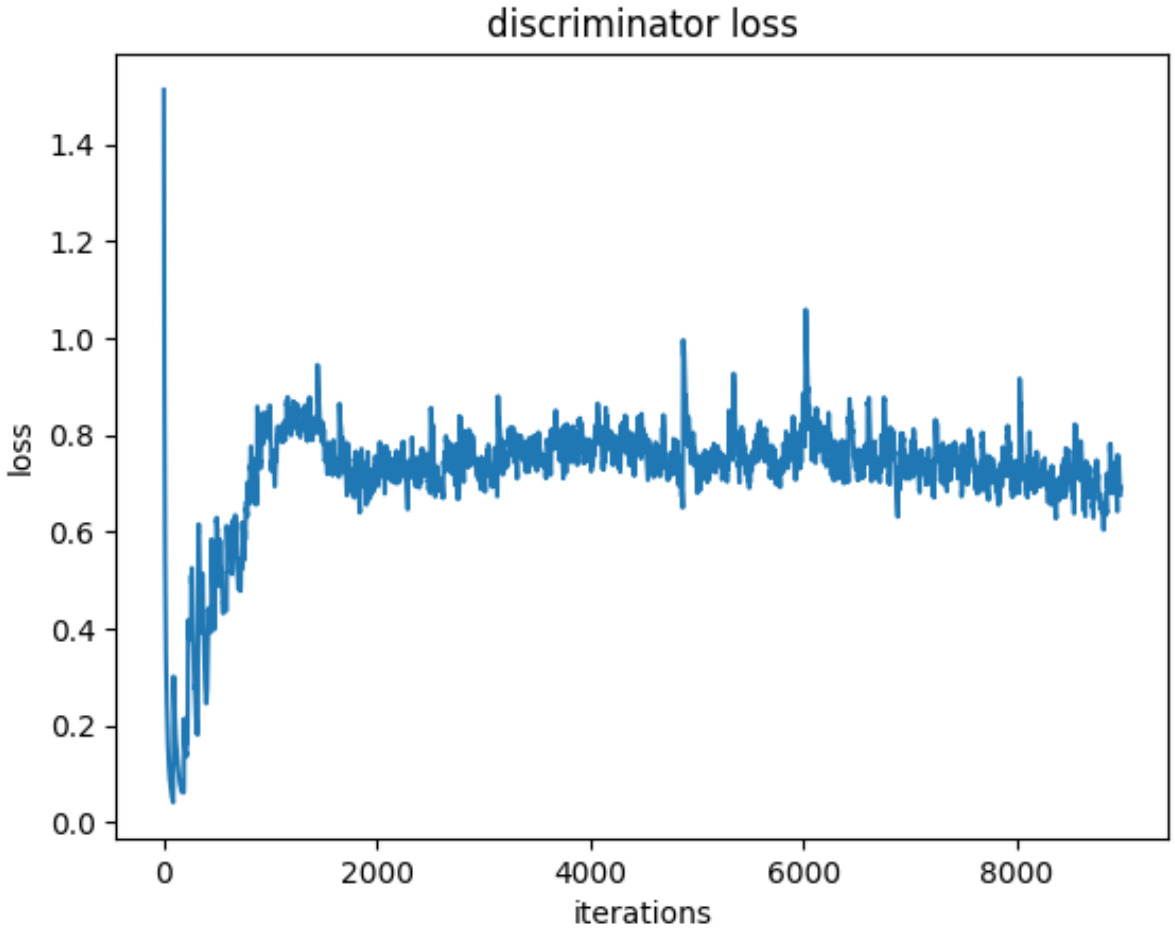

0
50
100
150
200
250
0
50
100
150
200
250

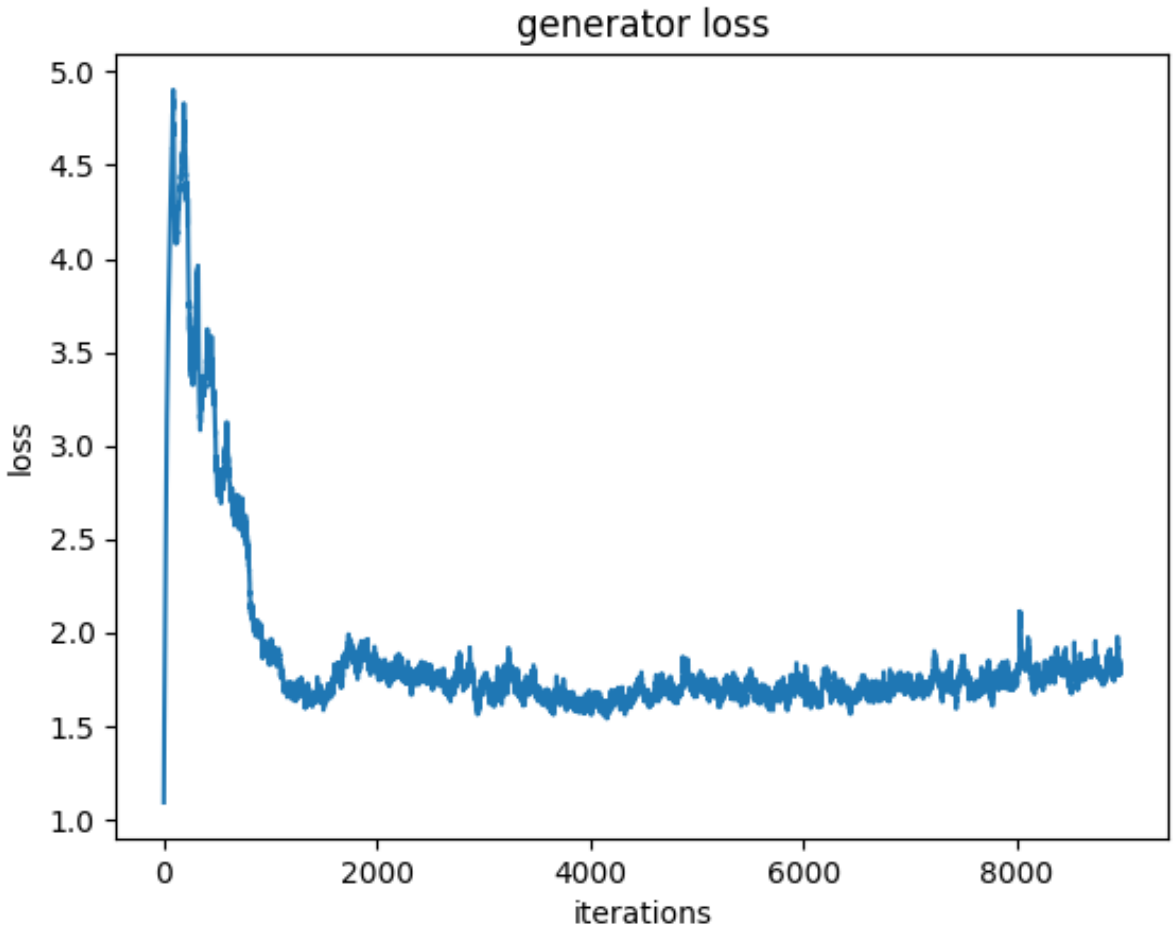

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
2000
4000
6000
8000




```
Iteration 8200/9750: dis loss = 0.8004, gen loss = 3.1011
Iteration 8300/9750: dis loss = 0.6565, gen loss = 1.6458
Iteration 8400/9750: dis loss = 1.0517, gen loss = 2.7182
Iteration 8500/9750: dis loss = 0.5644, gen loss = 1.8859
Iteration 8600/9750: dis loss = 0.7522, gen loss = 1.8172
Iteration 8700/9750: dis loss = 0.6867, gen loss = 2.1600
Iteration 8800/9750: dis loss = 0.6216, gen loss = 1.4626
Iteration 8900/9750: dis loss = 0.6458, gen loss = 1.4437
```

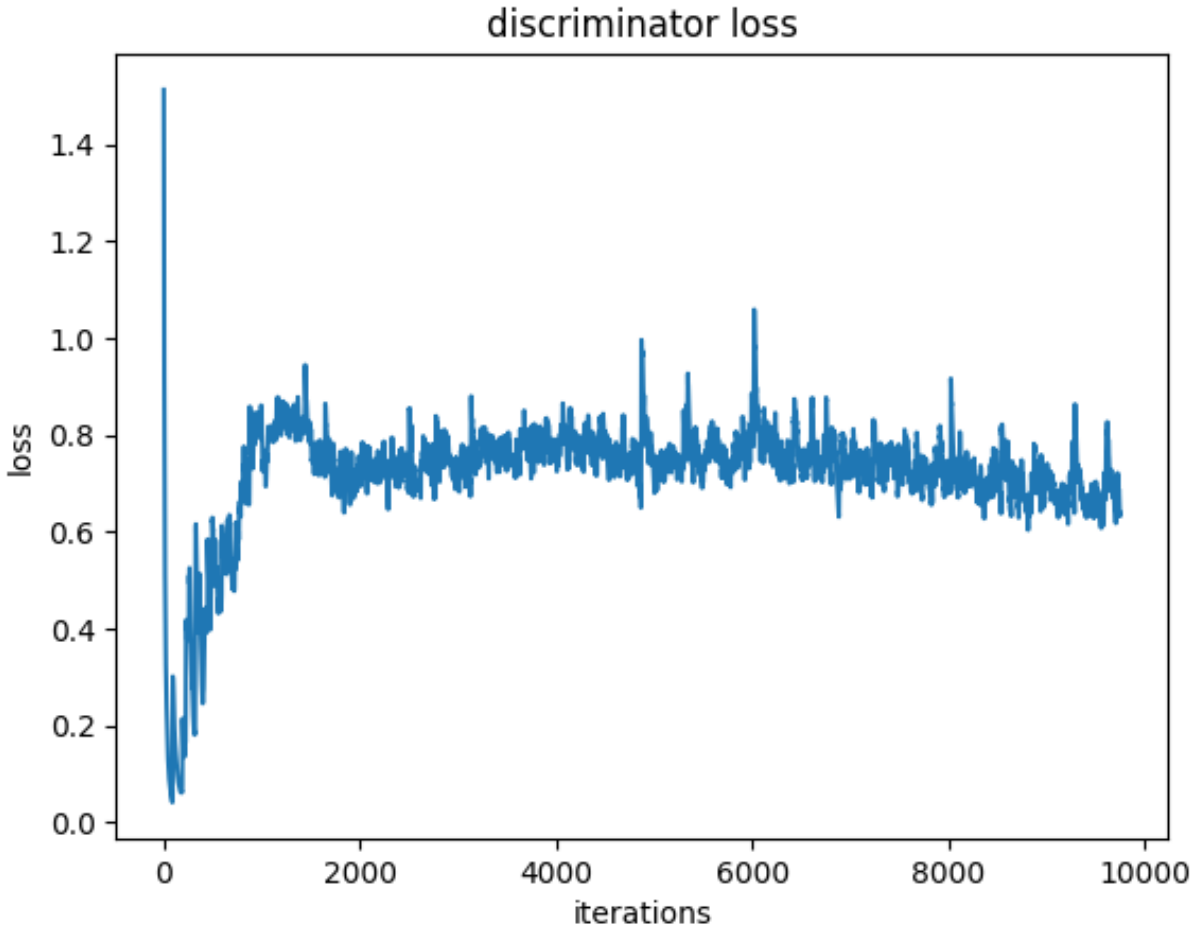

0
50
100
150
200
250
0
50
100
150
200
250

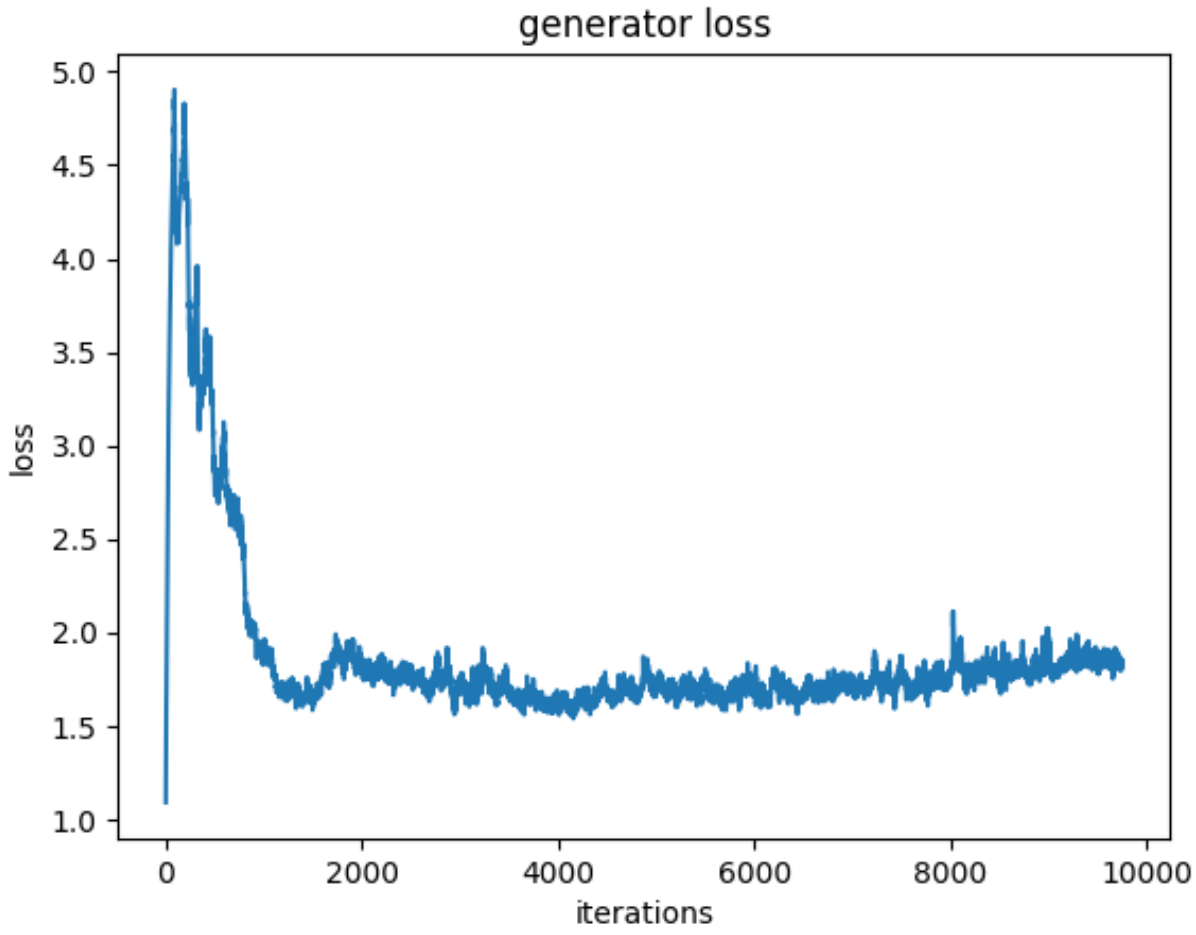

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
2000
4000
6000
8000




```
Iteration 9000/9750: dis loss = 0.7032, gen loss = 1.9108
Iteration 9100/9750: dis loss = 0.7988, gen loss = 1.6201
Iteration 9200/9750: dis loss = 0.5078, gen loss = 2.2565
Iteration 9300/9750: dis loss = 0.8670, gen loss = 1.9296
Iteration 9400/9750: dis loss = 0.4547, gen loss = 1.8954
Iteration 9500/9750: dis loss = 0.9004, gen loss = 0.8392
Iteration 9600/9750: dis loss = 0.6565, gen loss = 3.4776
Iteration 9700/9750: dis loss = 0.6194, gen loss = 2.2041
```


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
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 there were a BN layer between dis_conv1 and dis_lrelu1, it would normalize the activations of the LeakyReLU non-linearity. This would have serveral problems:

- The BN could reduce the expressive power of the model. Since LeakyReLU is a non-linear activation function, it can introduce asymmetries in the distribution of the activations. Normalizing these activations could eliminate these asymmetries, and potentially reduce the model's ability to learn complex representations.
- The BN could introduce dependencies between the examples in the batch, which would reduce the diversity of the samples used to train the discriminator. Since the discriminator needs to be able to distinguish between real and fake data, it should learn from diverse examples. But if the BN layer introduces dependencies between examples, the diversity of the training data will be reduced and it's harder for the discriminator to learn to distinguish between real and fake data.

Thus, by omitting the BN layer between dis_conv1 and dis_lrelu1, the model is able to learn more expressive representations and can learn from more diverse examples. The model is able to capture the asymmetries in the distribution of the activations, and also learn from more diverse examples with less dependencies. Our results also show that omitting the BN layer works well in practice.

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)

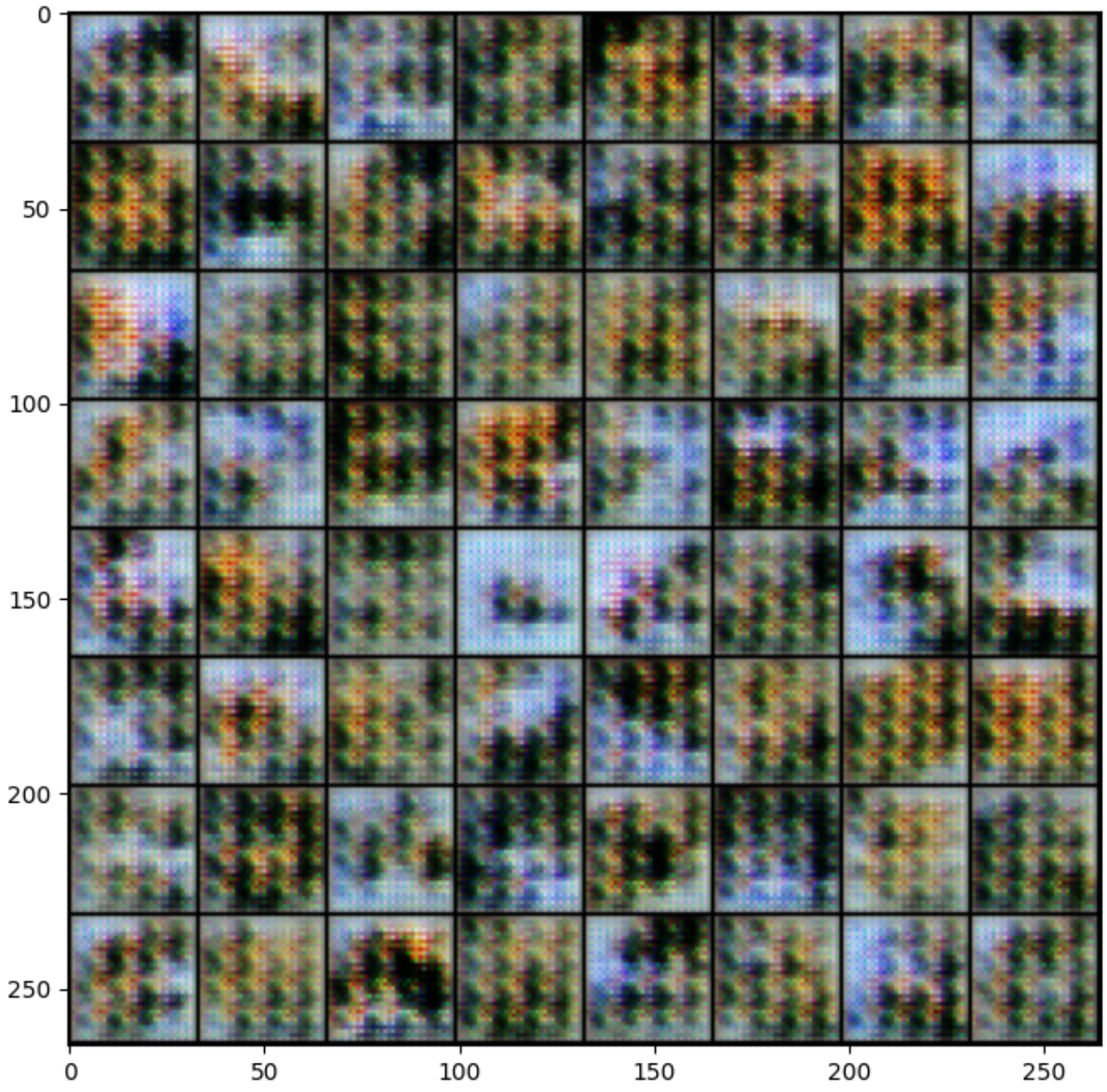
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

 ################################################################################
        # Prob 2-3:
#
```

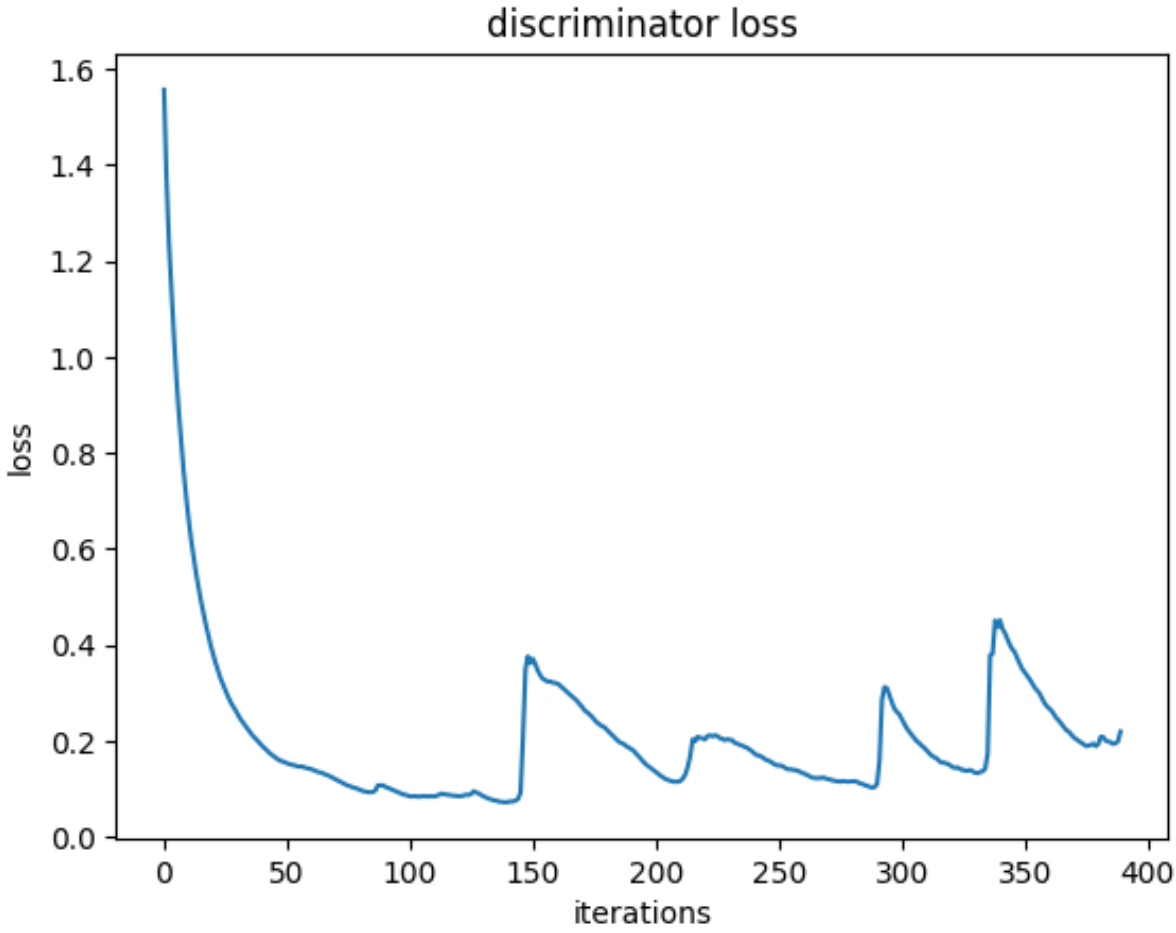
```
        # adding spectral norm to the discriminator
#

 ##############################################################################
        self.spectral_norm = nn.utils.spectral_norm
        self.downsample = nn.Sequential(
            self.spectral_norm(
                nn.Conv2d(
            in_channels=3, out_channels=32, kernel_size=4, stride=2, padding=1,
bias=True)),
            nn.LeakyReLU(),
            self.spectral_norm(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2,
padding=1, bias=True)),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            self.spectral_norm(
                nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2,
padding=1, bias=True)),
            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
```

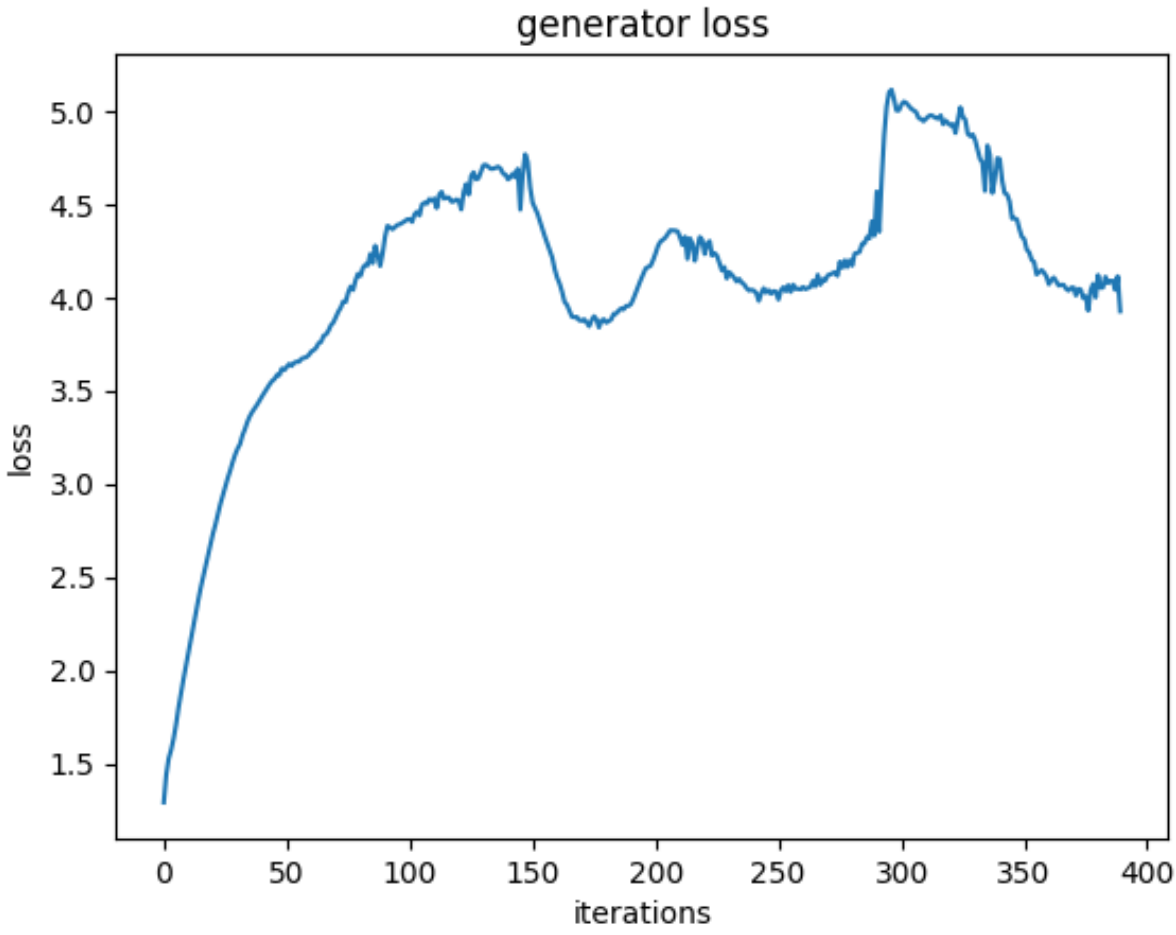
After adding the spectral norm to the discriminator, redo the training block below to see the effects.

```
set_seed(42)

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

```
Start training ...
Iteration 100/9750: dis loss = 0.0539, gen loss = 4.6053
Iteration 200/9750: dis loss = 0.0686, gen loss = 4.7702
Iteration 300/9750: dis loss = 0.1615, gen loss = 5.0190
```

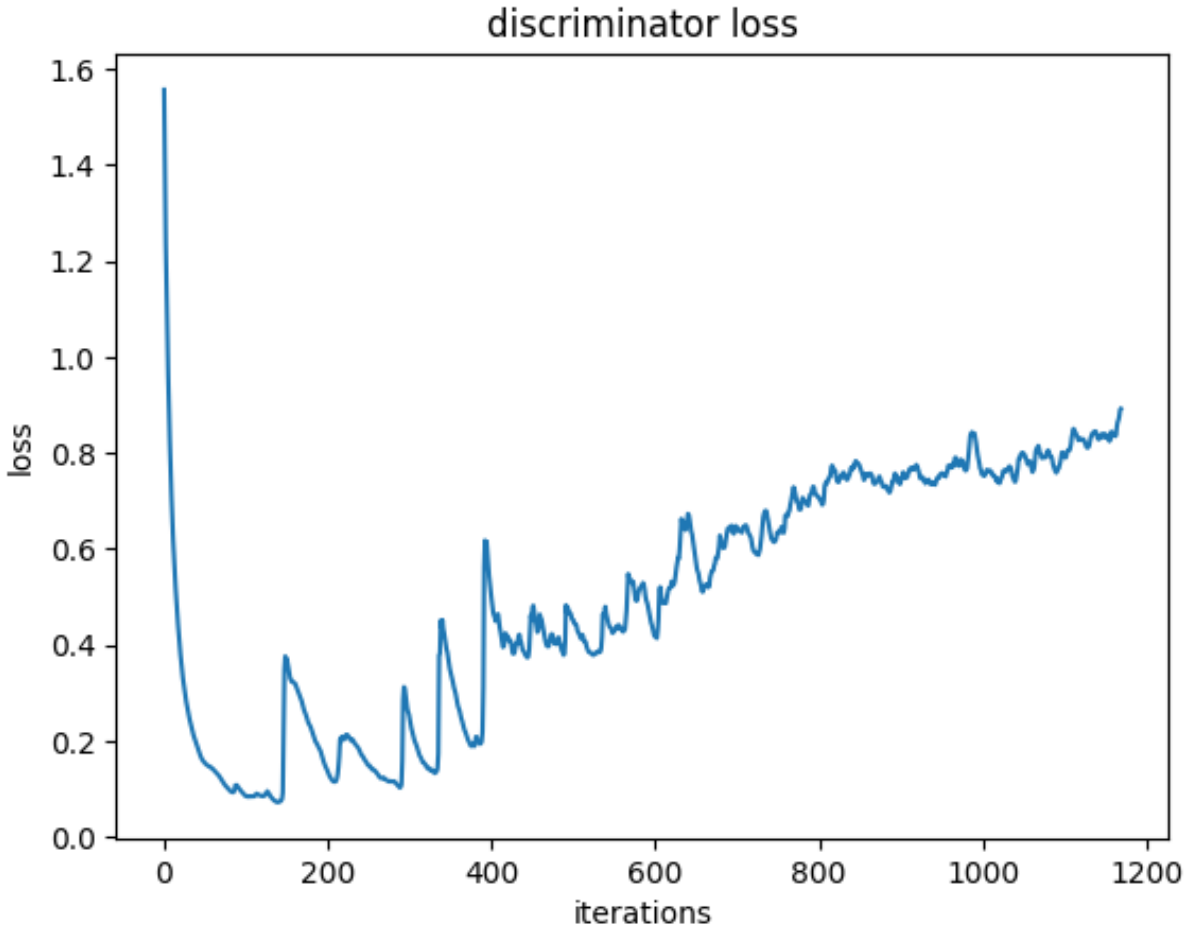

0
50
100
150
200
250
0
50
100
150
200
250

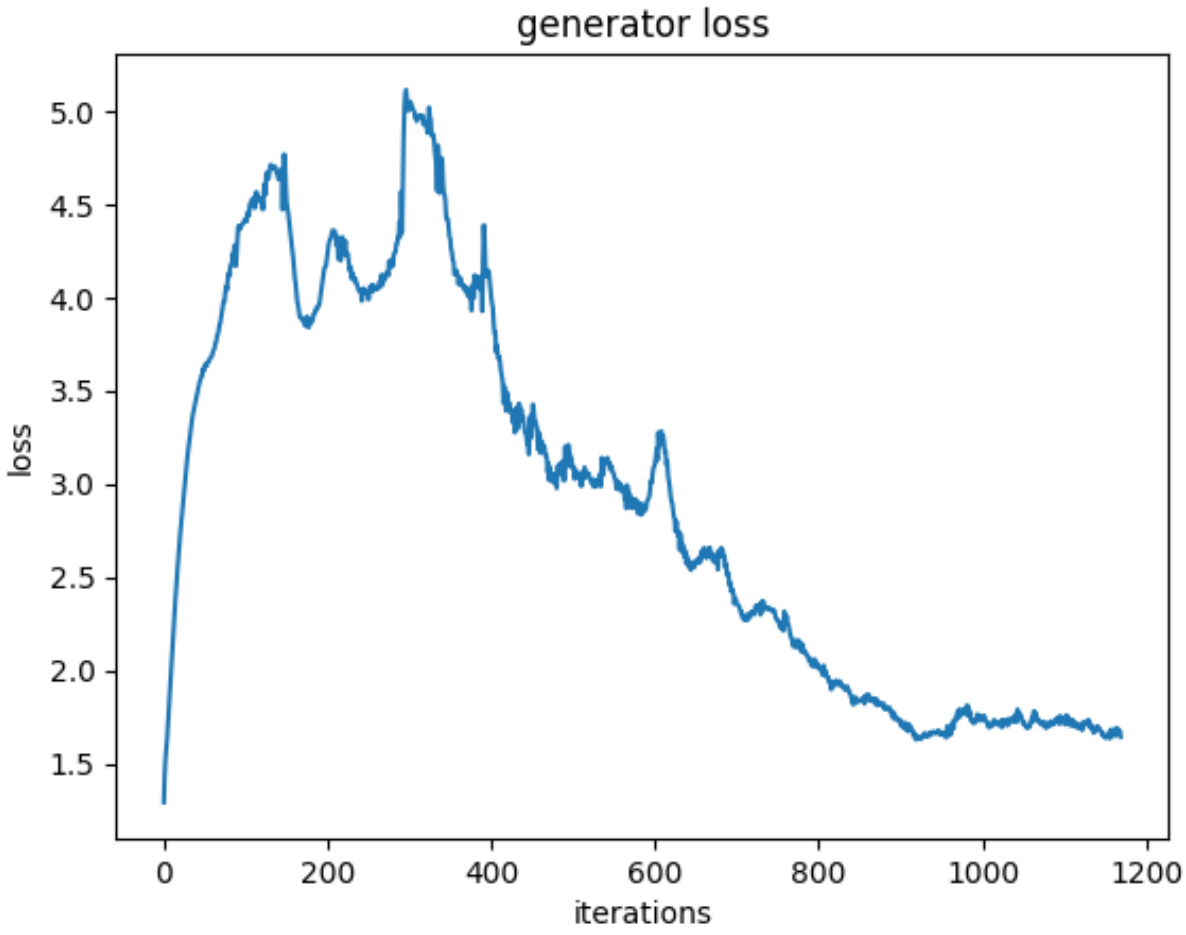

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
50
100
150
200
250
300
350
400



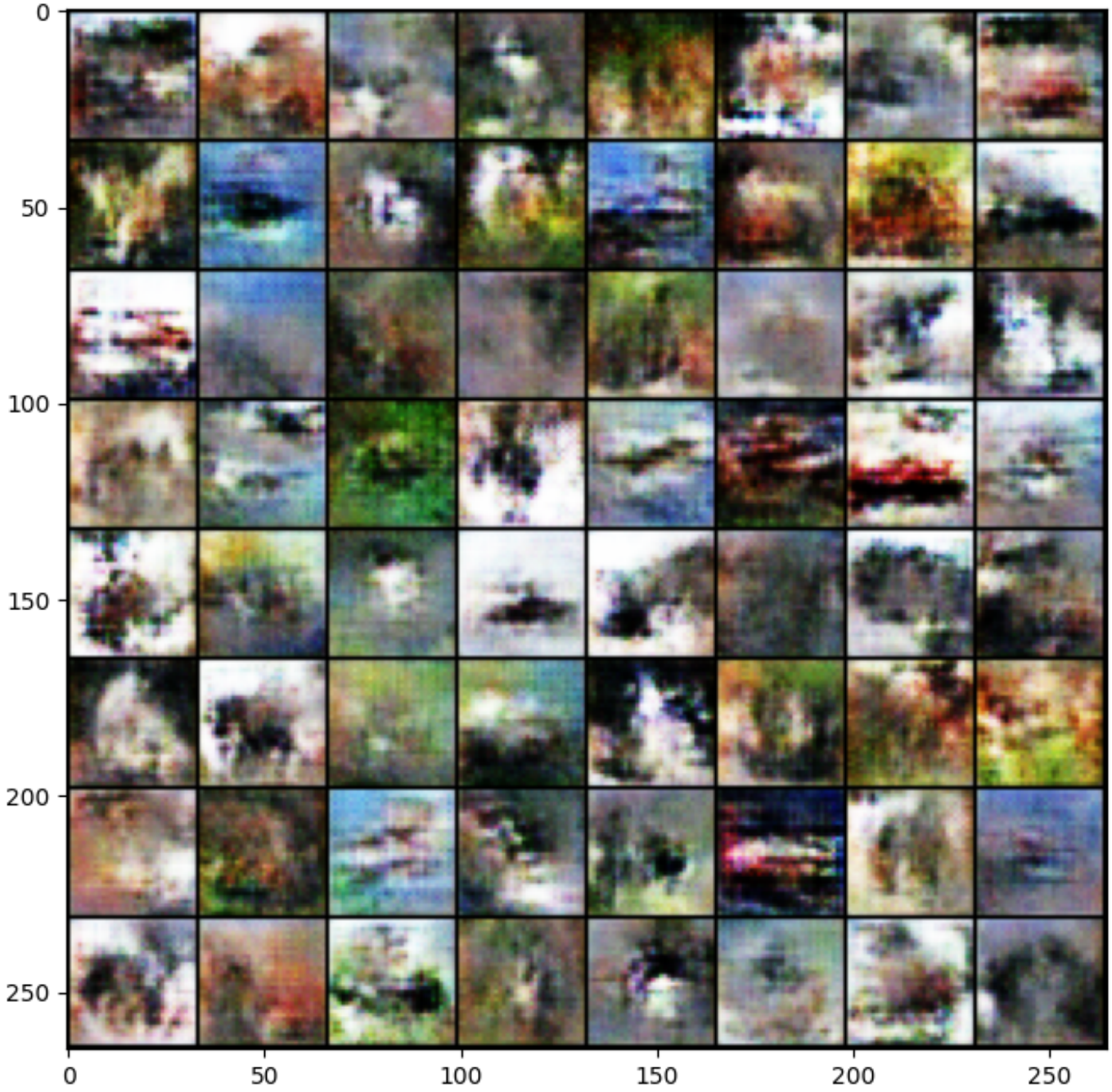

```
Iteration 400/9750: dis loss = 0.1894, gen loss = 3.1315
Iteration 500/9750: dis loss = 0.3874, gen loss = 2.0844
Iteration 600/9750: dis loss = 0.2390, gen loss = 3.2466
Iteration 700/9750: dis loss = 0.6399, gen loss = 2.1858
Iteration 800/9750: dis loss = 0.6522, gen loss = 2.3091
Iteration 900/9750: dis loss = 0.6852, gen loss = 1.3839
Iteration 1000/9750: dis loss = 0.6058, gen loss = 2.1169
Iteration 1100/9750: dis loss = 0.6436, gen loss = 1.4417
```

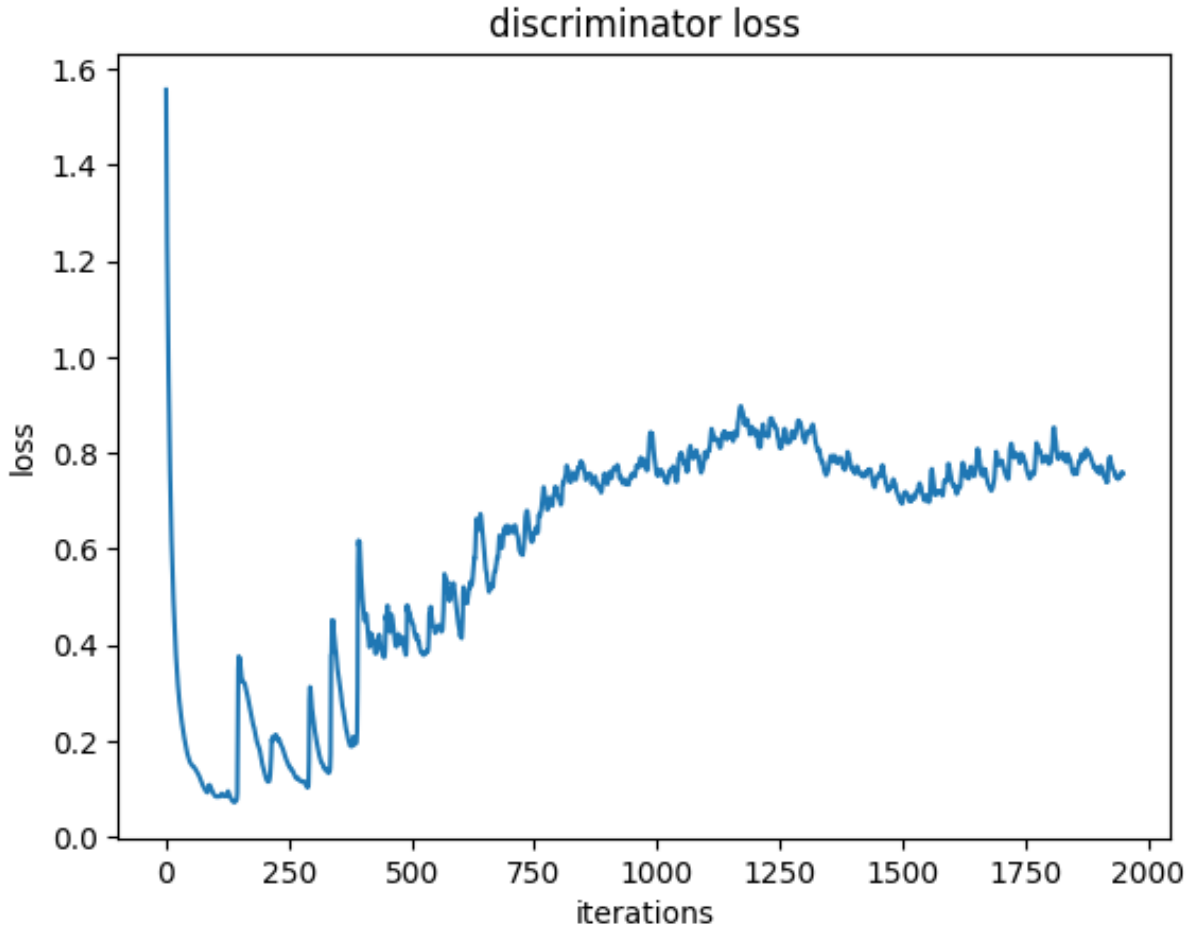

0
50
100
150
200
250
0
50
100
150
200
250

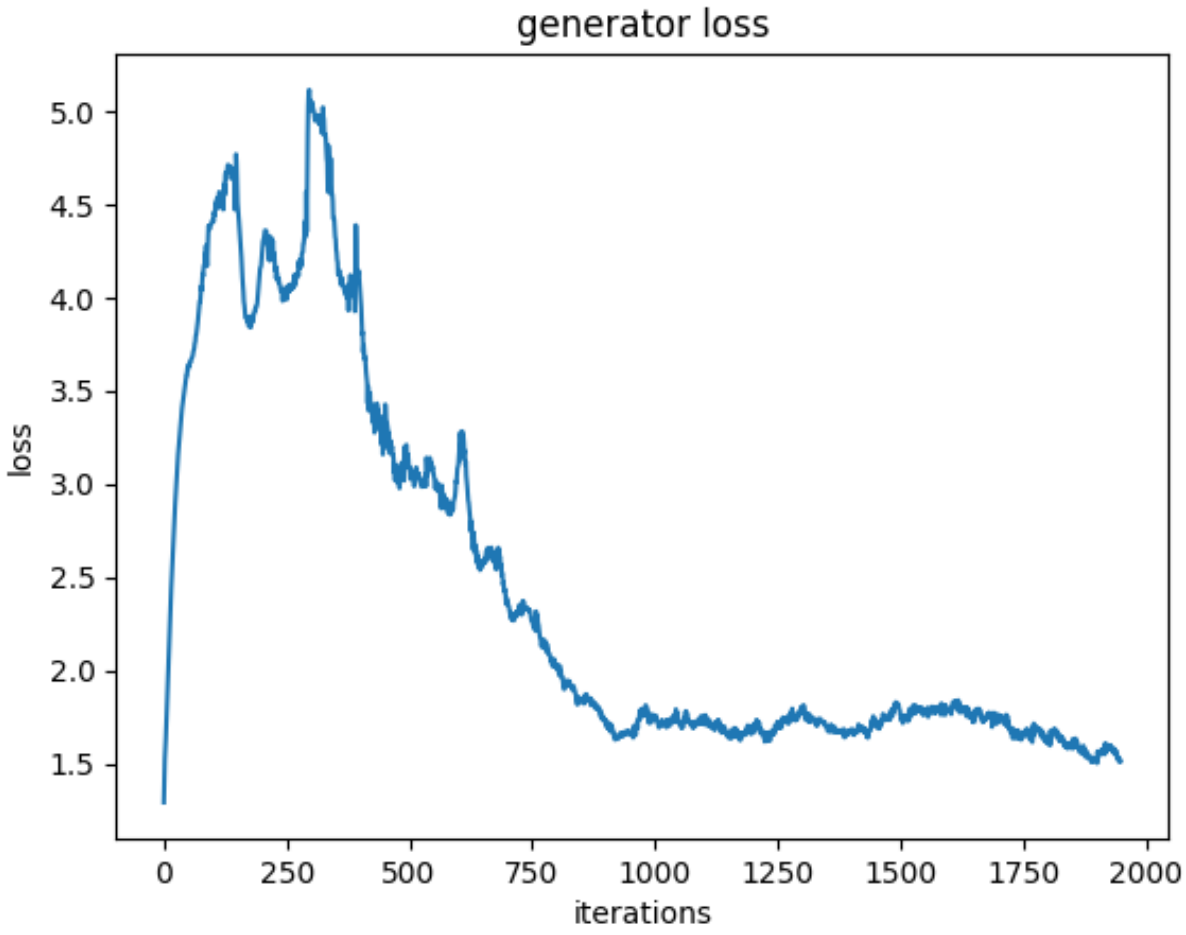

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
200
400
600
800
1000
1200



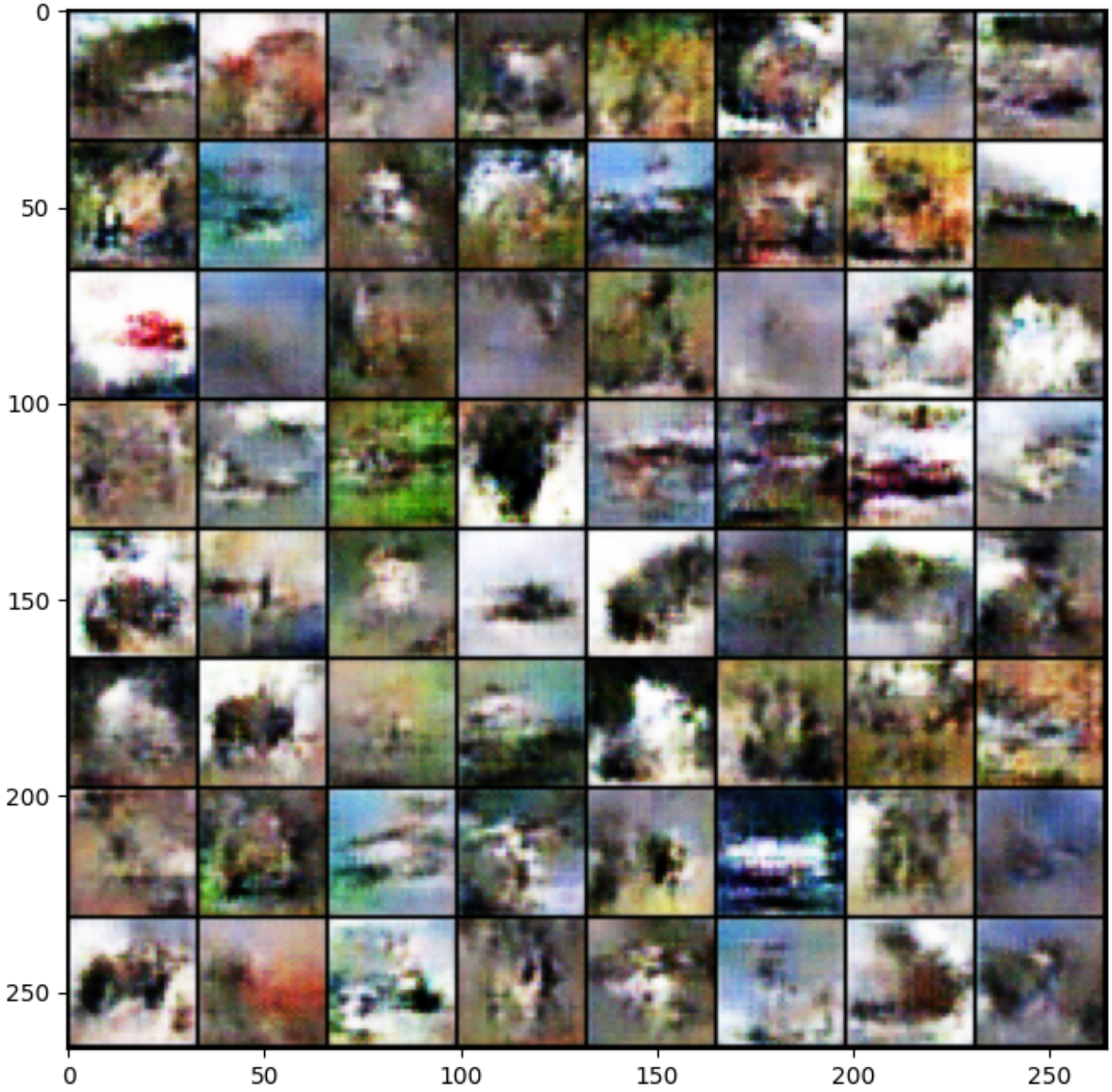

```
Iteration 1200/9750: dis loss = 0.7088, gen loss = 1.4780
Iteration 1300/9750: dis loss = 0.8110, gen loss = 1.5136
Iteration 1400/9750: dis loss = 0.7511, gen loss = 1.8280
Iteration 1500/9750: dis loss = 0.6731, gen loss = 1.7332
Iteration 1600/9750: dis loss = 0.6337, gen loss = 1.8145
Iteration 1700/9750: dis loss = 0.7944, gen loss = 1.3845
Iteration 1800/9750: dis loss = 0.8637, gen loss = 1.2021
Iteration 1900/9750: dis loss = 0.7057, gen loss = 1.4245
```

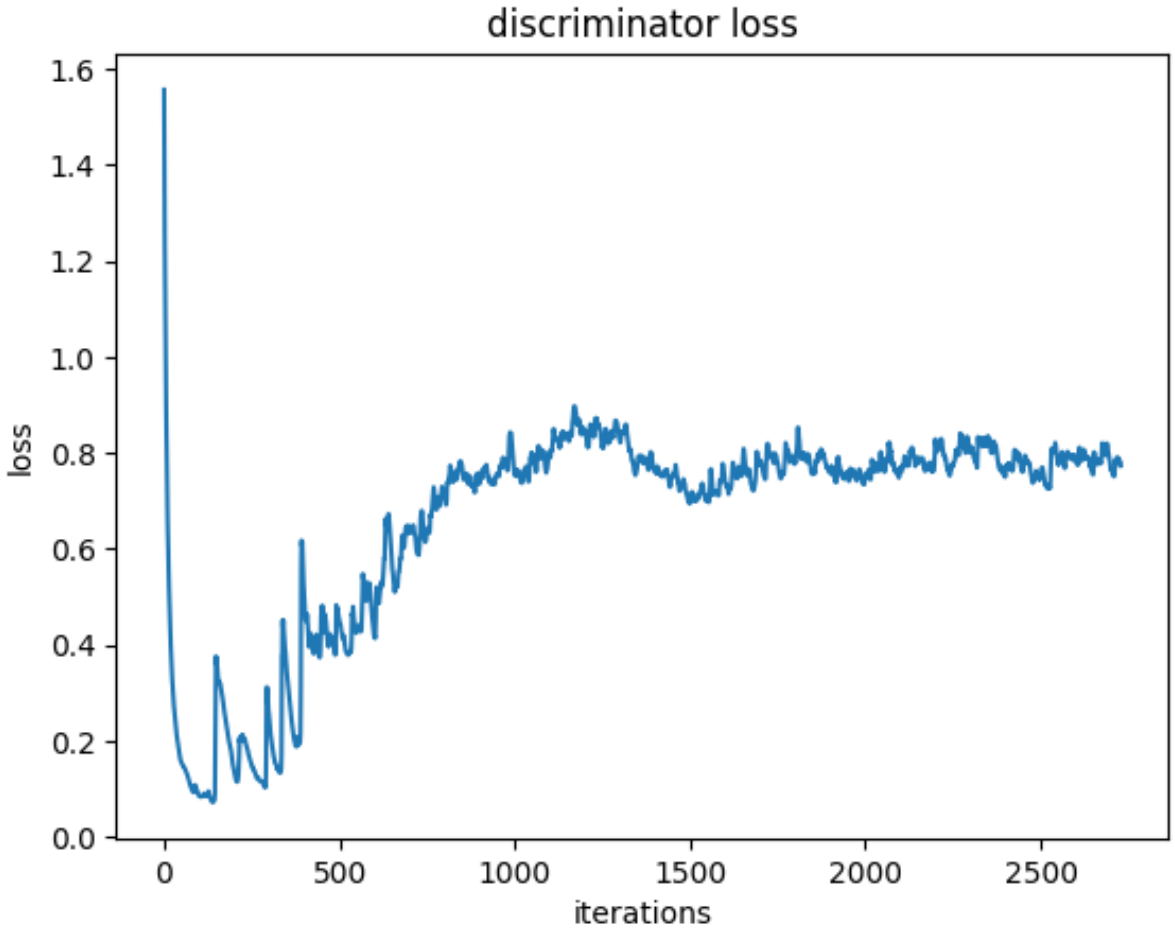

0
50
100
150
200
250
0
50
100
150
200
250

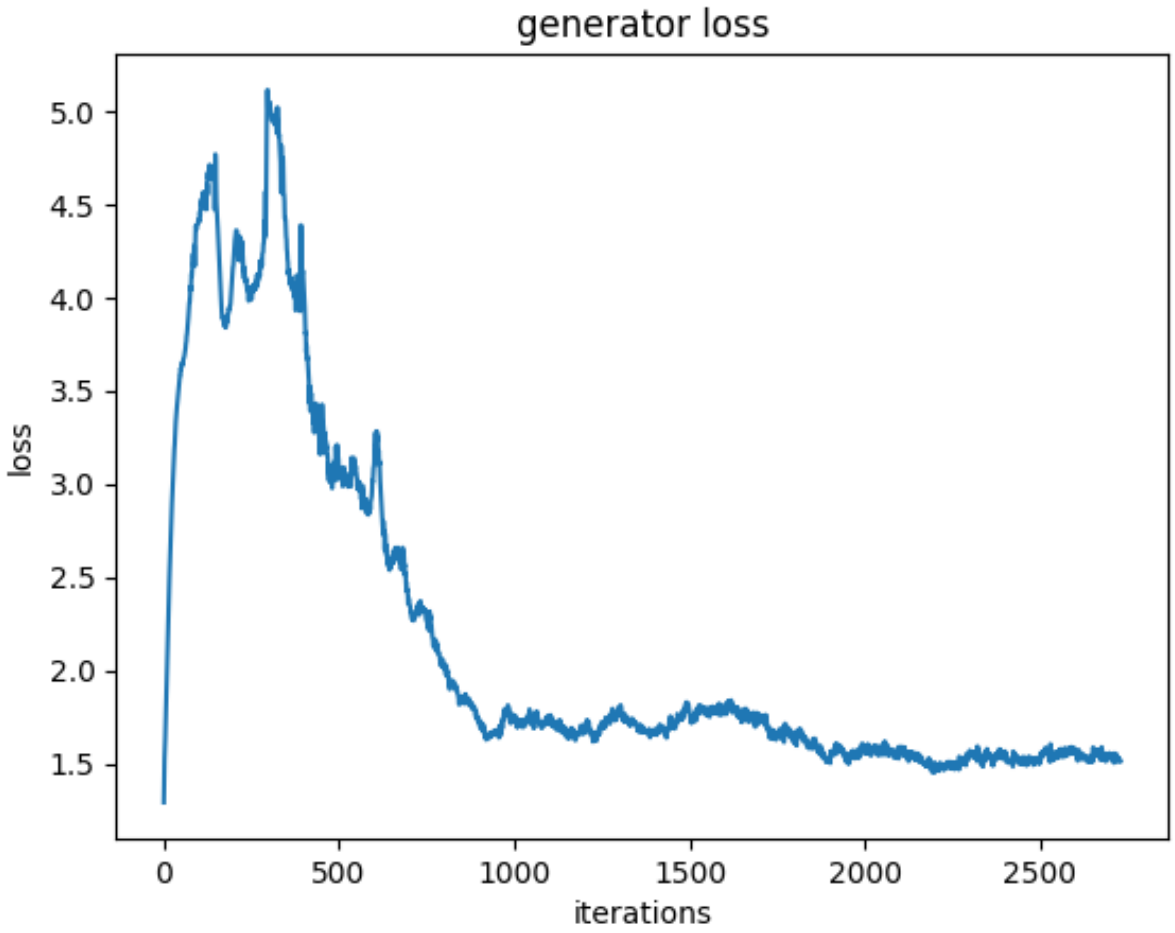

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
250
500
750
1000
1250
1500
1750
2000



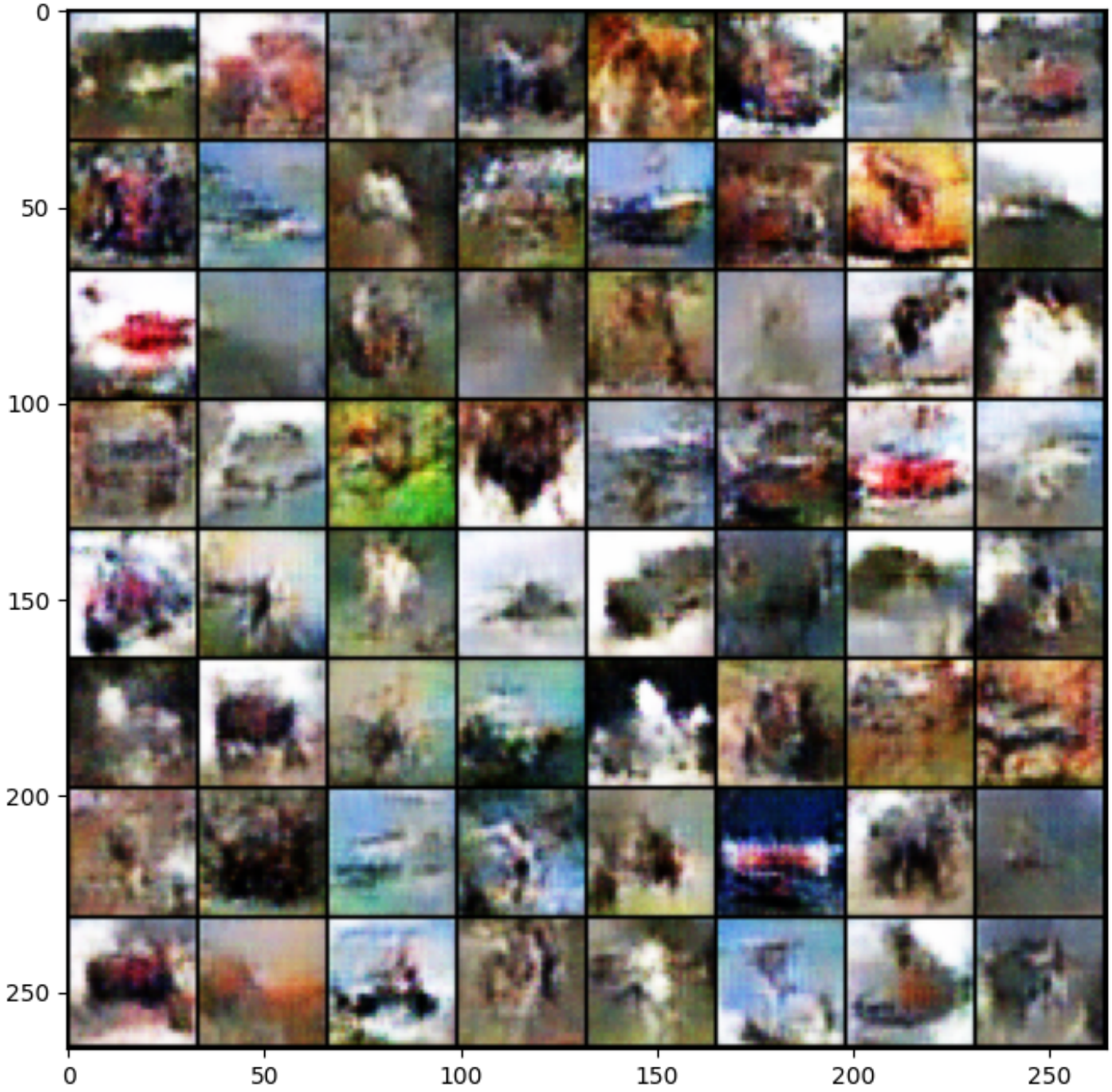

```
Iteration 2000/9750: dis loss = 0.8401, gen loss = 0.9331
Iteration 2100/9750: dis loss = 0.8711, gen loss = 1.9648
Iteration 2200/9750: dis loss = 1.2395, gen loss = 2.3983
Iteration 2300/9750: dis loss = 0.7411, gen loss = 2.1217
Iteration 2400/9750: dis loss = 0.6651, gen loss = 2.1589
Iteration 2500/9750: dis loss = 0.9365, gen loss = 0.8671
Iteration 2600/9750: dis loss = 0.8432, gen loss = 1.5856
Iteration 2700/9750: dis loss = 0.5812, gen loss = 1.8950
```

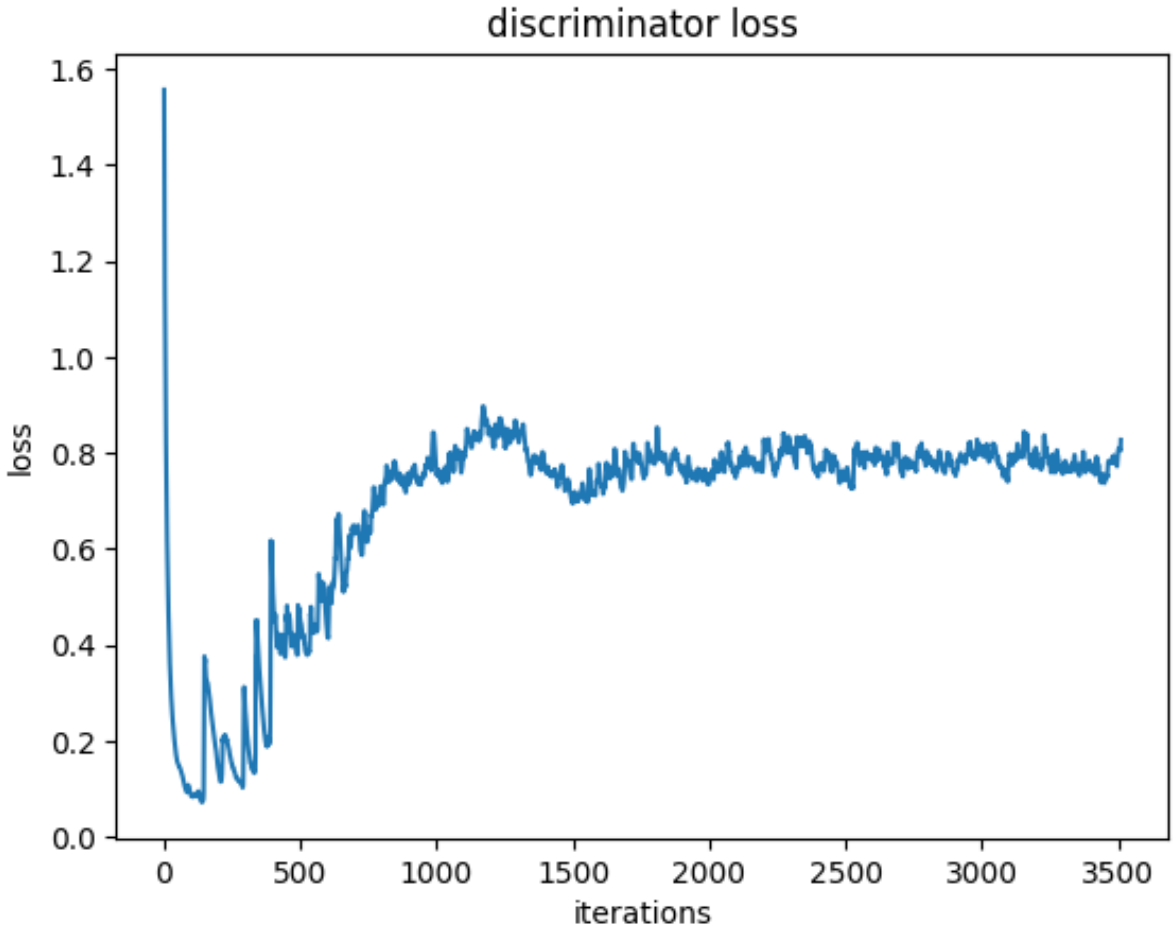

0
50
100
150
200
250
0
50
100
150
200
250

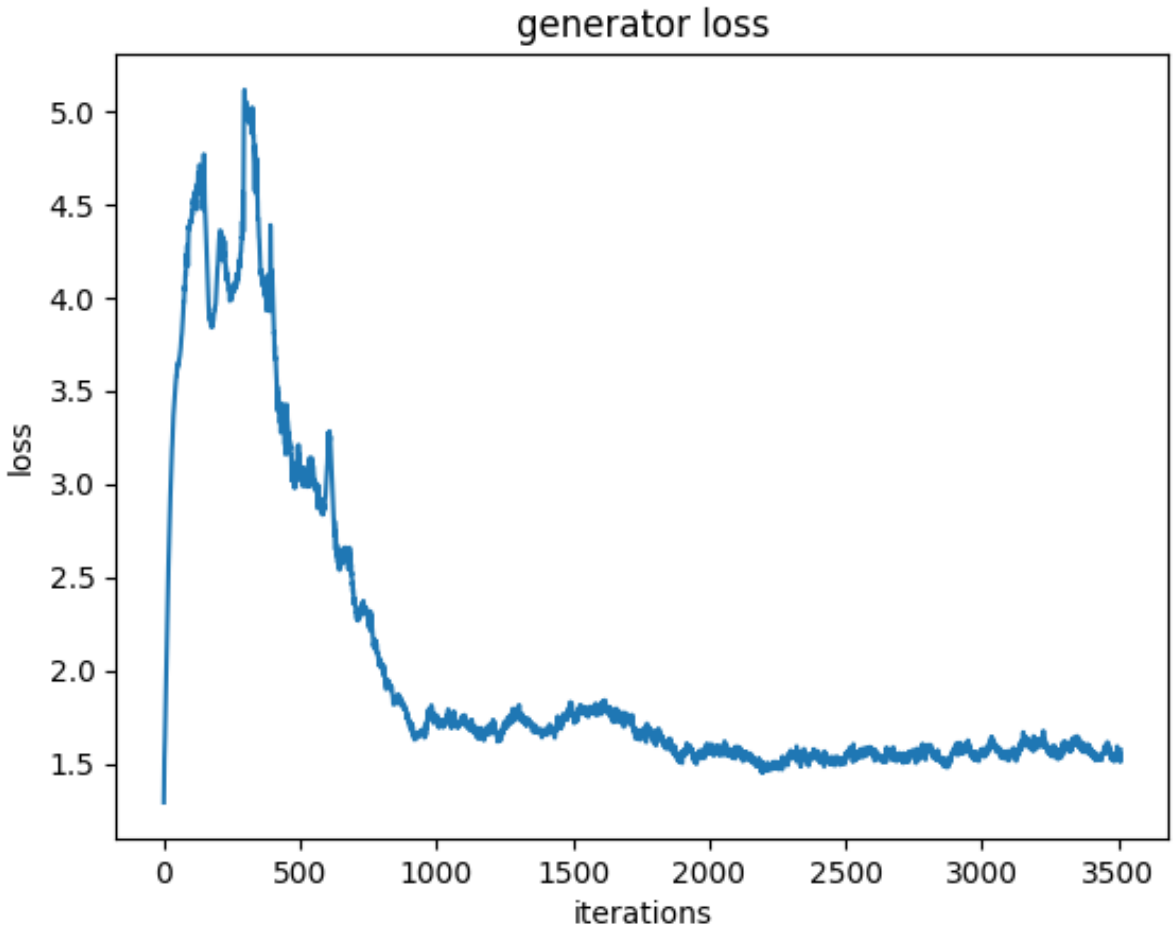

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
500
1000
1500
2000
2500



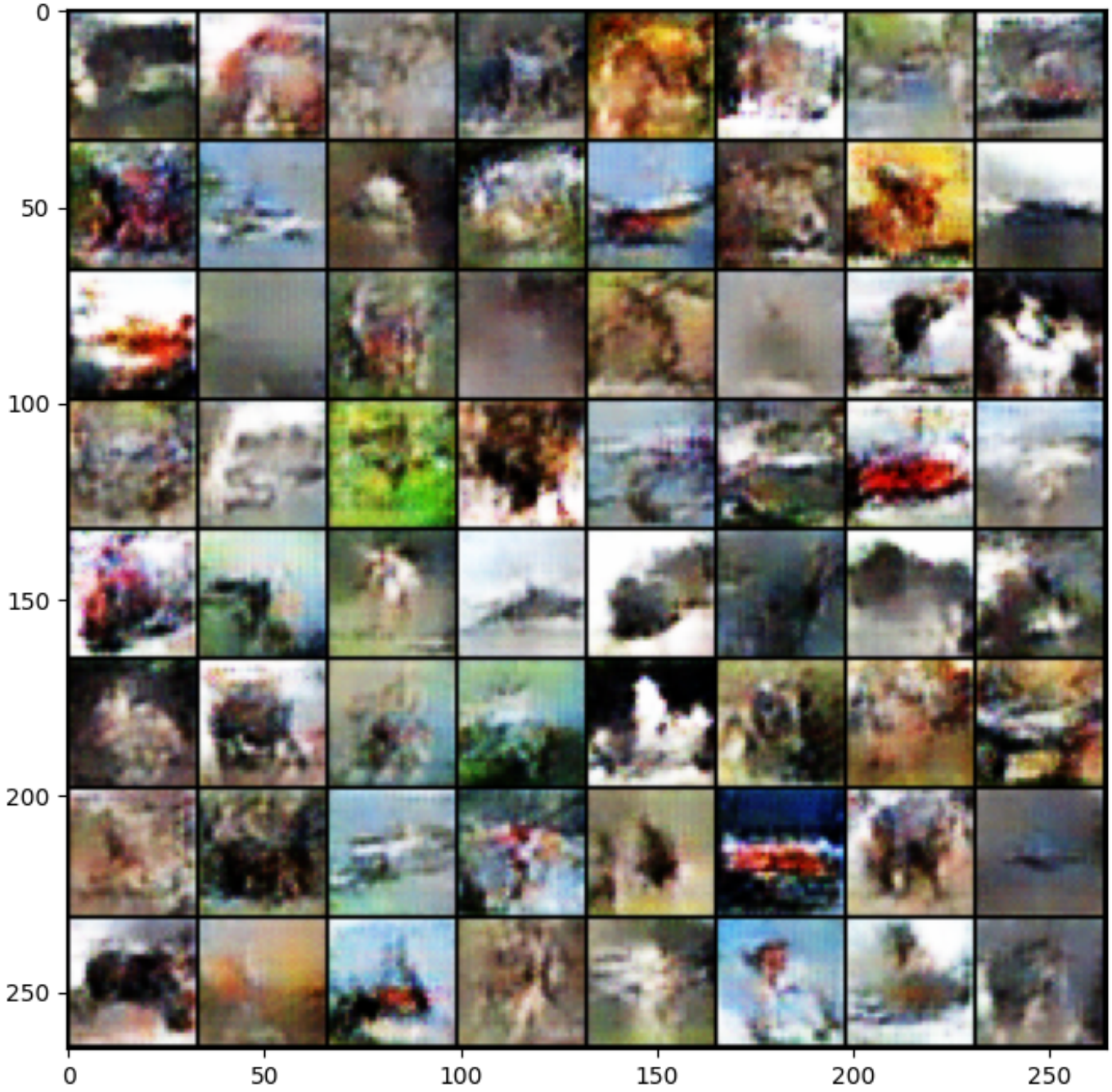

```
Iteration 2800/9750: dis loss = 0.6571, gen loss = 1.5403
Iteration 2900/9750: dis loss = 0.7316, gen loss = 1.9864
Iteration 3000/9750: dis loss = 0.7367, gen loss = 1.2610
Iteration 3100/9750: dis loss = 0.8019, gen loss = 1.7245
Iteration 3200/9750: dis loss = 0.7498, gen loss = 1.7100
Iteration 3300/9750: dis loss = 0.8361, gen loss = 1.3093
Iteration 3400/9750: dis loss = 0.7258, gen loss = 1.2754
Iteration 3500/9750: dis loss = 0.9230, gen loss = 1.9218
```

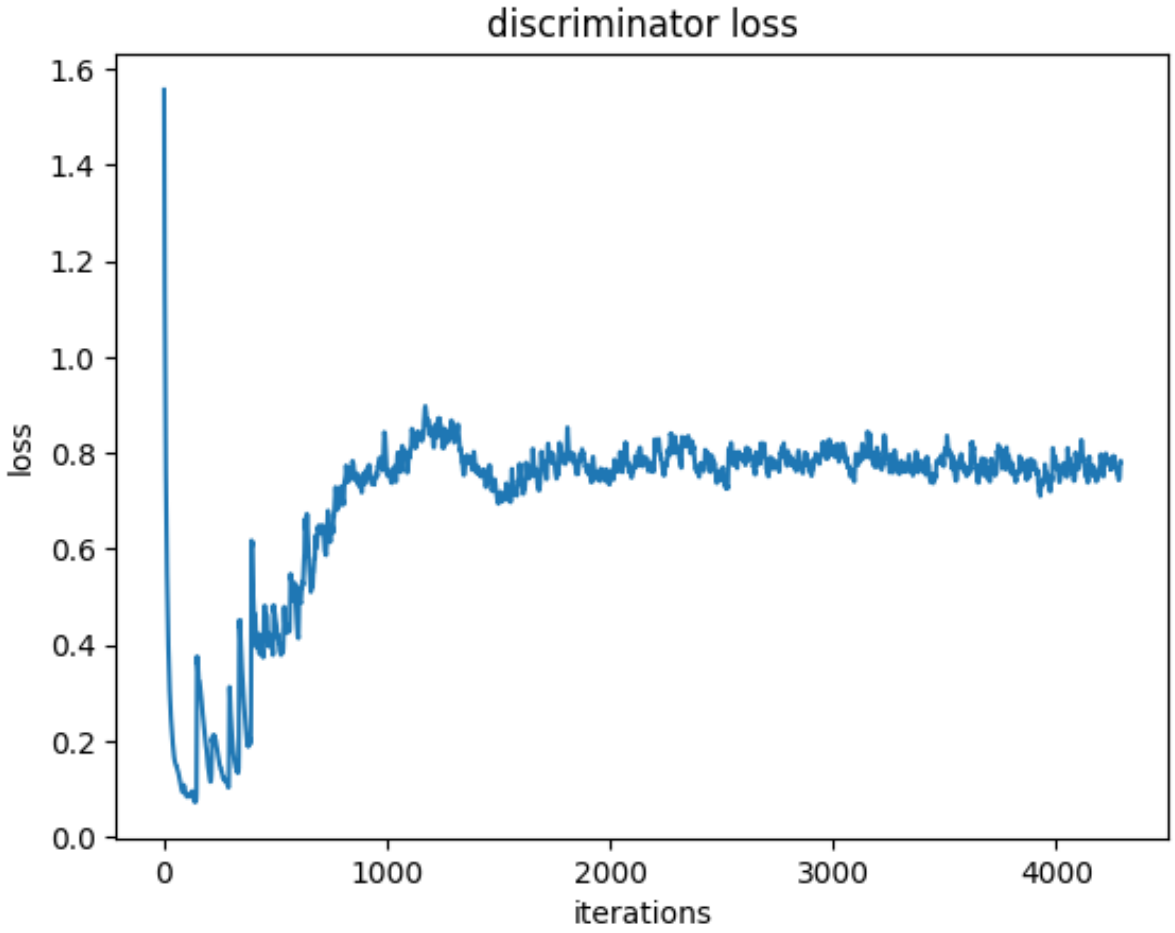

0
50
100
150
200
250
0
50
100
150
200
250

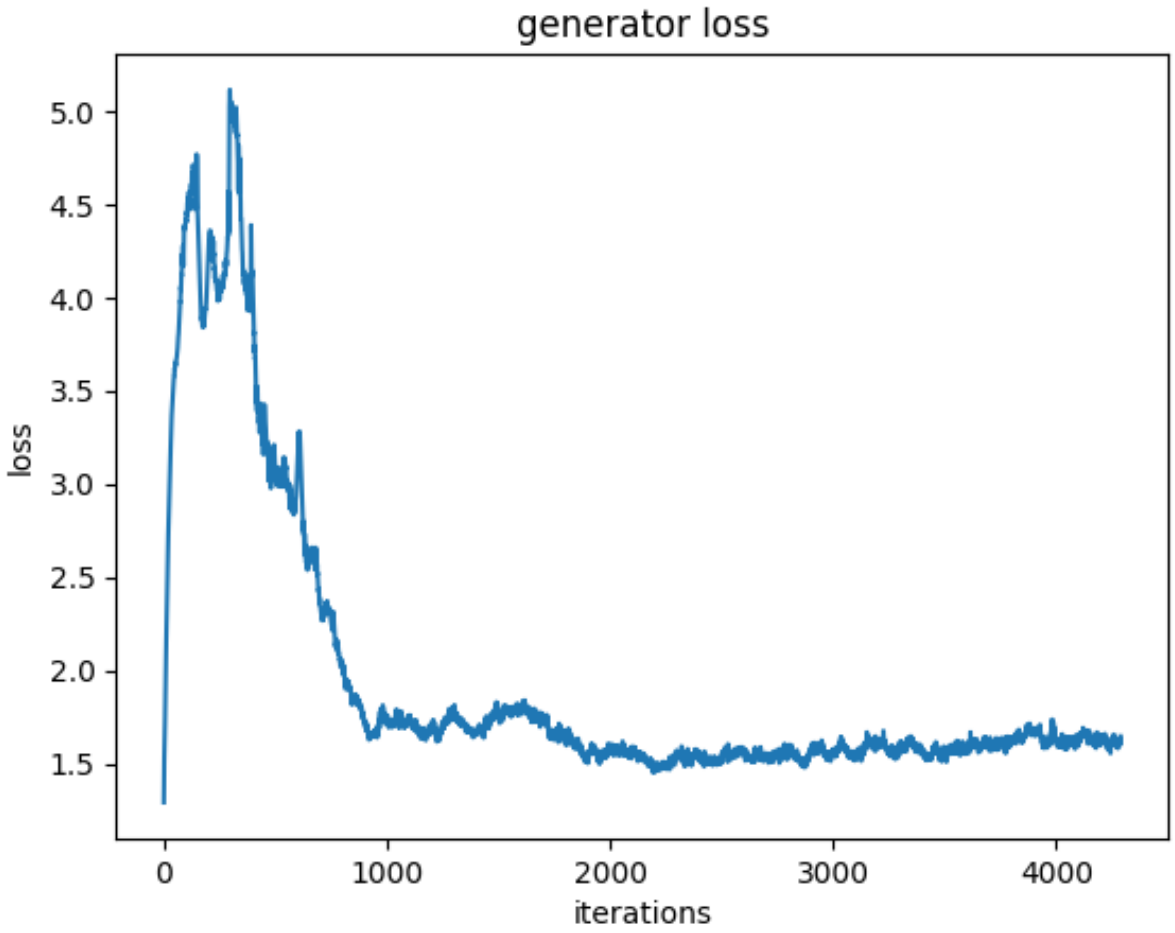

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
500
1000
1500
2000
2500
3000
3500



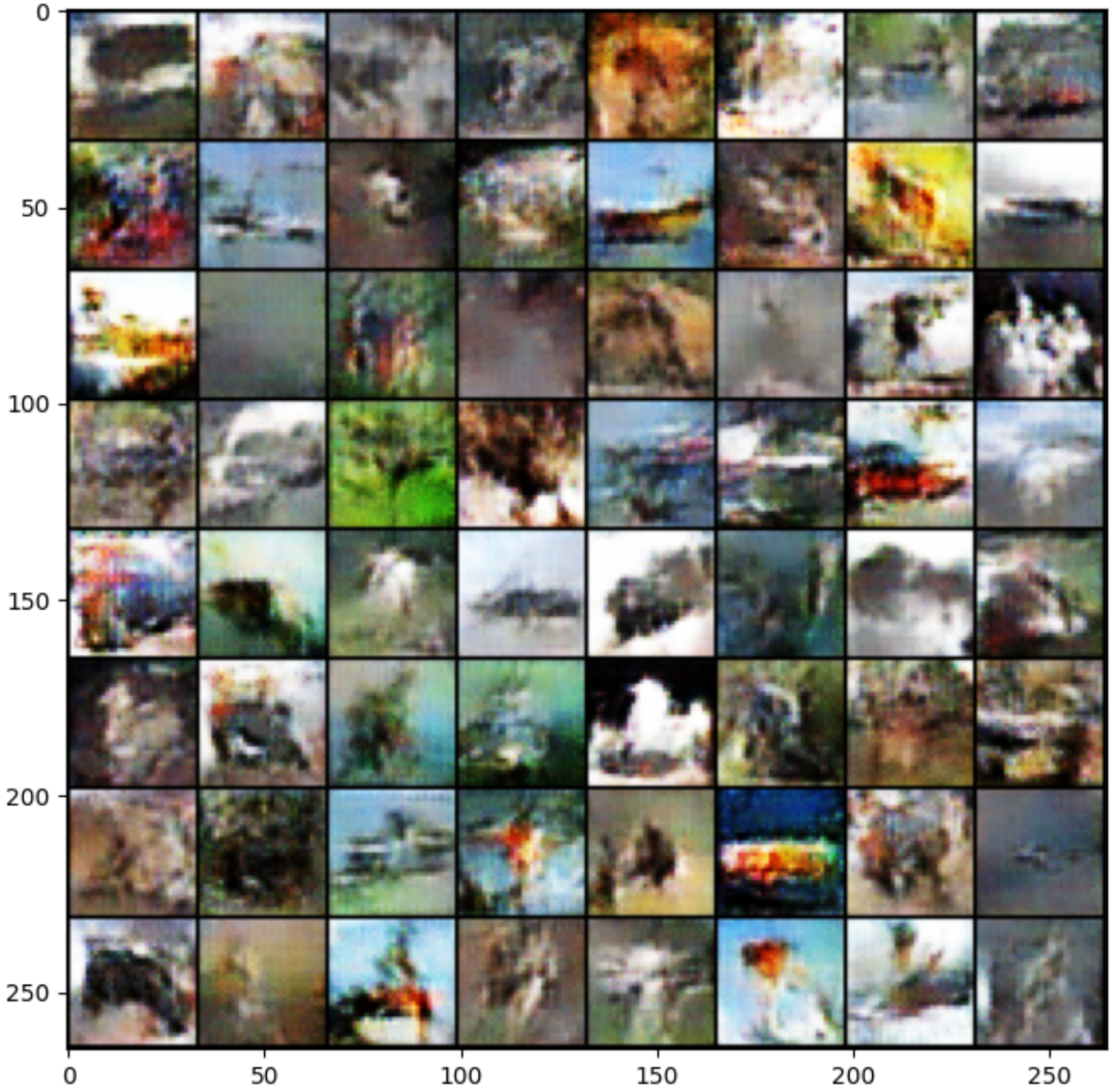

```
Iteration 3600/9750: dis loss = 0.7963, gen loss = 0.8346
Iteration 3700/9750: dis loss = 0.7665, gen loss = 1.1393
Iteration 3800/9750: dis loss = 0.7950, gen loss = 1.1007
Iteration 3900/9750: dis loss = 0.7816, gen loss = 1.0980
Iteration 4000/9750: dis loss = 0.6931, gen loss = 1.5344
Iteration 4100/9750: dis loss = 0.6327, gen loss = 1.6666
Iteration 4200/9750: dis loss = 1.1500, gen loss = 0.9170
```

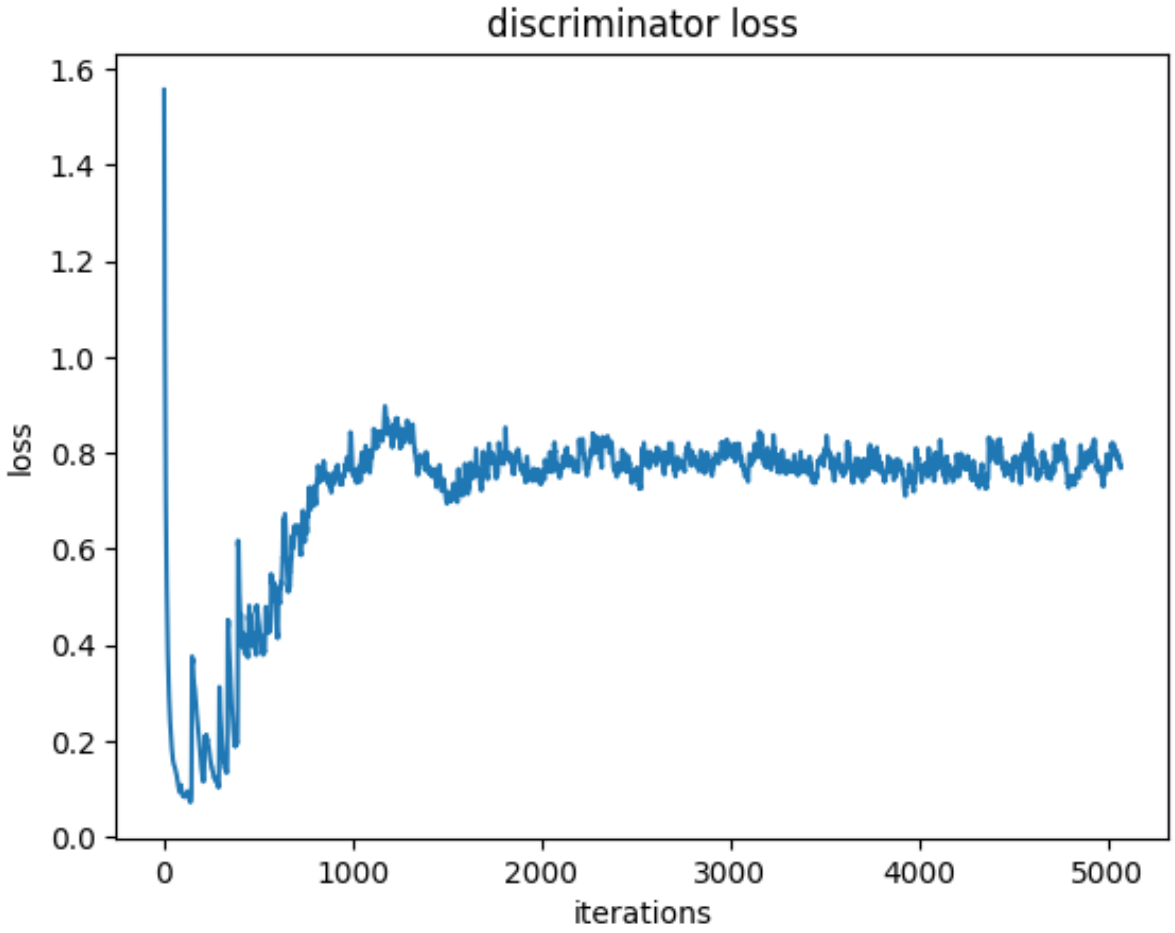

0
50
100
150
200
250
0
50
100
150
200
250

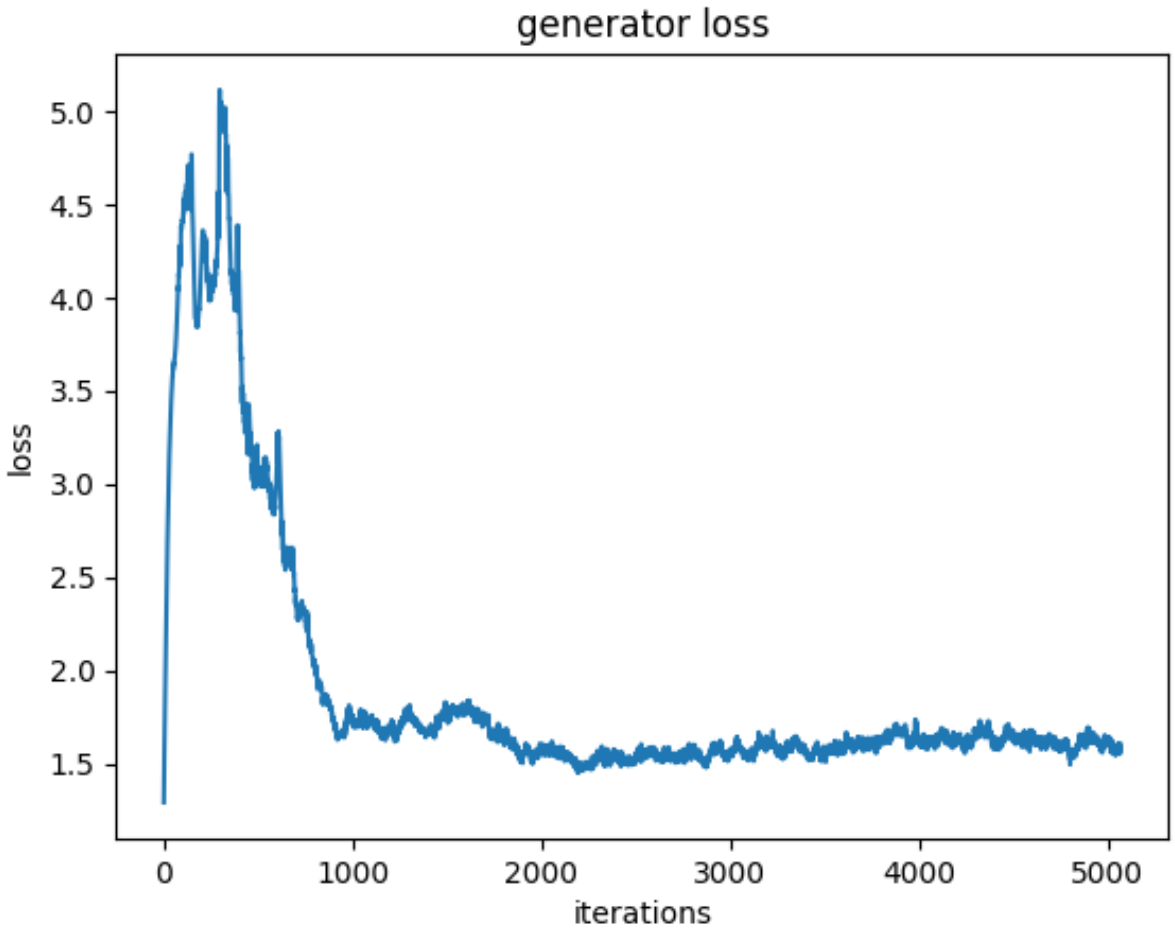

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000




```
Iteration 4300/9750: dis loss = 0.7540, gen loss = 1.3305
Iteration 4400/9750: dis loss = 0.7376, gen loss = 1.7924
Iteration 4500/9750: dis loss = 1.0584, gen loss = 2.6635
Iteration 4600/9750: dis loss = 0.6920, gen loss = 1.3996
Iteration 4700/9750: dis loss = 0.7326, gen loss = 1.5350
Iteration 4800/9750: dis loss = 0.7431, gen loss = 1.0100
Iteration 4900/9750: dis loss = 0.7700, gen loss = 1.3984
Iteration 5000/9750: dis loss = 0.7384, gen loss = 1.3174
```

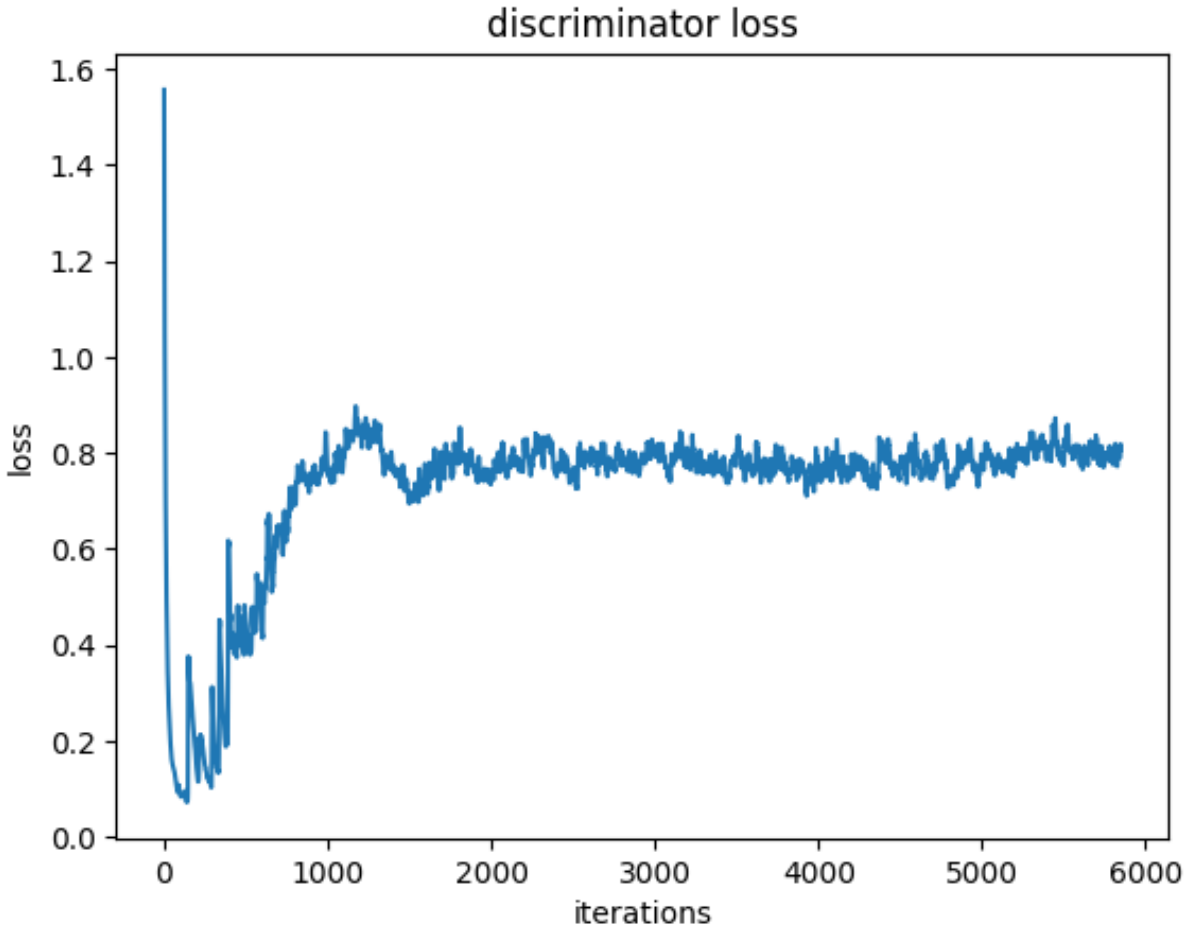

0
50
100
150
200
250
0
50
100
150
200
250

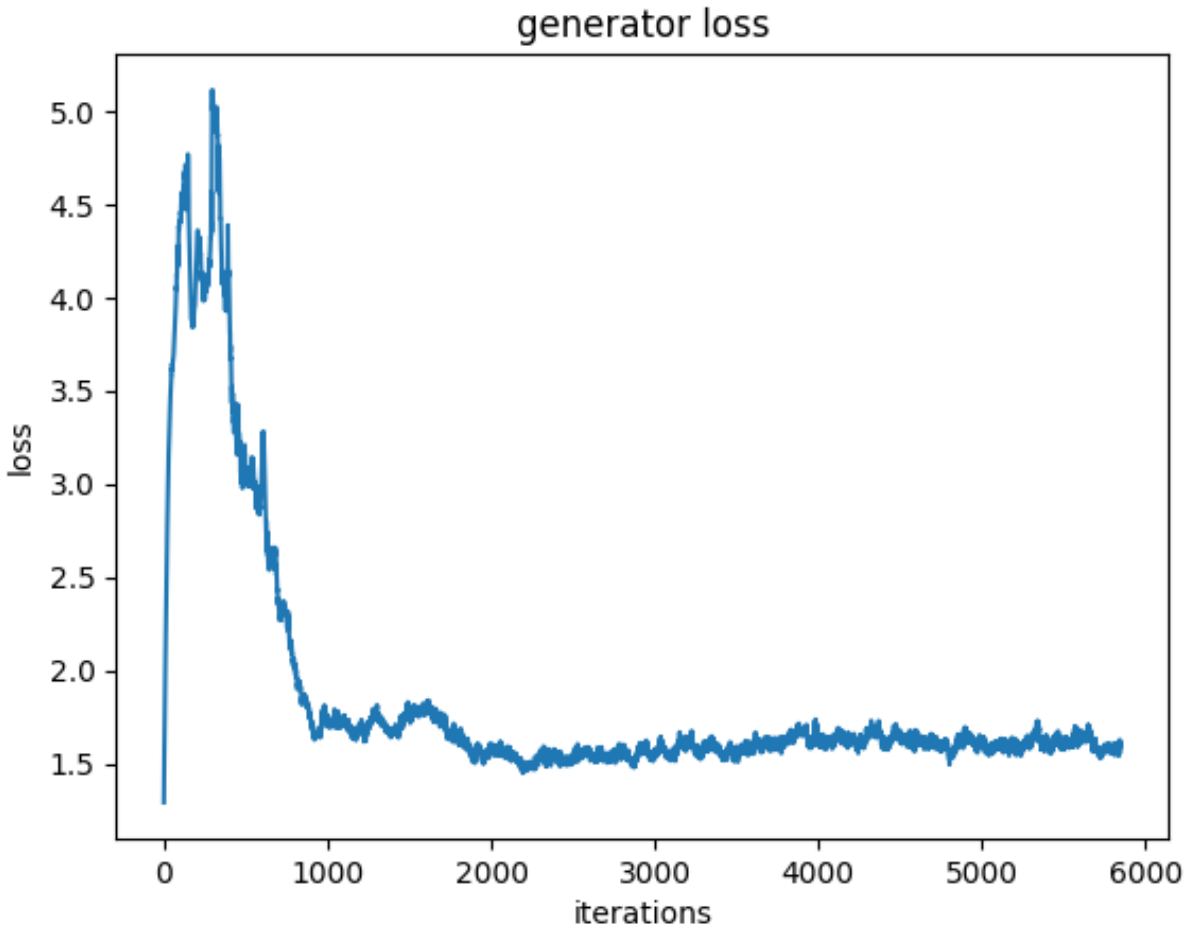

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000




```
Iteration 5100/9750: dis loss = 0.8755, gen loss = 1.1377
Iteration 5200/9750: dis loss = 1.1091, gen loss = 2.4292
Iteration 5300/9750: dis loss = 0.8409, gen loss = 1.9739
Iteration 5400/9750: dis loss = 0.7968, gen loss = 1.0593
Iteration 5500/9750: dis loss = 0.7999, gen loss = 1.9139
Iteration 5600/9750: dis loss = 0.7257, gen loss = 2.2147
Iteration 5700/9750: dis loss = 0.8030, gen loss = 1.4132
Iteration 5800/9750: dis loss = 0.8848, gen loss = 1.5824
```

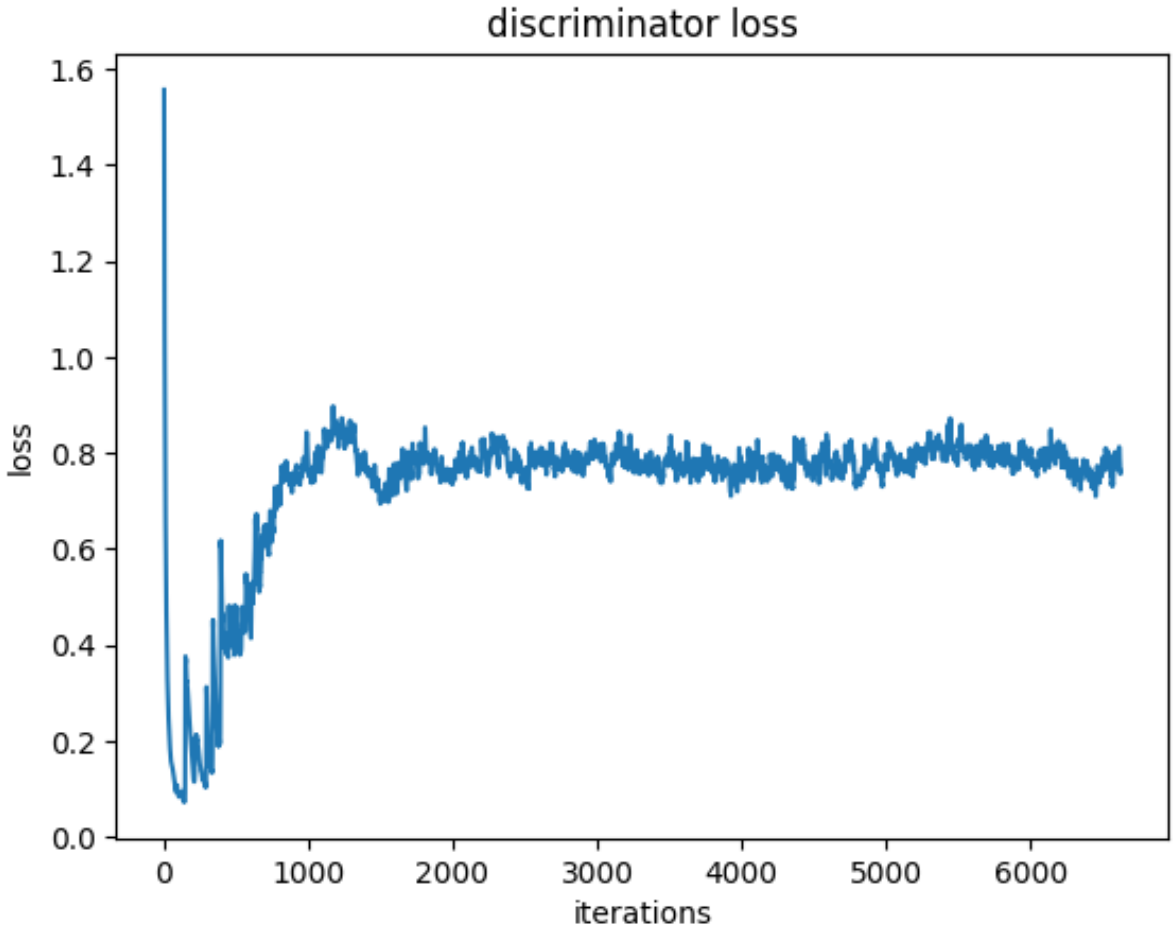

0
50
100
150
200
250
0
50
100
150
200
250

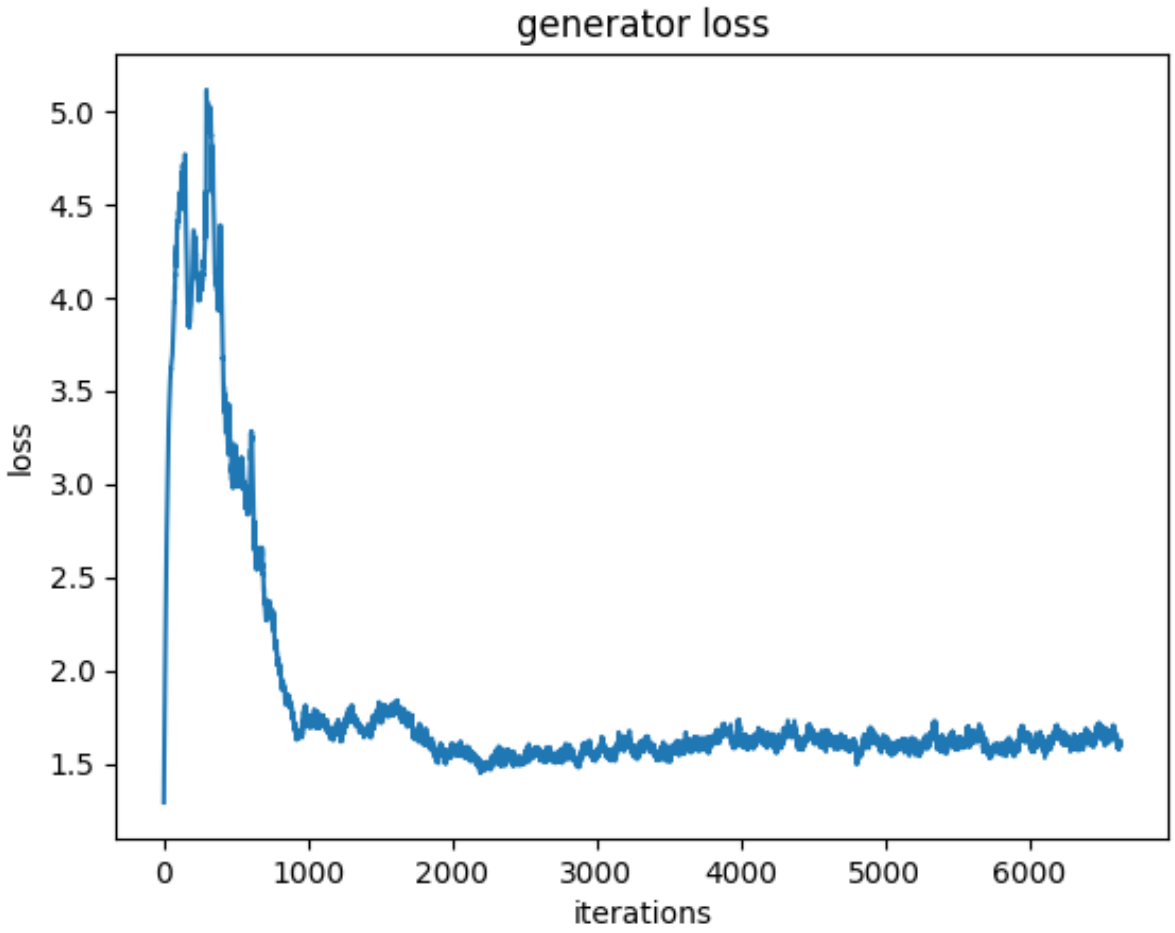

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000
6000



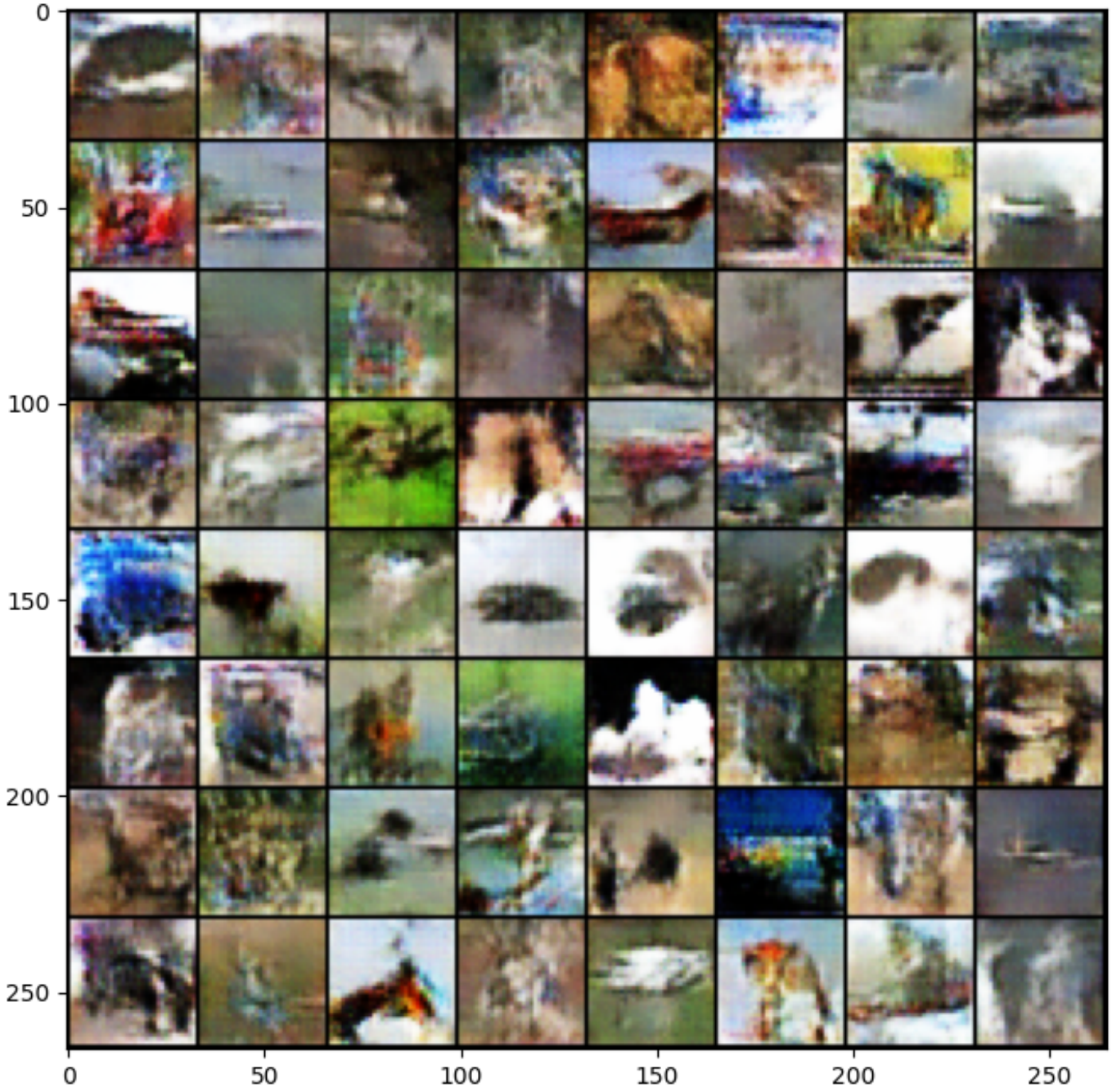

```
Iteration 5900/9750: dis loss = 0.8022, gen loss = 1.0957
Iteration 6000/9750: dis loss = 0.7542, gen loss = 1.9495
Iteration 6100/9750: dis loss = 0.6829, gen loss = 1.4492
Iteration 6200/9750: dis loss = 0.7307, gen loss = 1.5672
Iteration 6300/9750: dis loss = 0.6656, gen loss = 1.7371
Iteration 6400/9750: dis loss = 0.6690, gen loss = 2.1547
Iteration 6500/9750: dis loss = 0.6235, gen loss = 2.2482
Iteration 6600/9750: dis loss = 0.8733, gen loss = 1.5447
```

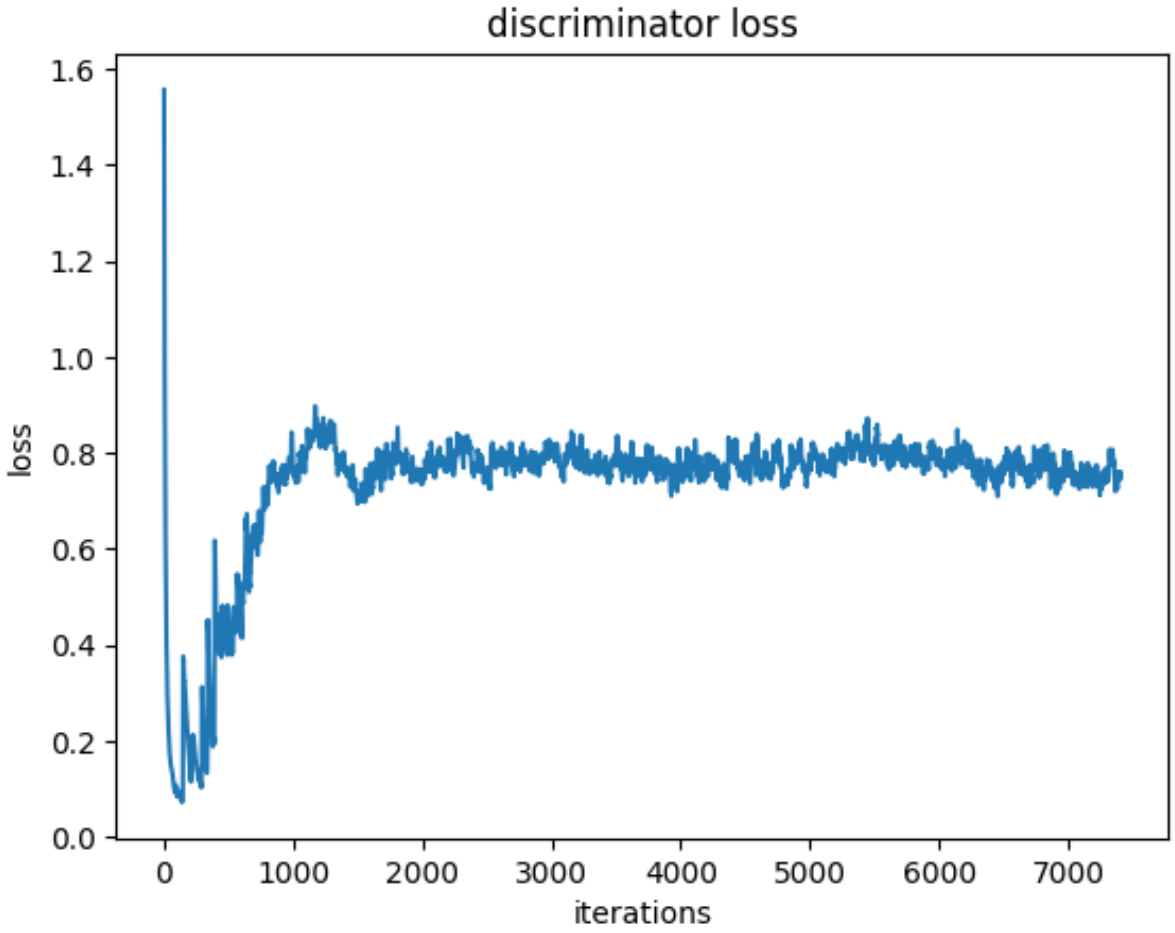

0
50
100
150
200
250
0
50
100
150
200
250

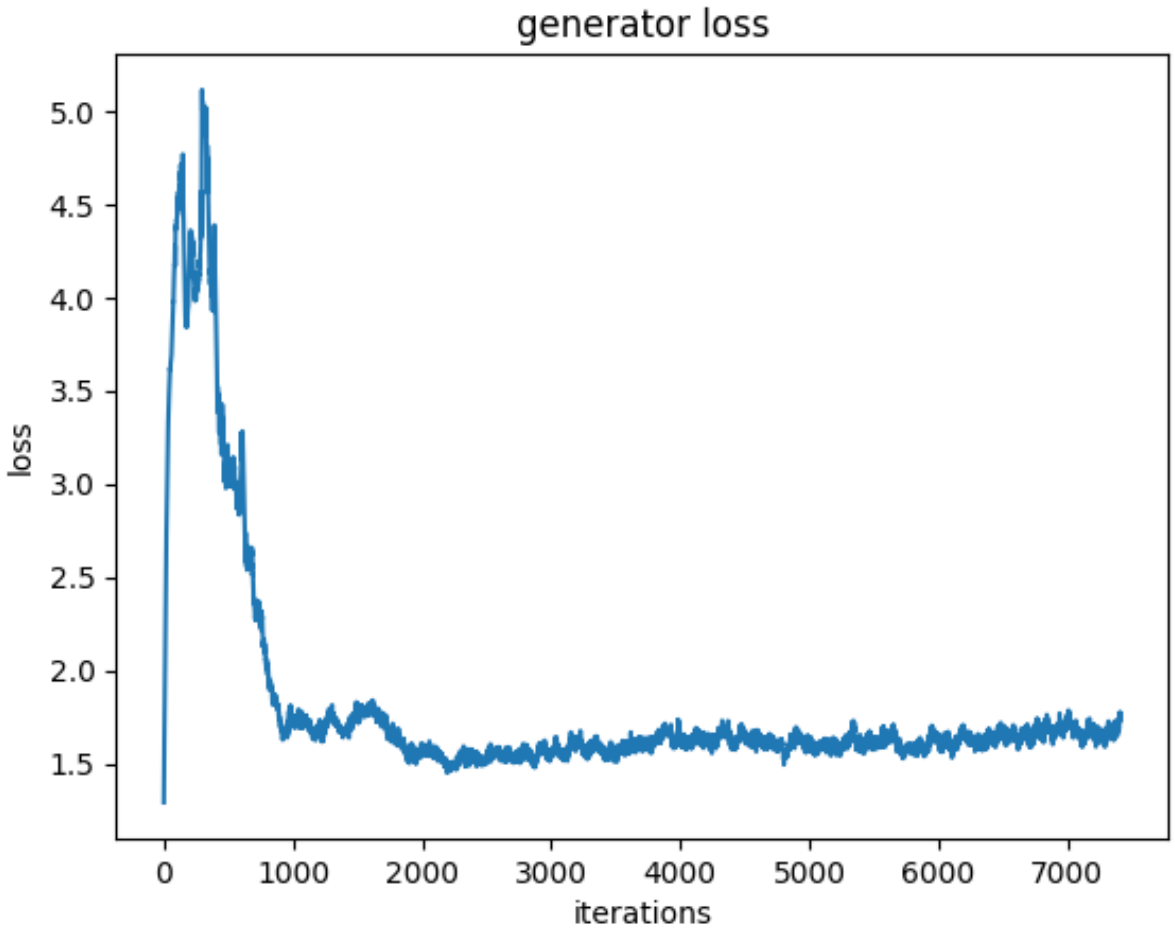

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000
6000




```
Iteration 6700/9750: dis loss = 0.8571, gen loss = 0.7673
Iteration 6800/9750: dis loss = 0.6732, gen loss = 1.6070
Iteration 6900/9750: dis loss = 0.6267, gen loss = 1.9393
Iteration 7000/9750: dis loss = 0.7319, gen loss = 1.6532
Iteration 7100/9750: dis loss = 0.7837, gen loss = 2.3271
Iteration 7200/9750: dis loss = 0.8219, gen loss = 2.4813
Iteration 7300/9750: dis loss = 0.6894, gen loss = 1.5732
Iteration 7400/9750: dis loss = 0.7606, gen loss = 3.0267
```

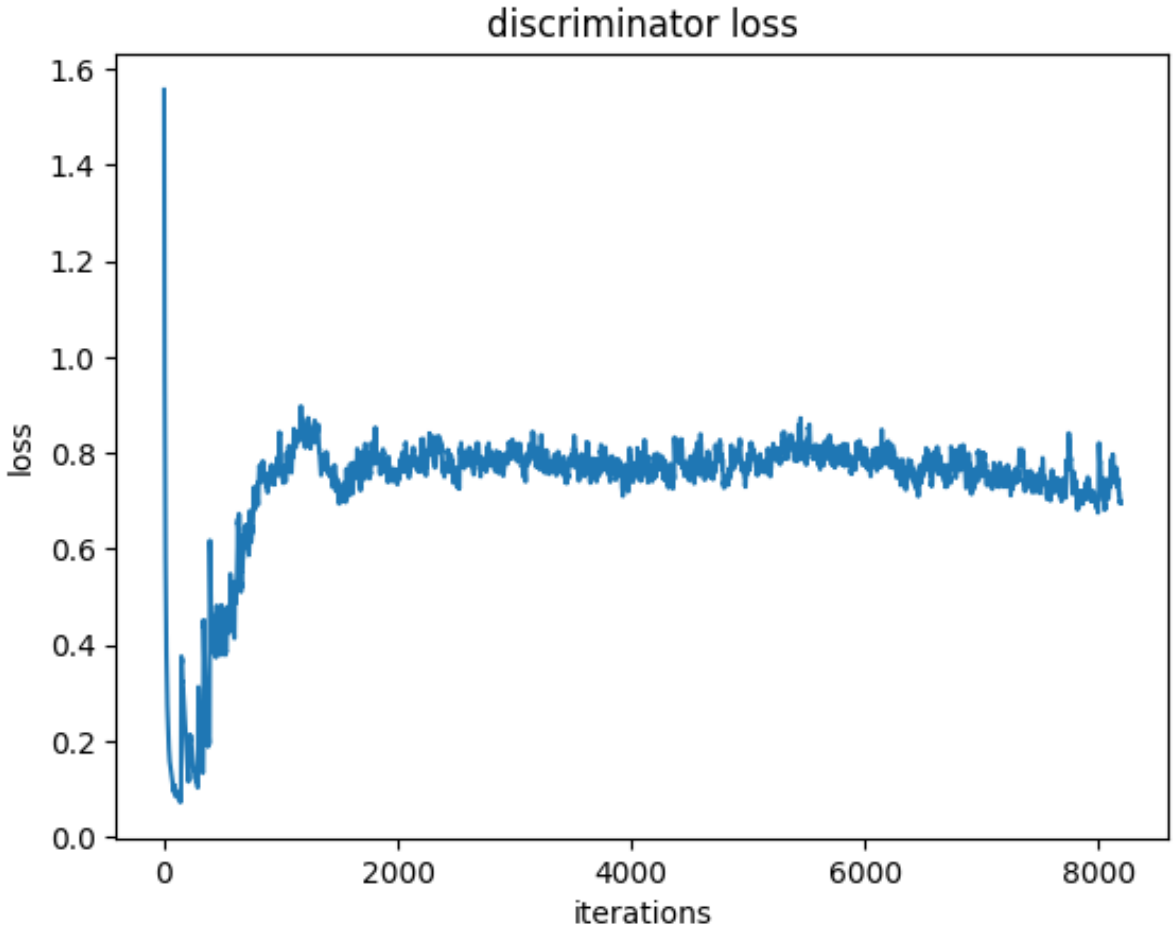

0
50
100
150
200
250
0
50
100
150
200
250

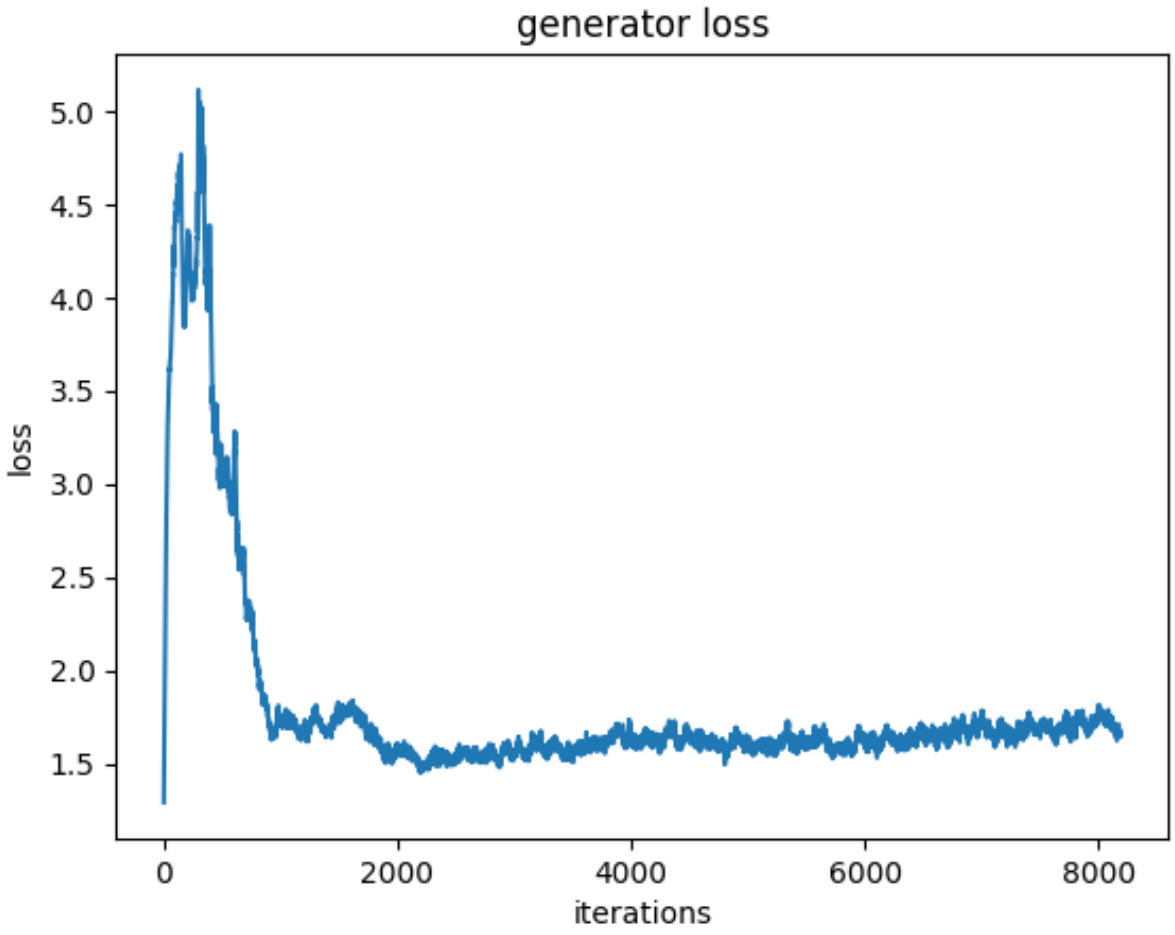

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000
5000
6000
7000




```
Iteration 7500/9750: dis loss = 0.7947, gen loss = 2.1142
Iteration 7600/9750: dis loss = 0.8206, gen loss = 0.8490
Iteration 7700/9750: dis loss = 0.8314, gen loss = 2.2984
Iteration 7800/9750: dis loss = 0.9620, gen loss = 1.0801
Iteration 7900/9750: dis loss = 0.7580, gen loss = 1.0877
Iteration 8000/9750: dis loss = 1.1480, gen loss = 3.4799
Iteration 8100/9750: dis loss = 0.7794, gen loss = 1.0463
```

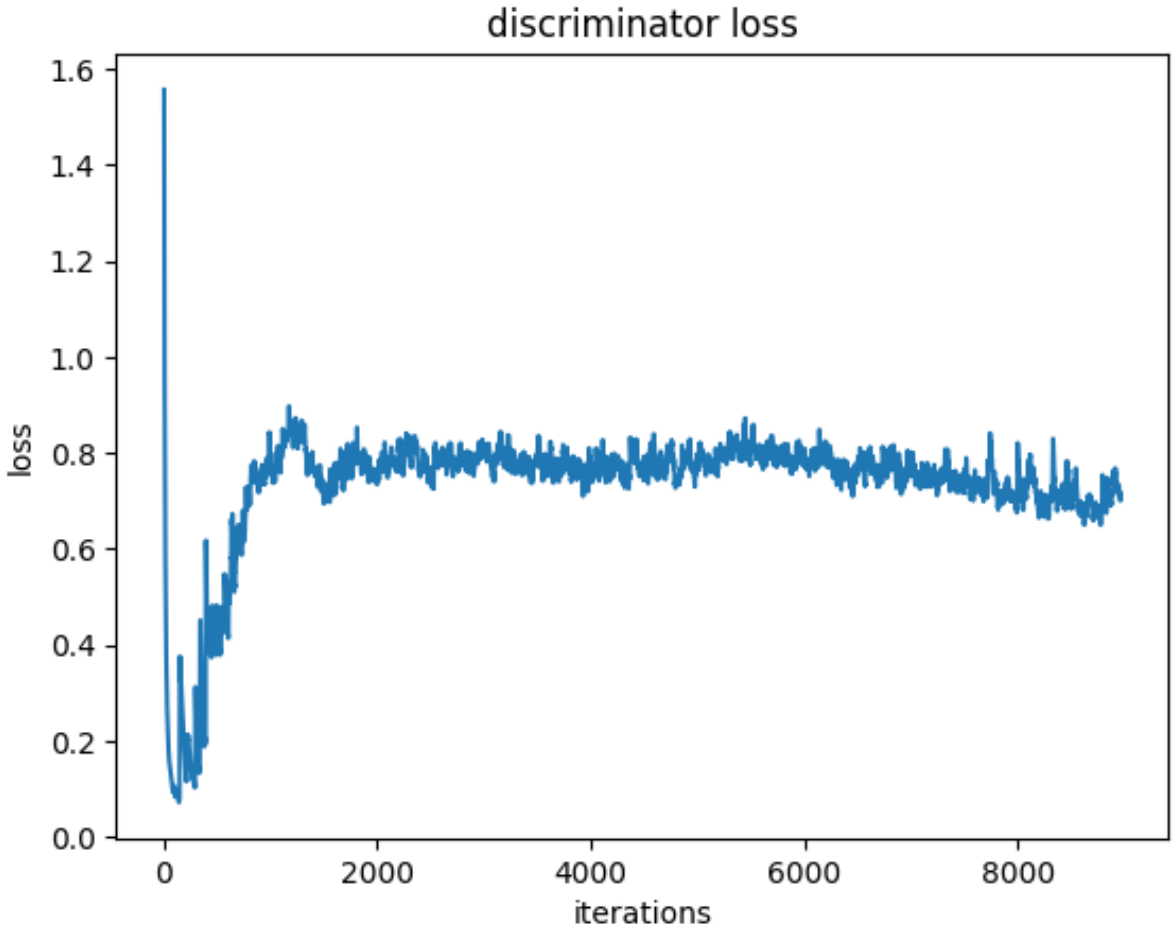

0
50
100
150
200
250
0
50
100
150
200
250

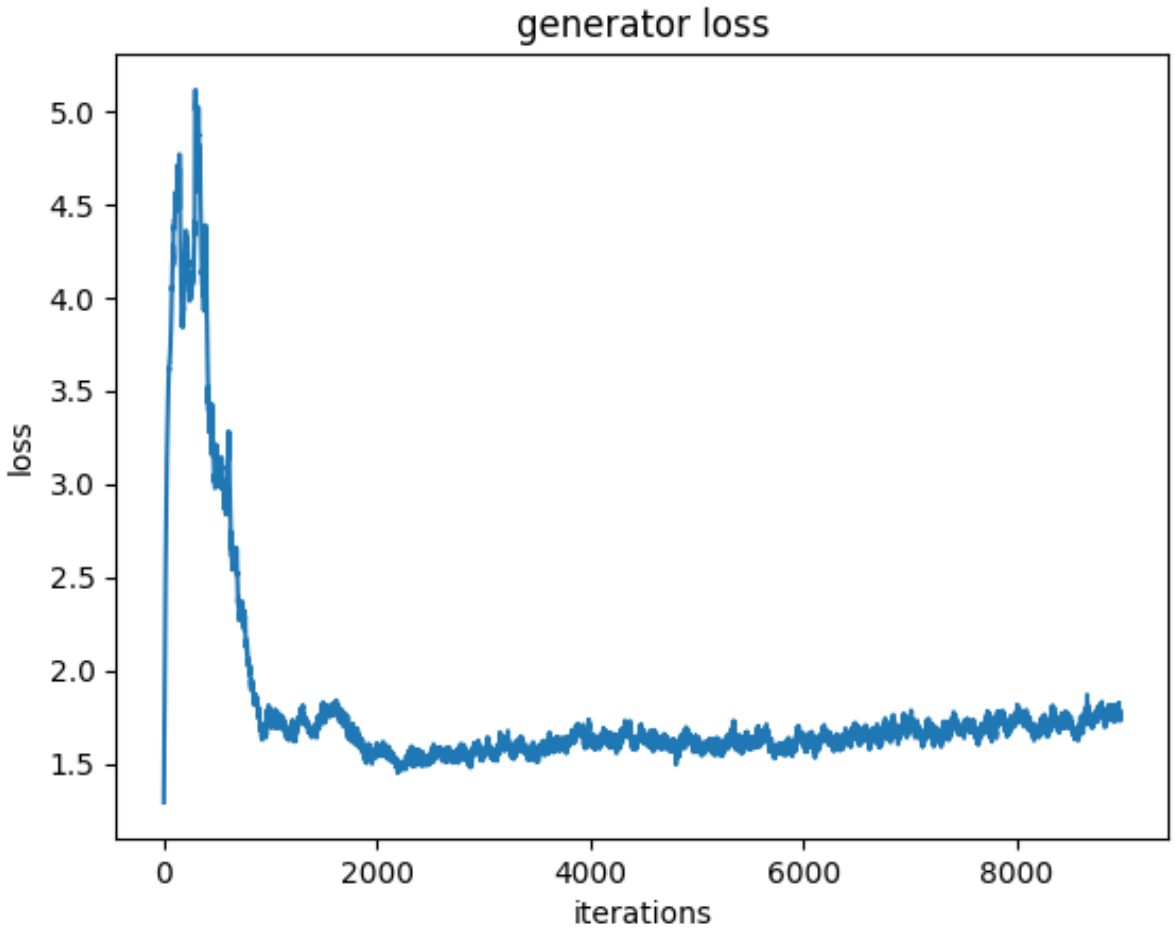

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
2000
4000
6000
8000



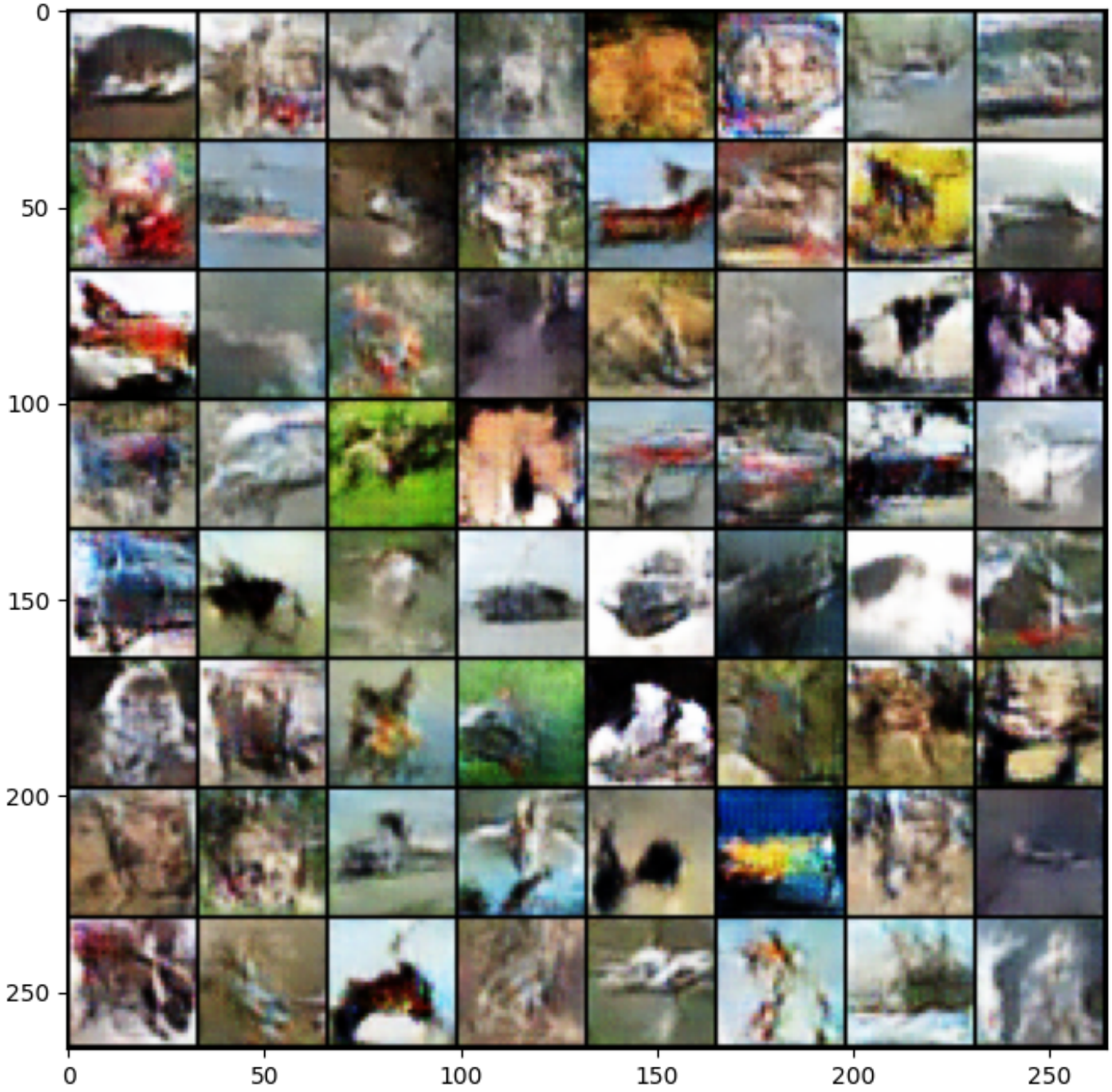

```
Iteration 8200/9750: dis loss = 0.5844, gen loss = 1.4524
Iteration 8300/9750: dis loss = 0.9695, gen loss = 0.7061
Iteration 8400/9750: dis loss = 0.6709, gen loss = 2.2241
Iteration 8500/9750: dis loss = 0.6629, gen loss = 1.7931
Iteration 8600/9750: dis loss = 0.6186, gen loss = 1.4471
Iteration 8700/9750: dis loss = 0.6444, gen loss = 1.8957
Iteration 8800/9750: dis loss = 0.7366, gen loss = 1.3582
Iteration 8900/9750: dis loss = 1.0623, gen loss = 0.7952
```

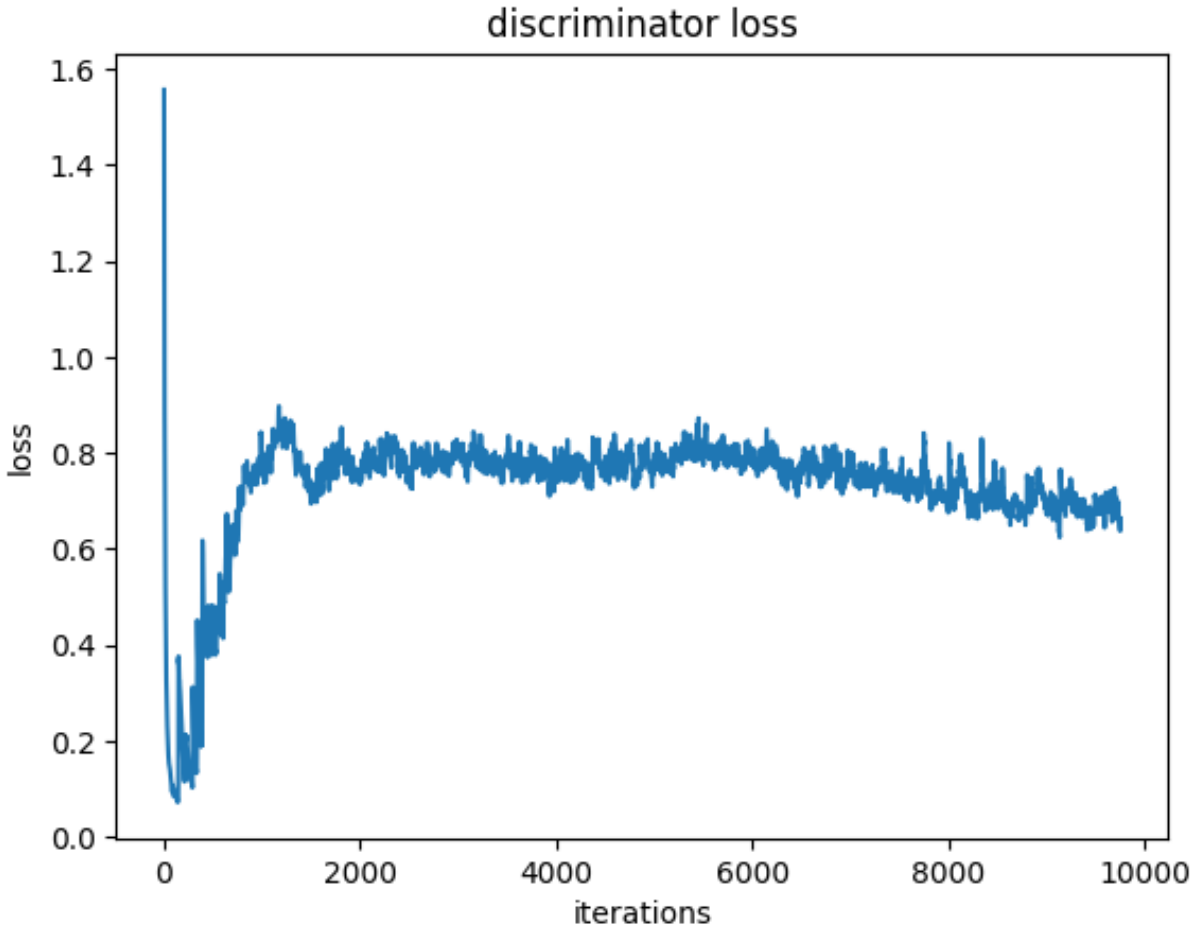

0
50
100
150
200
250
0
50
100
150
200
250

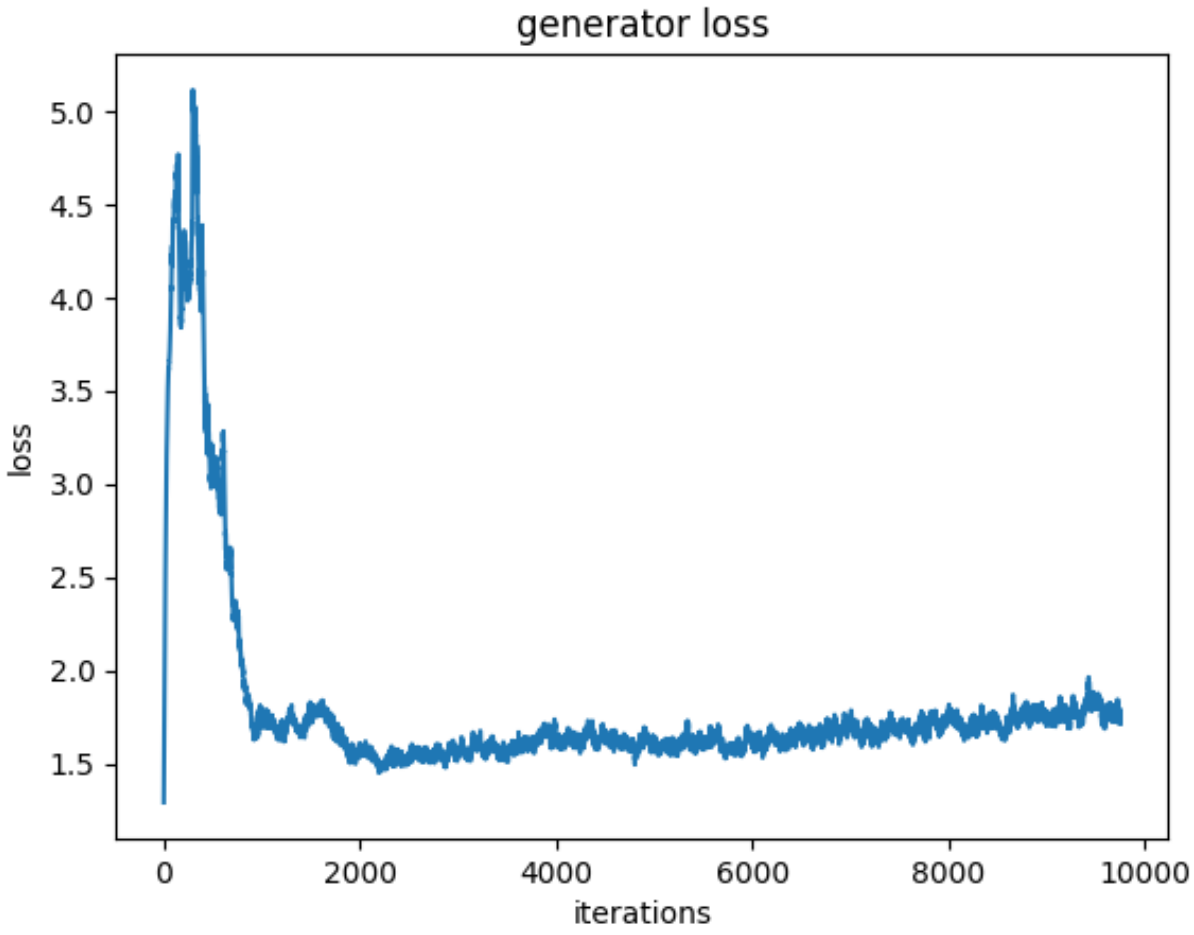

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
2000
4000
6000
8000




```
Iteration 9000/9750: dis loss = 0.6159, gen loss = 2.2461
Iteration 9100/9750: dis loss = 0.7087, gen loss = 2.1600
Iteration 9200/9750: dis loss = 0.6690, gen loss = 1.3973
Iteration 9300/9750: dis loss = 0.6230, gen loss = 1.9766
Iteration 9400/9750: dis loss = 0.4603, gen loss = 1.9263
Iteration 9500/9750: dis loss = 0.7027, gen loss = 2.0198
Iteration 9600/9750: dis loss = 0.9318, gen loss = 2.3378
Iteration 9700/9750: dis loss = 0.8138, gen loss = 0.9159
```


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
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:

```
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,
2, 0)))
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:

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

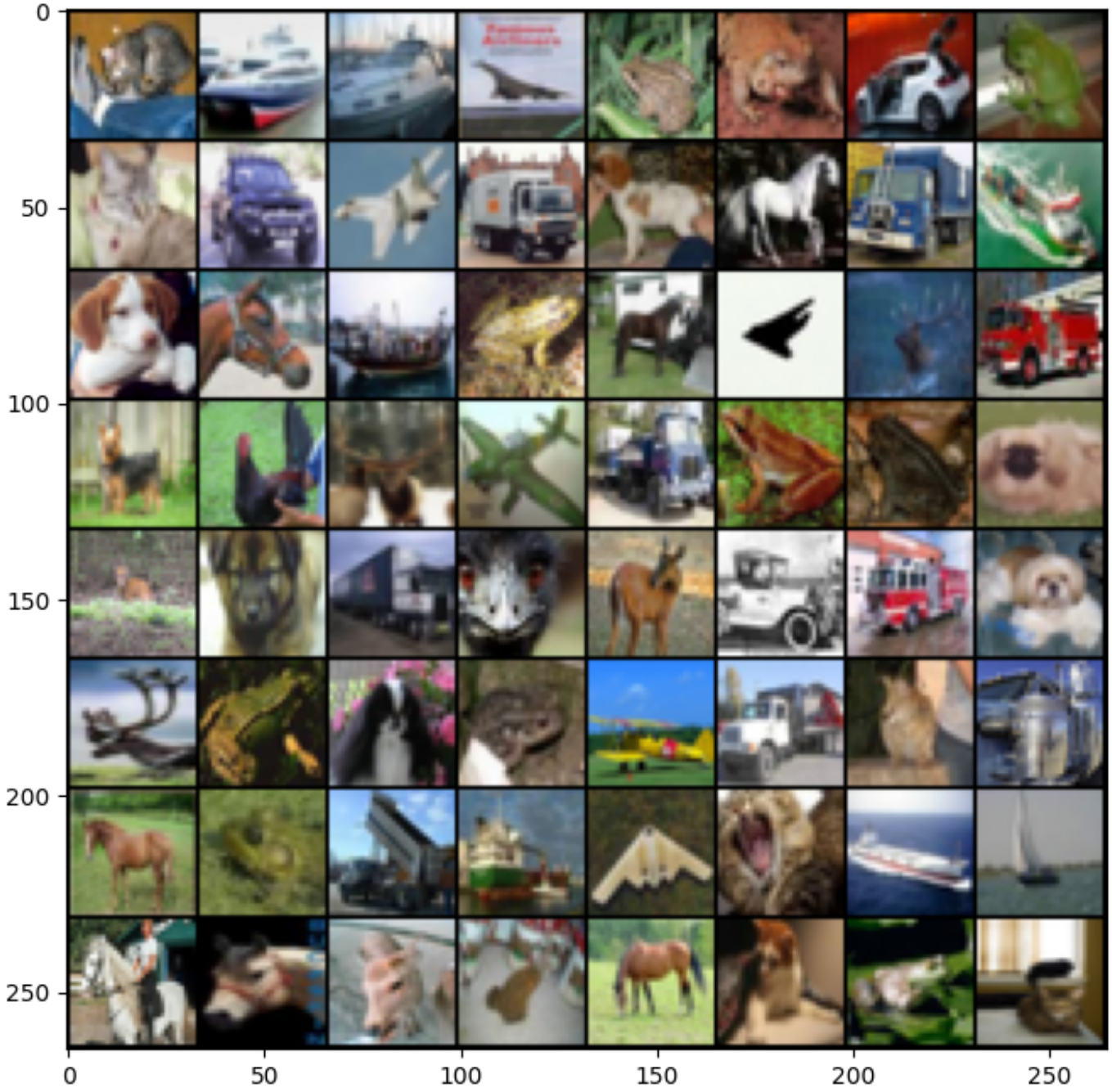
This time, we always start from a zero vector.

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.

```
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().transpose((1, 2, 0)))
plt.show()
fig = plt.figure(figsize = (8, 8))
ax1 = plt.subplot(111)
ax1.imshow(make_grid(reconstructions, padding=1, normalize=True).numpy().transpose((1,
2, 0)))
plt.show()
```

```
average reconstruction loss = 0.0139
```


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

Submission Instruction

See the pinned Piazza post for detailed instruction.