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

✓ 4s completed at 6:26 PM

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

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 23:08:08--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.
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... con
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  69.1MB/s    in 2.4s

2023-04-03 23:08:10 (69.1 MB/s) - 'data/cifar-10-python.tar.gz' saved [170498

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, st
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.ConvTranspose2d(in_channels=64, out_channels=32, kernel_size=4, str
            nn.BatchNorm2d(32),
            nn.LeakyReLU(),
            nn.ConvTranspose2d(in_channels=32, out_channels=3, kernel_size=4, stri
            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, pad
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2, pa
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2, p
            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", respectiv
```

```
    # (also move them to torch device for accelerating the training later)
    #start_shape_tuple = tuple(self.tracked_noise.shape)
    self._generator = Generator(starting_shape=self.code_size).to(device)
    self._discriminator = Discriminator().to(device)

    # To-Do: Apply weight initialization (first implement the weight initializ
    # function below by following the given link)
    self._generator.apply(self._weight_initialization)
    self._discriminator.apply(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
    # Setup Adam optimizers for both G and D
    self.optimizerD = torch.optim.Adam(self._discriminator.parameters(), lr=se
    self.optimizerG = torch.optim.Adam(self._generator.parameters(), lr=self.l


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



# To-Do: Define a general classification loss function (sigmoid followed by bi
def _classification_loss(self,prediction,label):

  loss_func = nn.BCEWithLogitsLoss()
  loss = loss_func(prediction,label)# should be prediction, target

  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
        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 f
        self.optimizerD.zero_grad()

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

        # To-Do: calculate the discriminator loss for real samples
        # use the variable name "real_dis_loss"
        real_dis_loss = self._classification_loss(dis_real_output,real_lab
        #real_label is just a array of 1s saying the output is real
        ########################################################################
        # 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_noise = torch.randn([self.batch_size, self.code_size], device
        # To-Do: generate fake samples from the noise using the generator
        fake_samples = self._generator.forward(fake_noise)
        # To-Do: feed fake samples to discriminator
        # Make sure to detach the fake samples from the gradient calculati
        # when feeding to the discriminator, we don't want the discriminat
        # receive gradient info from the Generator
        fake_samples = fake_samples.detach()
        dis_fake_output = self._discriminator.forward(fake_samples)

        # To-Do: calculate the discriminator loss for fake samples
```

```
                # use the variable name "fake_dis_loss"
                fake_dis_loss = self._classification_loss(dis_fake_output,fake_lab


                # To-Do: calculate the total discriminator loss (real loss + fake
                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! (.ze
                self.optimizerG.zero_grad()

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

                # To-Do: generate fake samples from the noise using the generator
                fake_samples_g = self._generator.forward(fake_noise_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.
                dis_fake_output_g = self._discriminator.forward(fake_samples_g)
                #print(dis_fake_output_g)
                #print(real_label)
                #print(self._classification_loss(dis_fake_output_g,real_label))

                # 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(dis_fake_output_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_f
                plot_gen_s = plot_gen_s * smooth_factor + gen_loss * (1 - smooth_f
                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:.4

            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()).
                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,prediction,label):
  mse_loss = nn.MSELoss()
  loss = mse_loss(prediction,label)
  return loss



def reconstruct(self, samples):
```

```
        recon_code = torch.zeros([samples.shape[0], self.code_size], device=device
        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
        ########################################################################
        # To-Do: define the optimizer
        # Hinit: Use self.vis_learning_rate as one of the parameters for Adam opti
        recon_optimizer = 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 probl
            ########################################################################
            # To-Do: eliminate the gradients
            recon_optimizer.zero_grad()

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


            # 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
            recon_optimizer.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
        actmax_optimizer = torch.optim.Adam([actmax_code], lr=self.vis_learning_ra
        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()
```

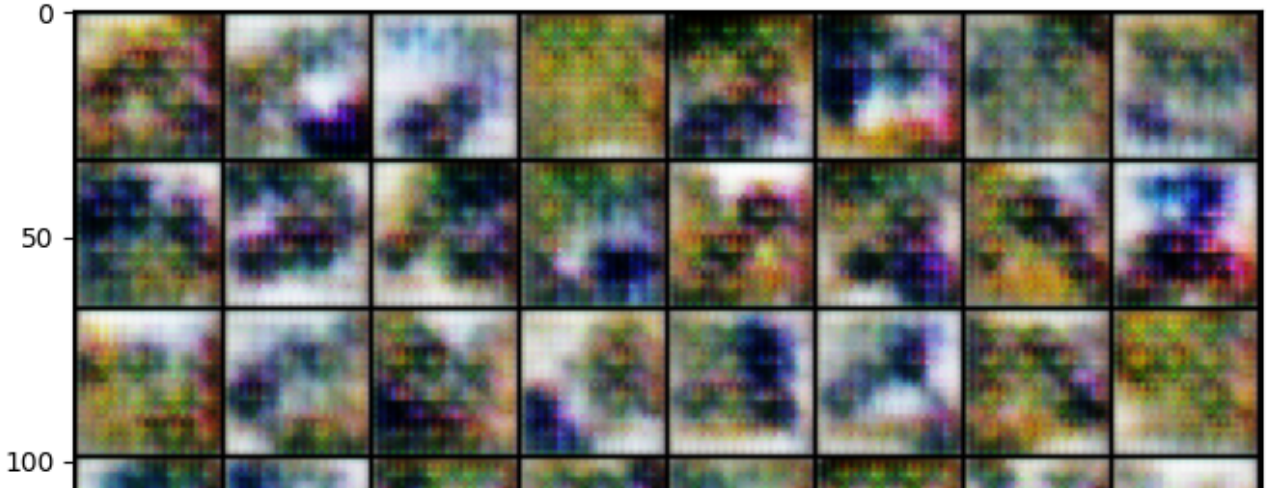
Now let's do the training!

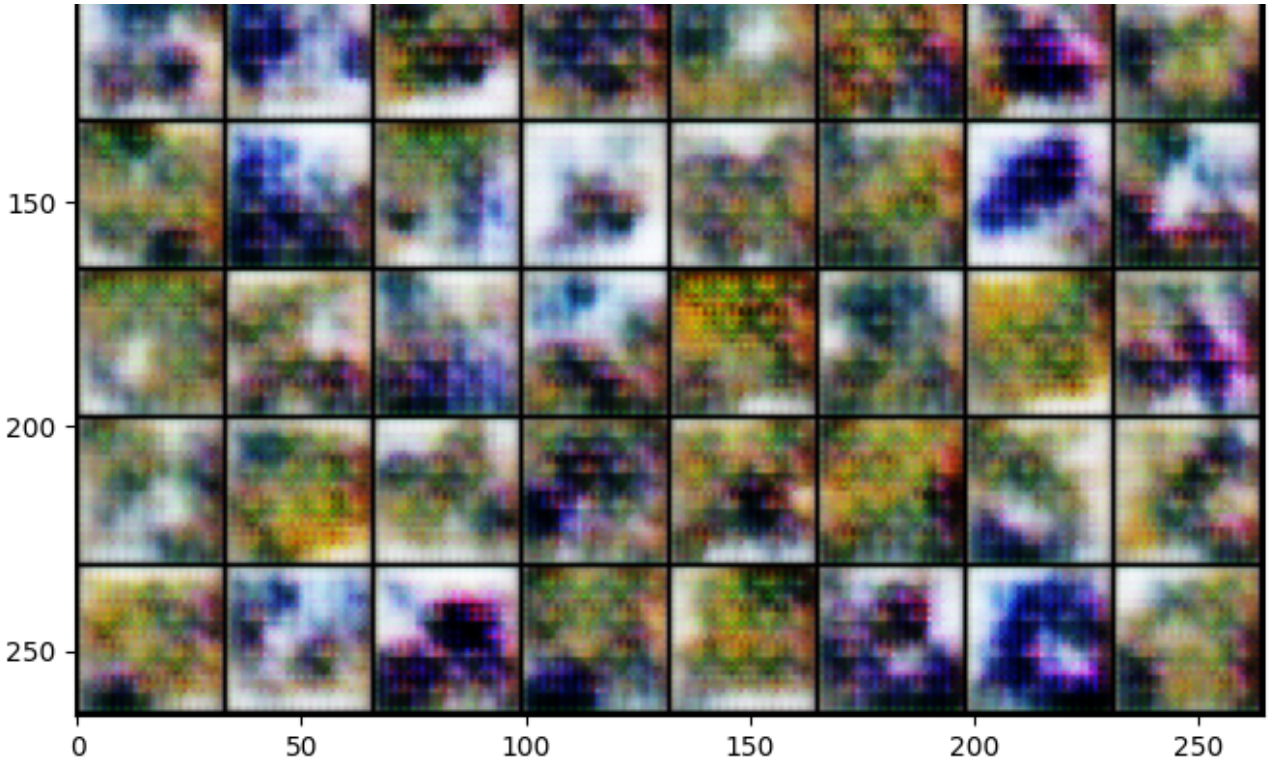
Don't panic if the loss curve goes wild. The two networks are competing for the loss curve to go different directions, so virtually anything can happen. If your code is correct, the generated samples should have a high variety.

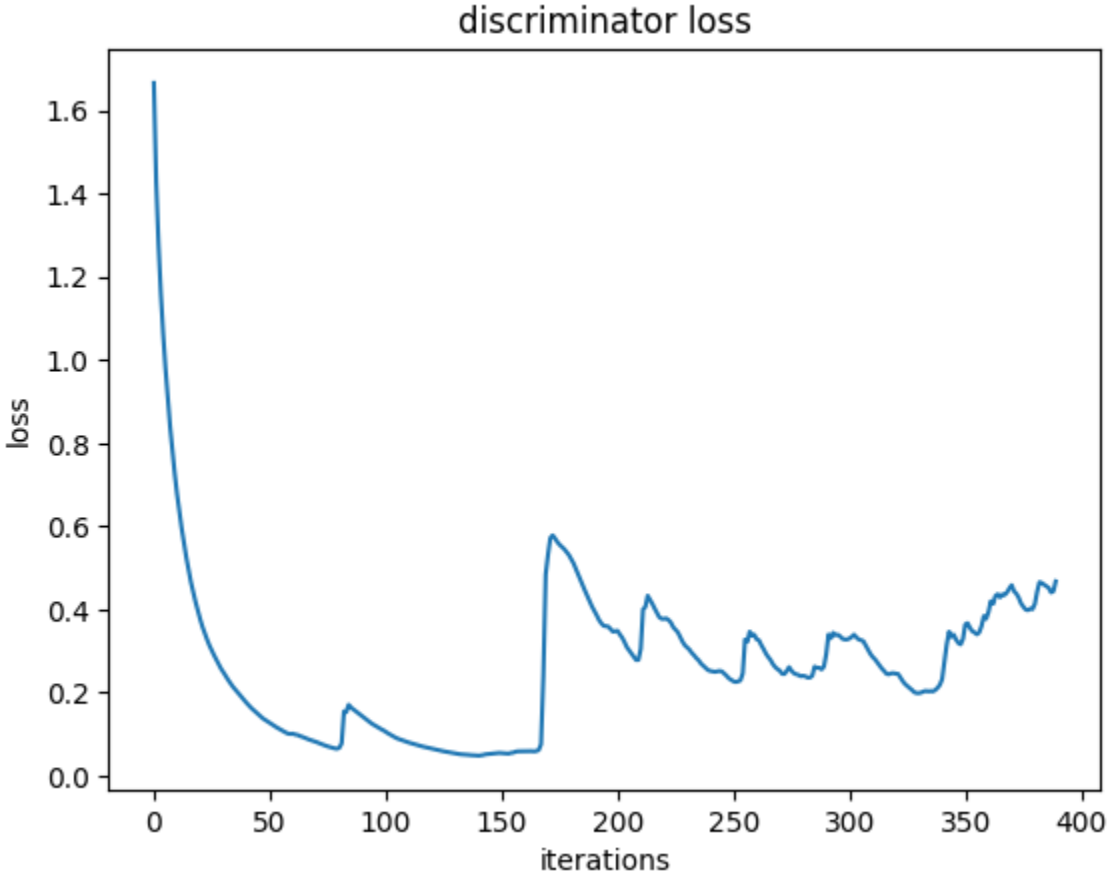
Do NOT change the number of epochs, learning rate, or batch size. If you're using Google Colab, the batch size will not be an issue during training.

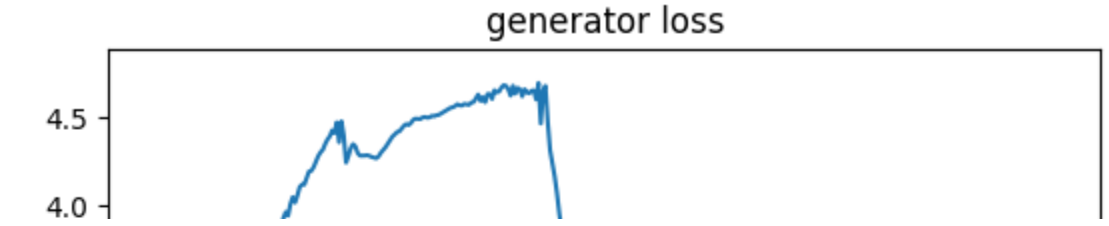
```
set_seed(42)

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

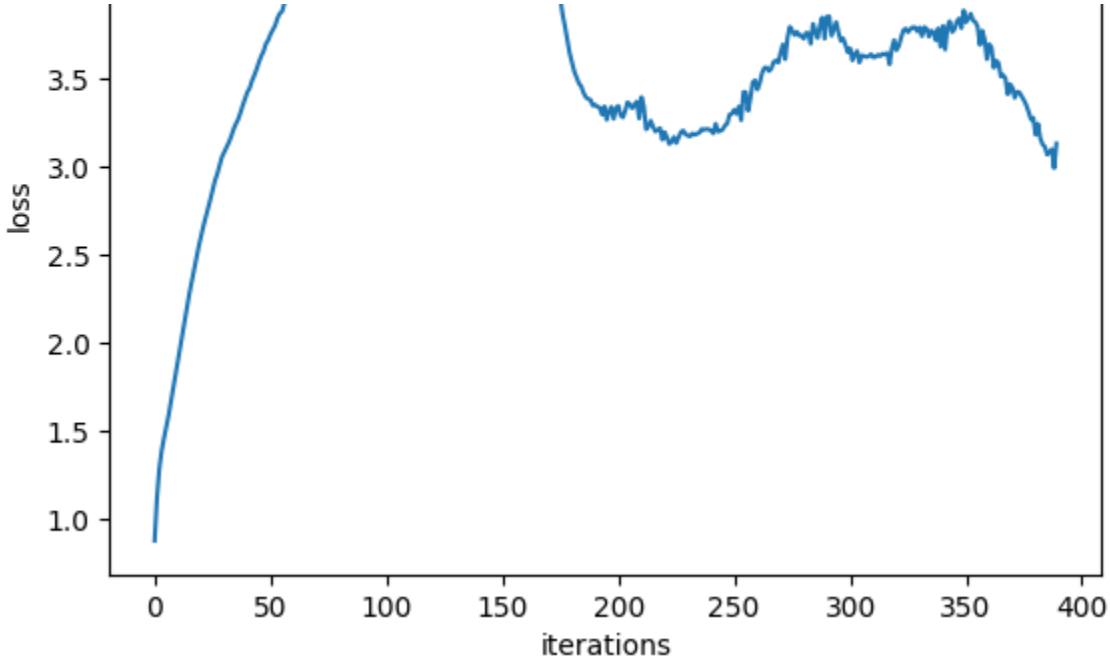
```
    Start training ...
    Iteration 100/9750: dis loss = 0.0554, gen loss = 4.6041
    Iteration 200/9750: dis loss = 0.3625, gen loss = 4.5111
    Iteration 300/9750: dis loss = 0.3287, gen loss = 2.8567
```



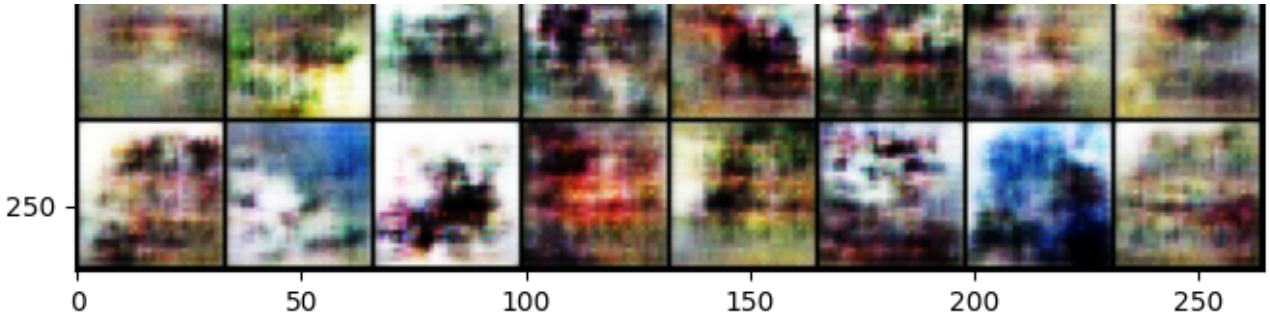

150
200
250
0
50
100
150
200
250

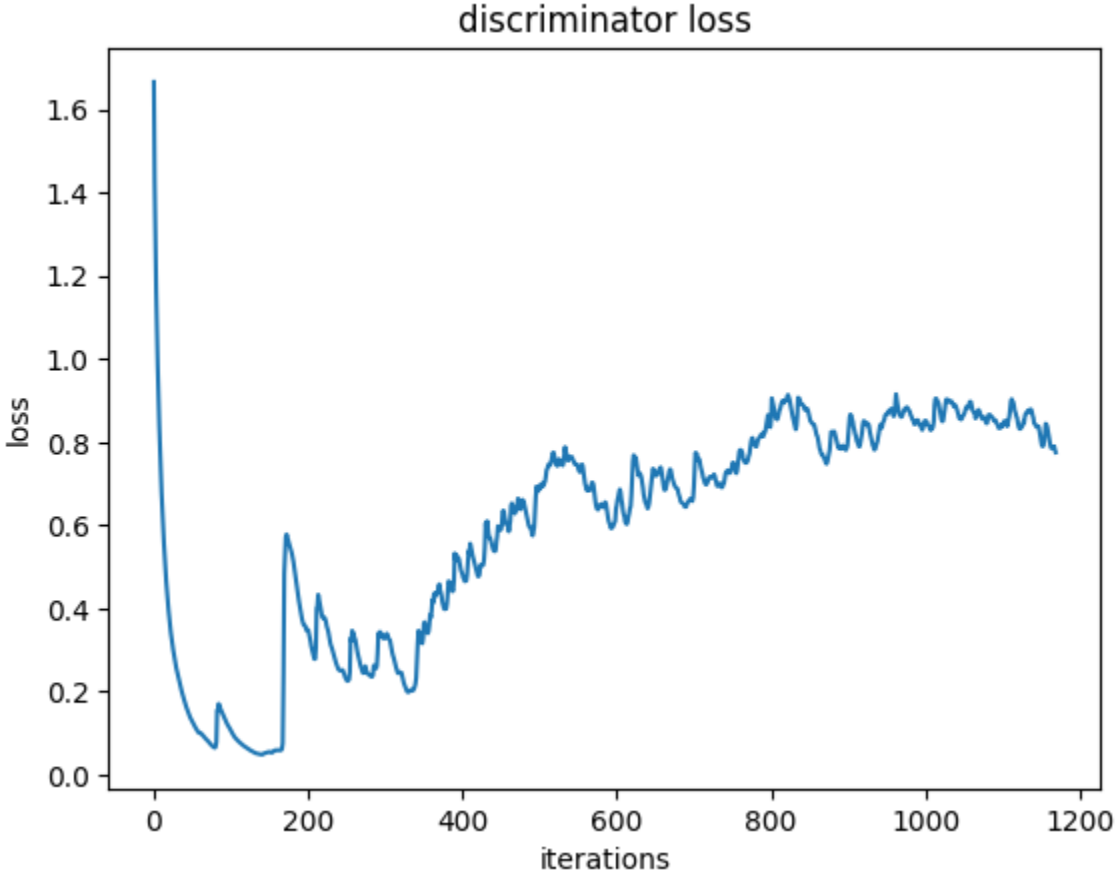

discriminator loss
loss
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300
350
400
iterations

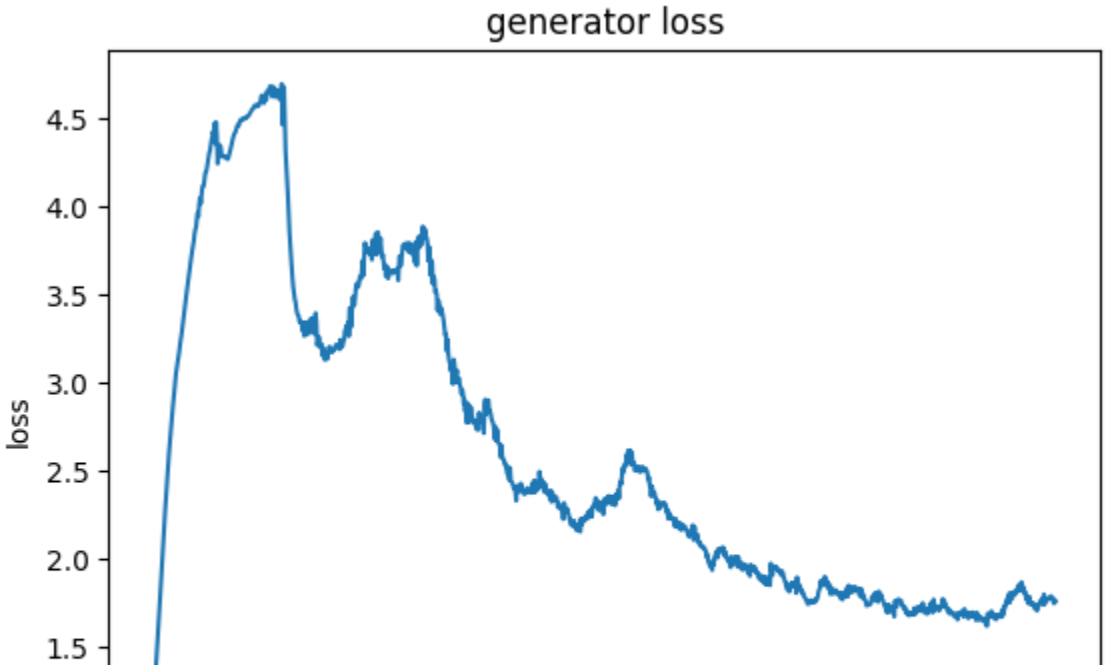

generator loss
4.5
4.0

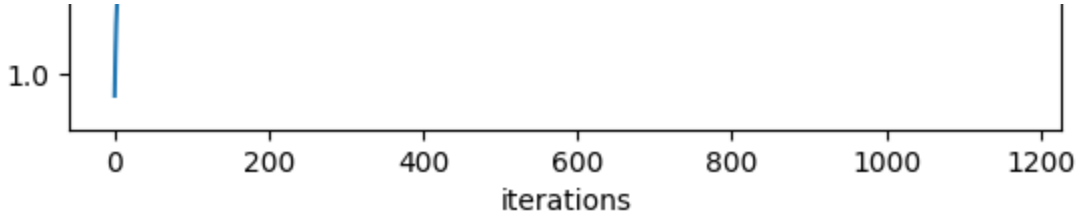



```
Iteration 400/9750: dis loss = 0.4015, gen loss = 2.2841
Iteration 500/9750: dis loss = 0.8298, gen loss = 1.9793
Iteration 600/9750: dis loss = 1.1969, gen loss = 3.5656
Iteration 700/9750: dis loss = 0.8285, gen loss = 3.0505
Iteration 800/9750: dis loss = 0.6558, gen loss = 0.9602
Iteration 900/9750: dis loss = 0.9028, gen loss = 2.6846
Iteration 1000/9750: dis loss = 1.0068, gen loss = 1.7632
Iteration 1100/9750: dis loss = 0.7844, gen loss = 1.3334
```

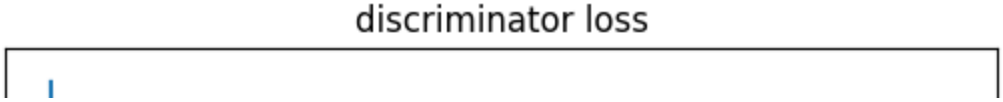

250
0
50
100
150
200
250

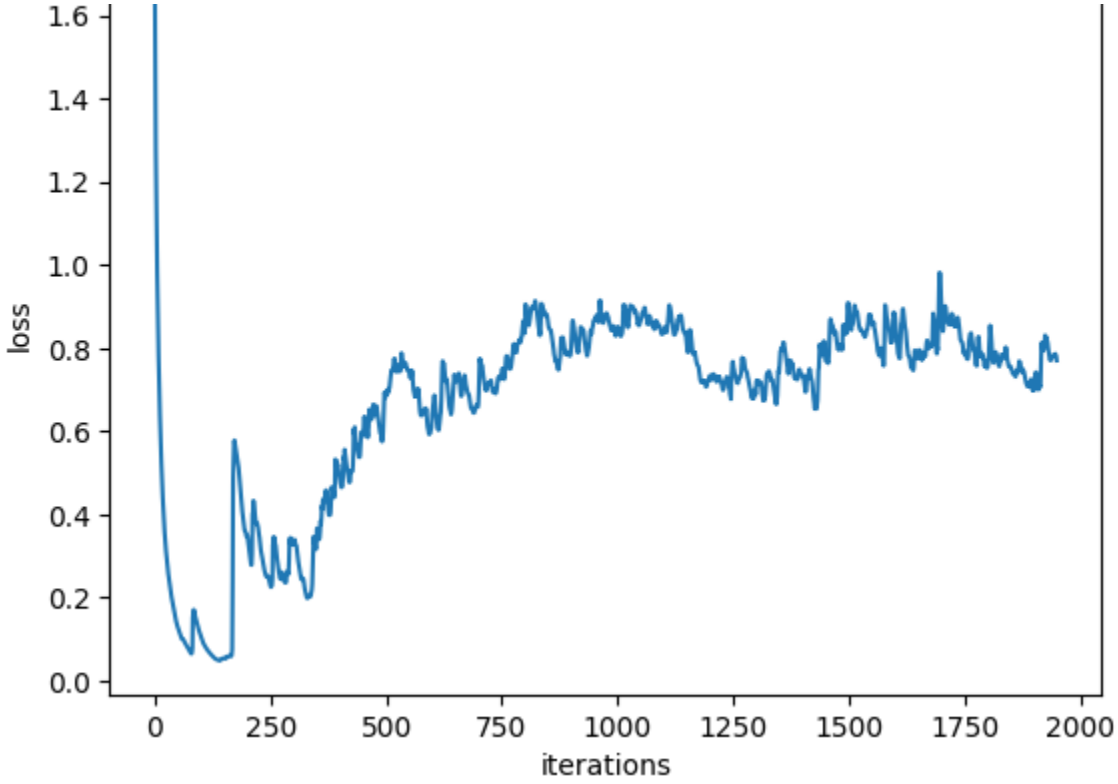

discriminator loss
loss
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
1200
iterations

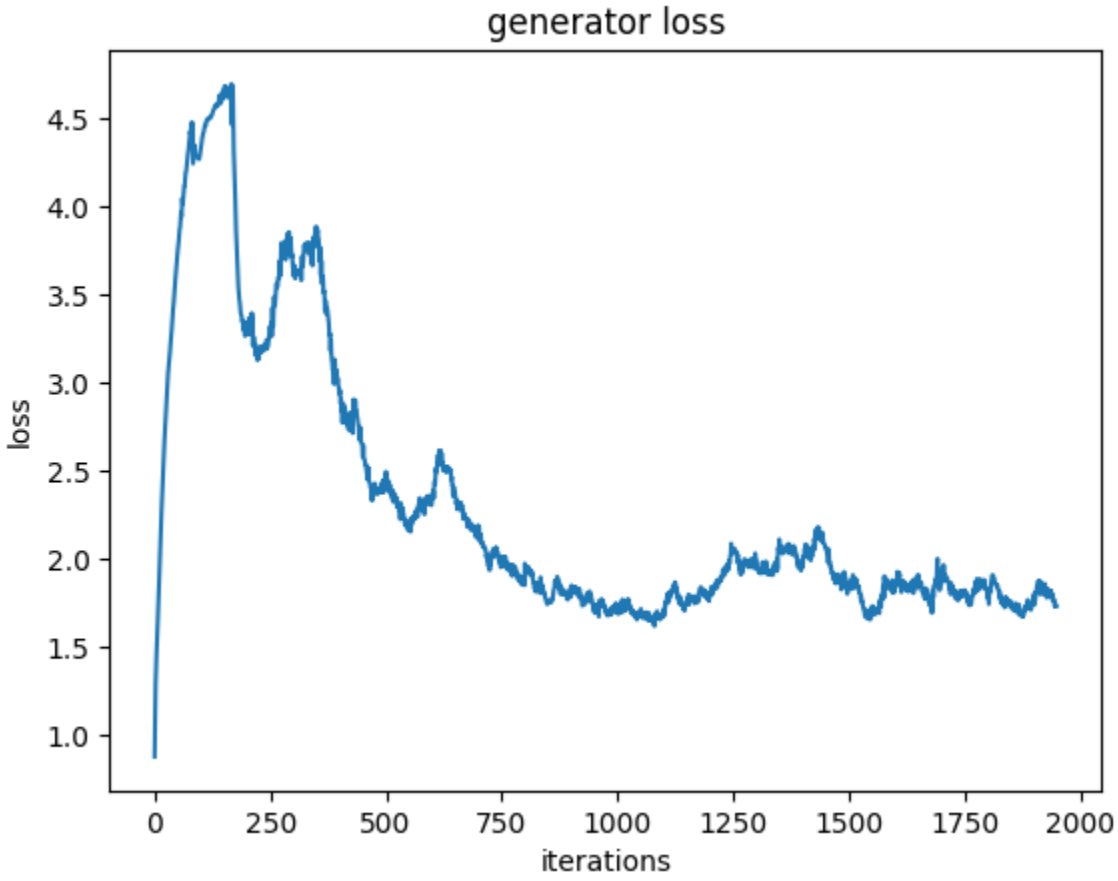

generator loss
loss
4.5
4.0
3.5
3.0
2.5
2.0
1.5




```
Iteration 1200/9750: dis loss = 0.8884, gen loss = 2.6926
Iteration 1300/9750: dis loss = 0.7974, gen loss = 1.5016
Iteration 1400/9750: dis loss = 0.7543, gen loss = 1.6699
Iteration 1500/9750: dis loss = 0.8747, gen loss = 1.7004
Iteration 1600/9750: dis loss = 0.8606, gen loss = 2.5302
Iteration 1700/9750: dis loss = 0.4576, gen loss = 2.1098
Iteration 1800/9750: dis loss = 0.7656, gen loss = 1.4283
Iteration 1900/9750: dis loss = 0.6376, gen loss = 1.8482
```

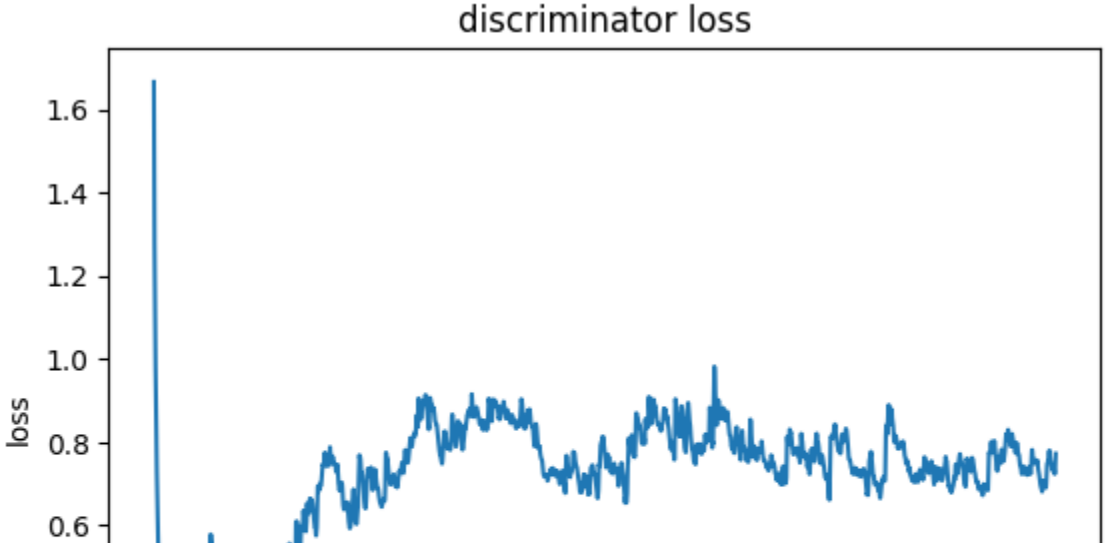


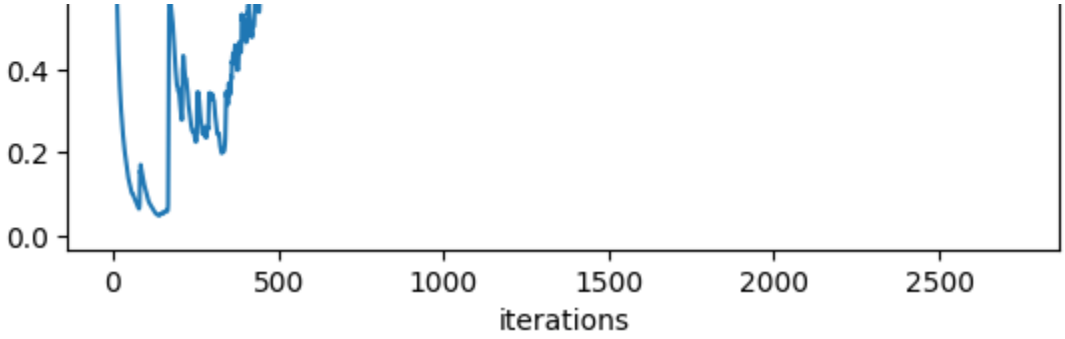


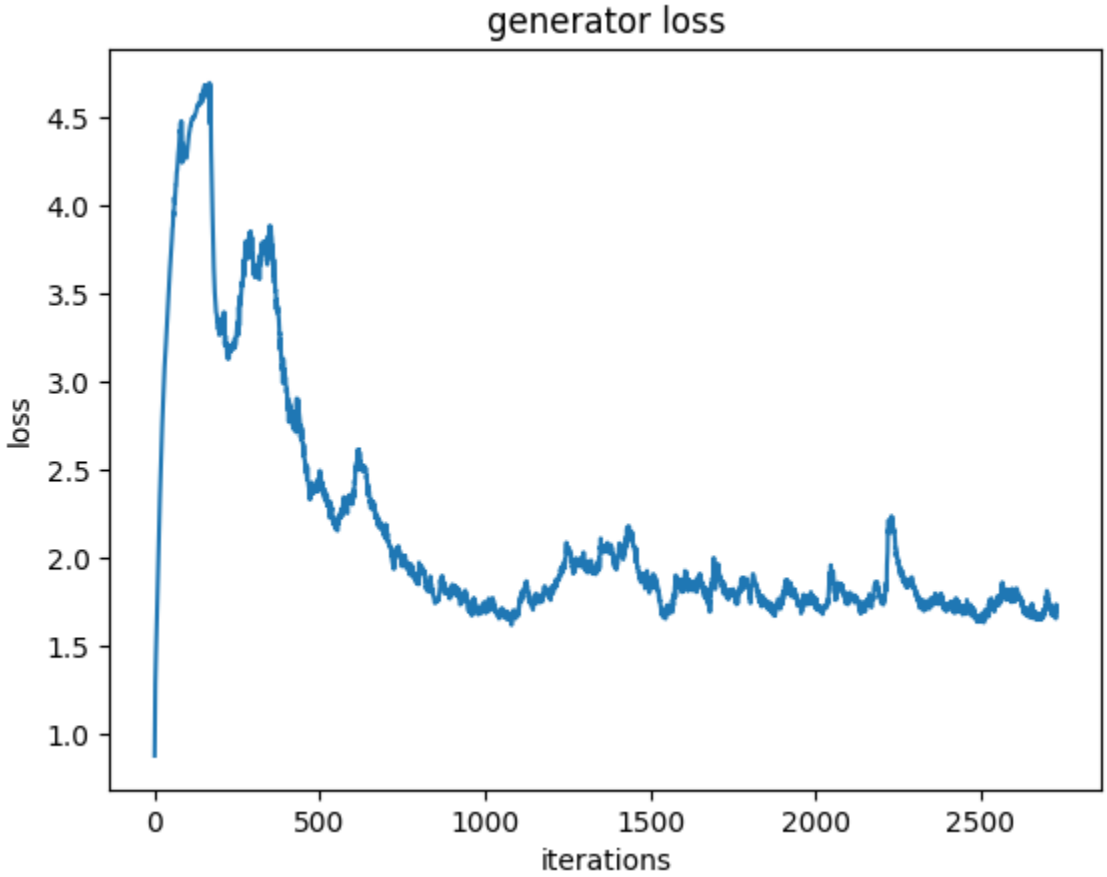



```
Iteration 2000/9750: dis loss = 0.7462, gen loss = 1.2248
Iteration 2100/9750: dis loss = 0.6294, gen loss = 1.4800
Iteration 2200/9750: dis loss = 0.7847, gen loss = 2.4640
Iteration 2300/9750: dis loss = 0.6061, gen loss = 2.1480
Iteration 2400/9750: dis loss = 0.7626, gen loss = 2.2608
Iteration 2500/9750: dis loss = 0.7536, gen loss = 1.5006
```

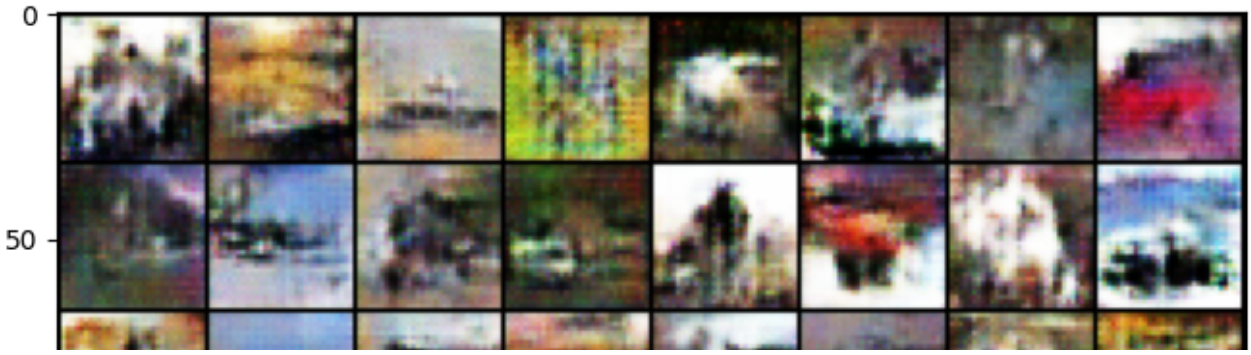
```
Iteration 2600/9750: dis loss = 0.7621, gen loss = 1.2335
Iteration 2700/9750: dis loss = 0.5984, gen loss = 1.3180
```

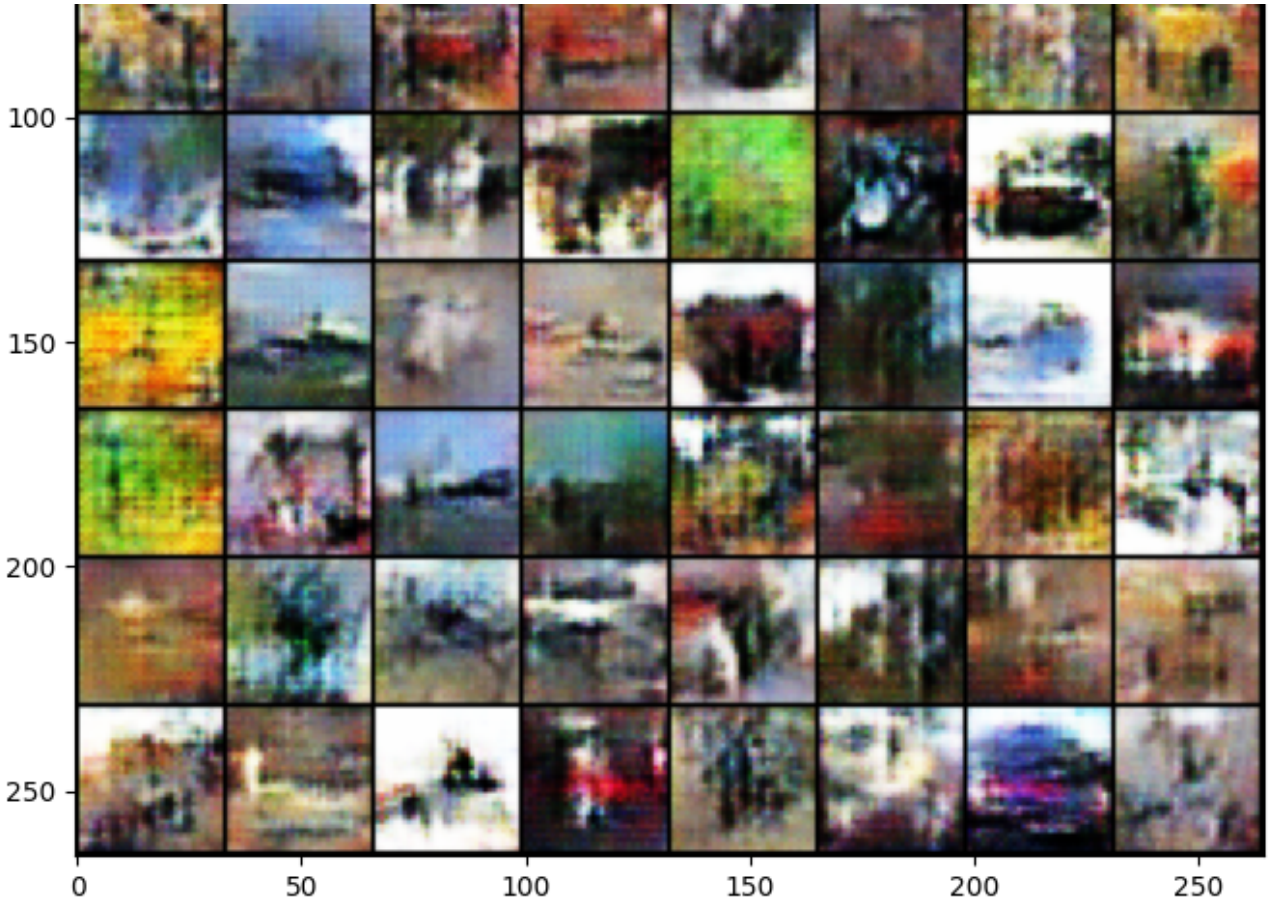


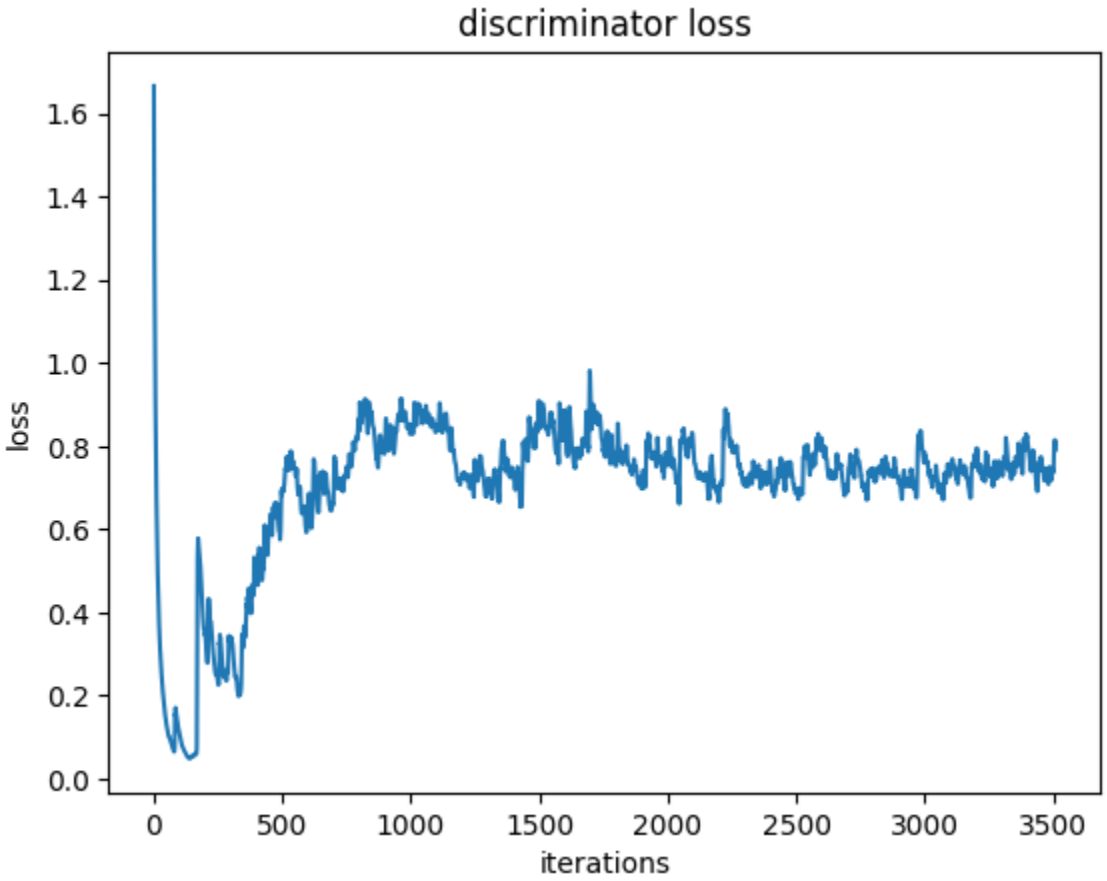


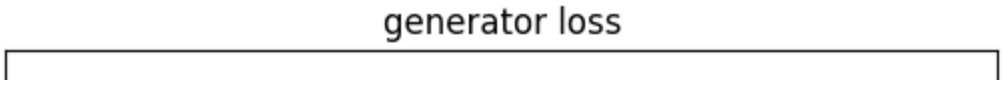


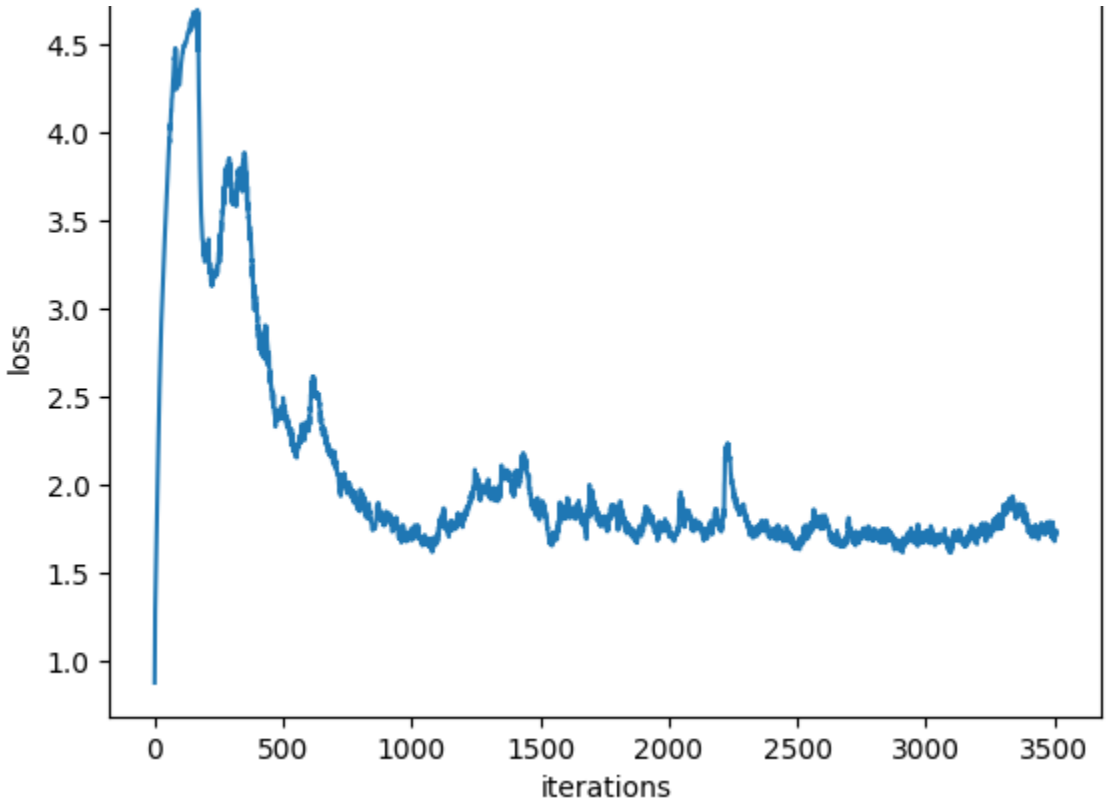

```
Iteration 2800/9750: dis loss = 0.6459, gen loss = 2.0577
Iteration 2900/9750: dis loss = 0.6198, gen loss = 1.0890
Iteration 3000/9750: dis loss = 0.8019, gen loss = 1.8013
Iteration 3100/9750: dis loss = 0.7010, gen loss = 0.8456
Iteration 3200/9750: dis loss = 0.7127, gen loss = 2.0951
Iteration 3300/9750: dis loss = 0.6665, gen loss = 2.1296
Iteration 3400/9750: dis loss = 0.6289, gen loss = 1.8013
Iteration 3500/9750: dis loss = 0.8316, gen loss = 1.1769
```



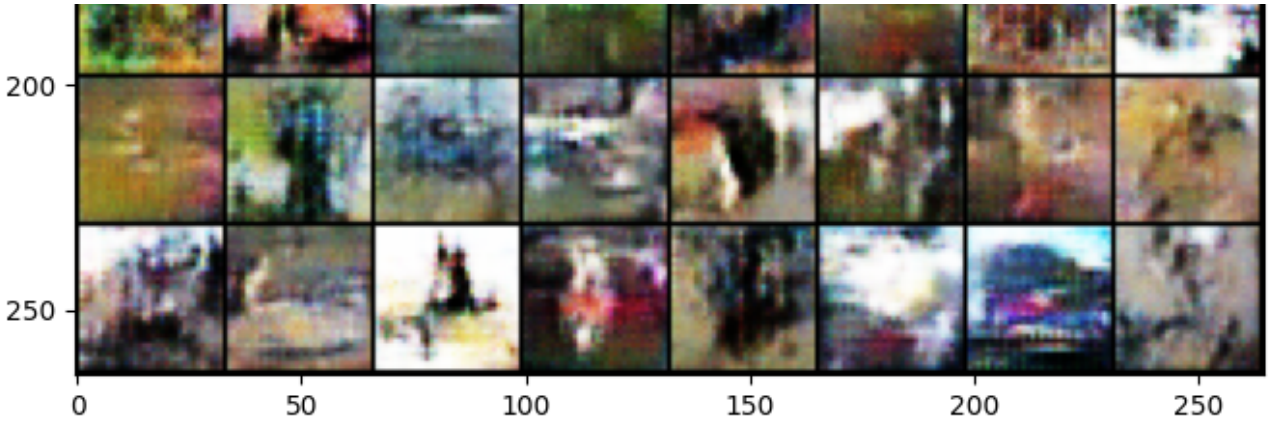

100
150
200
250
0
50
100
150
200
250

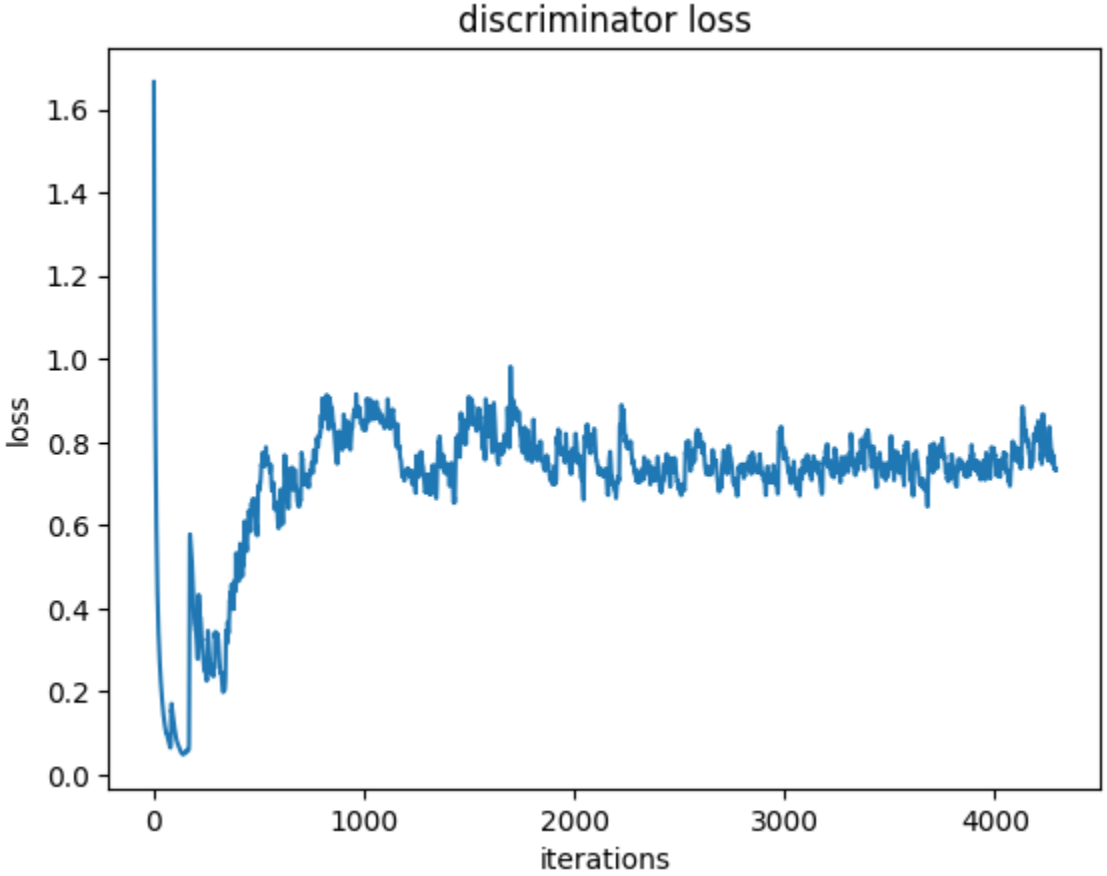

discriminator loss
loss
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
500
1000
1500
2000
2500
3000
3500
iterations

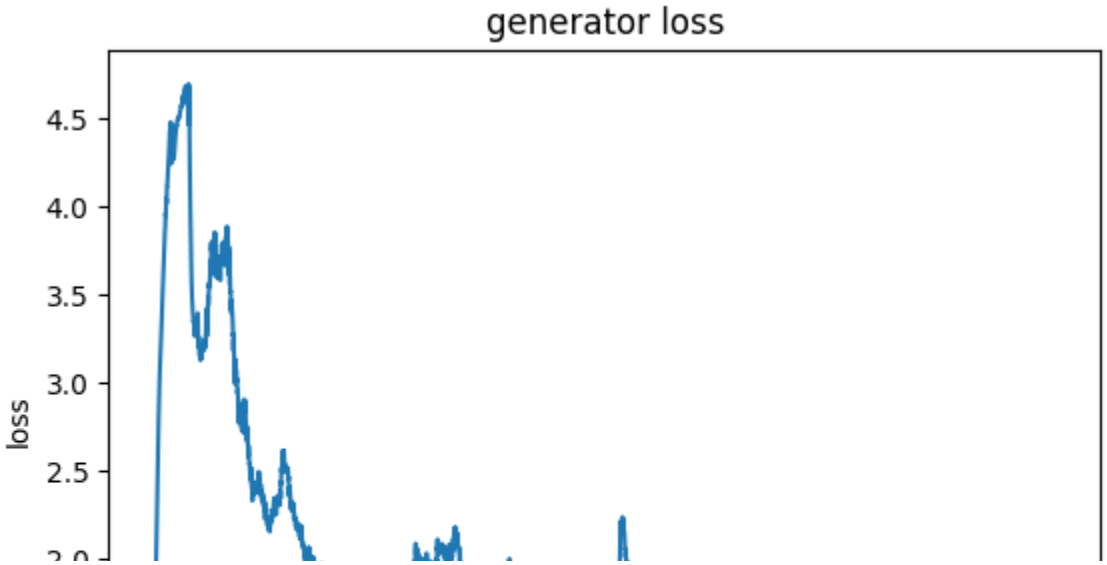

generator loss

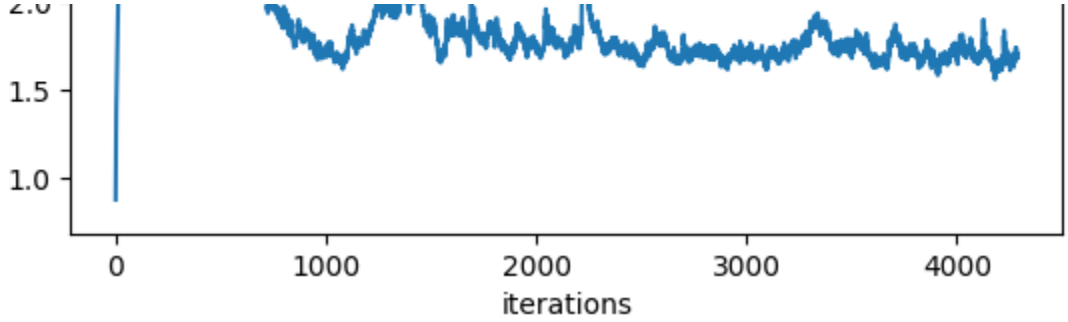



```
Iteration 3600/9750: dis loss = 0.7745, gen loss = 1.4667
Iteration 3700/9750: dis loss = 0.7870, gen loss = 1.6688
Iteration 3800/9750: dis loss = 0.6015, gen loss = 1.8530
Iteration 3900/9750: dis loss = 0.6256, gen loss = 1.1920
Iteration 4000/9750: dis loss = 0.7623, gen loss = 0.9299
Iteration 4100/9750: dis loss = 0.7966, gen loss = 2.0440
Iteration 4200/9750: dis loss = 1.0568, gen loss = 2.5500
```

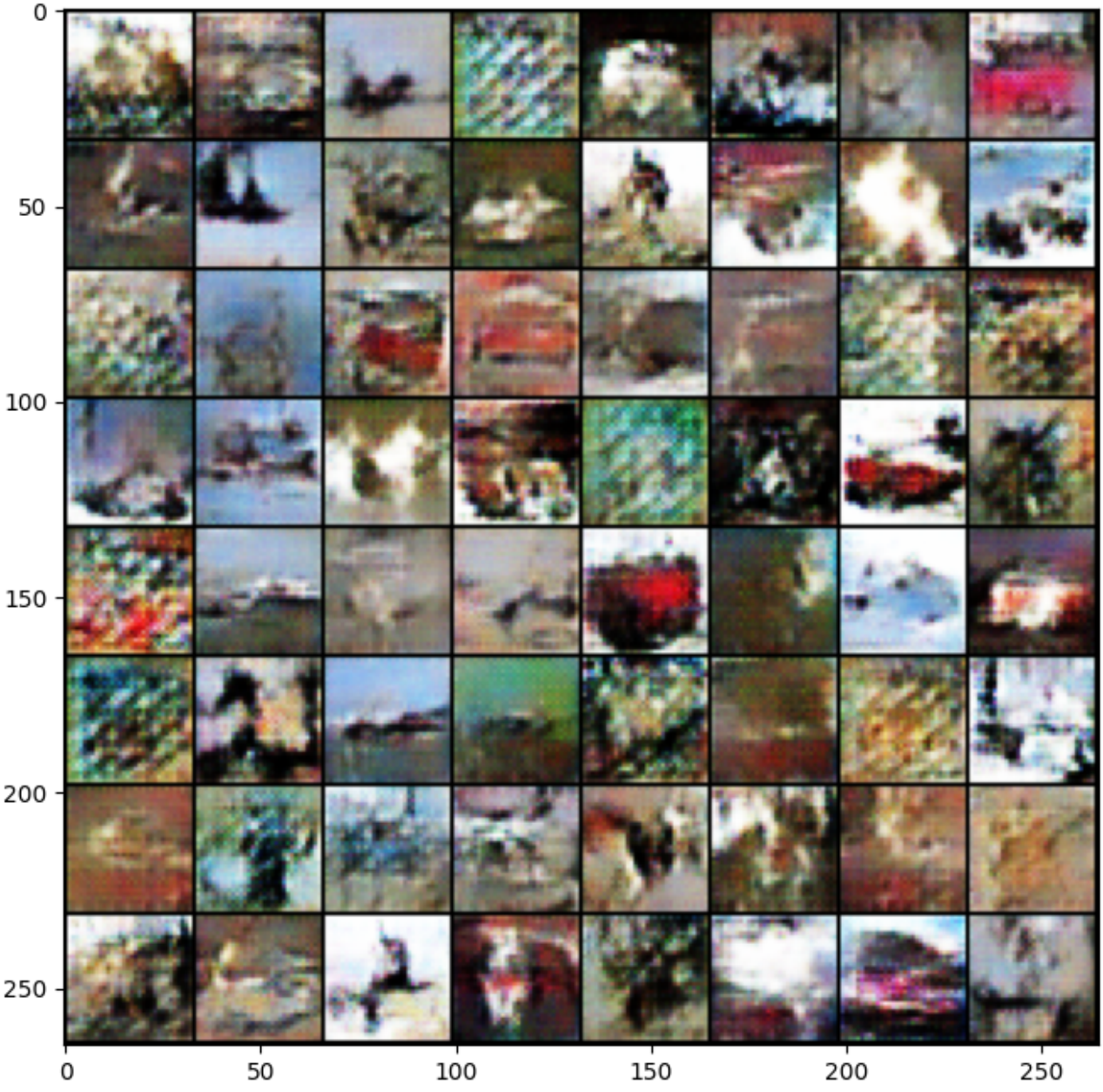

200
250
0
50
100
150
200
250

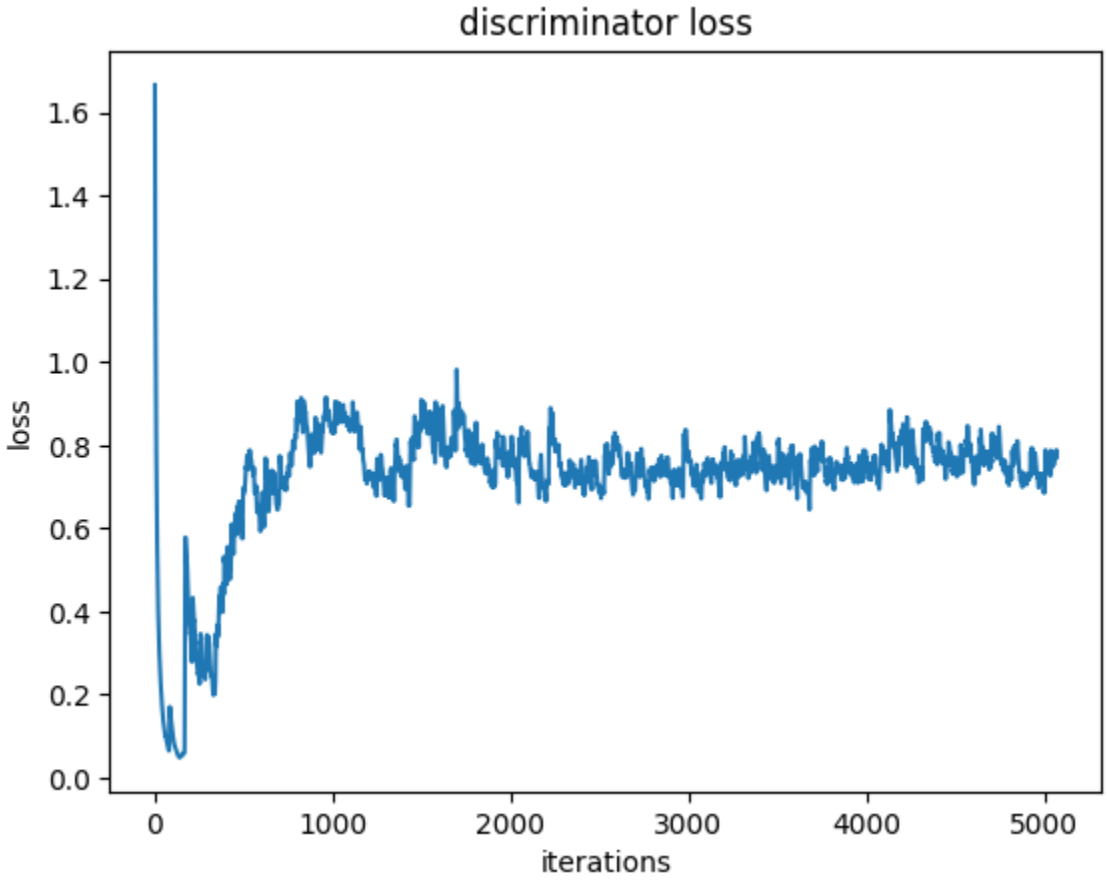

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
1000
2000
3000
4000

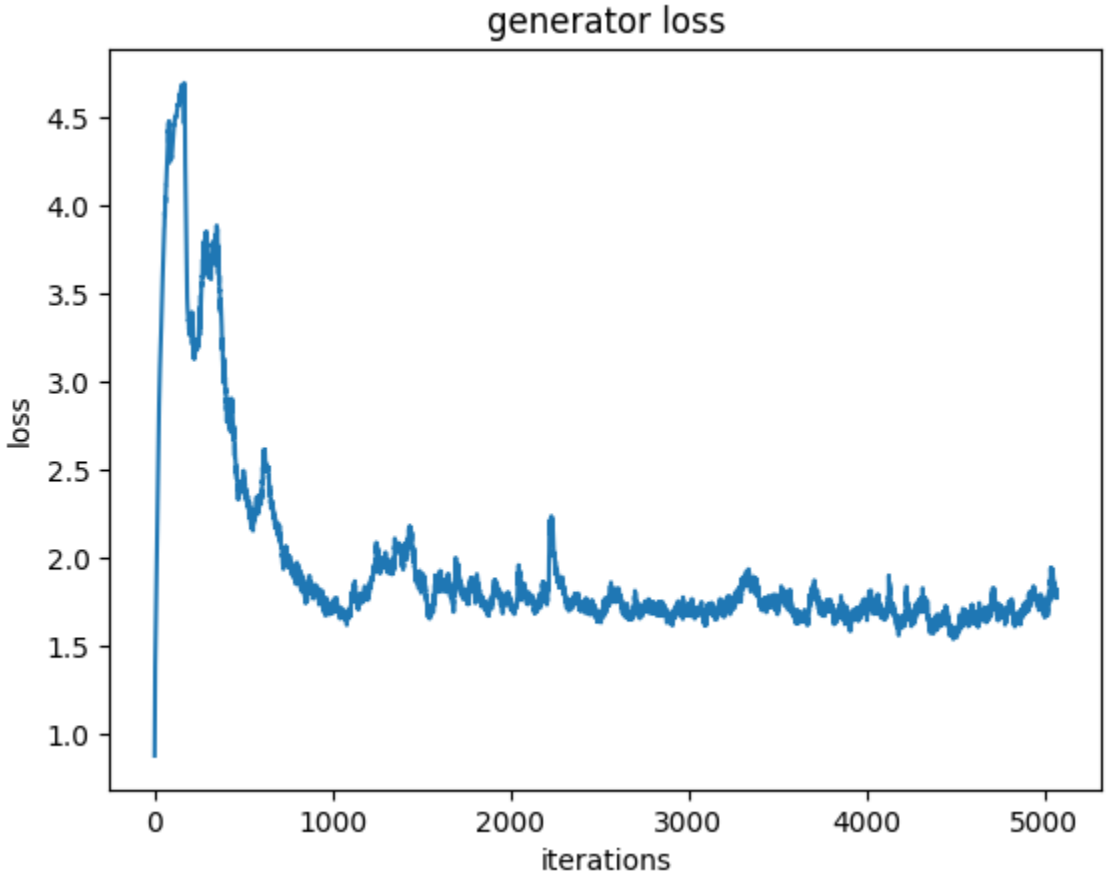

generator loss
loss
2.5
3.0
3.5
4.0
4.5




```
Iteration 4300/9750: dis loss = 0.6268, gen loss = 1.0057
Iteration 4400/9750: dis loss = 1.2545, gen loss = 3.3571
Iteration 4500/9750: dis loss = 0.6692, gen loss = 1.2501
Iteration 4600/9750: dis loss = 0.6717, gen loss = 1.7159
Iteration 4700/9750: dis loss = 0.8642, gen loss = 1.3993
Iteration 4800/9750: dis loss = 0.5799, gen loss = 1.9871
Iteration 4900/9750: dis loss = 0.5572, gen loss = 1.7354
Iteration 5000/9750: dis loss = 0.7787, gen loss = 2.0611
```

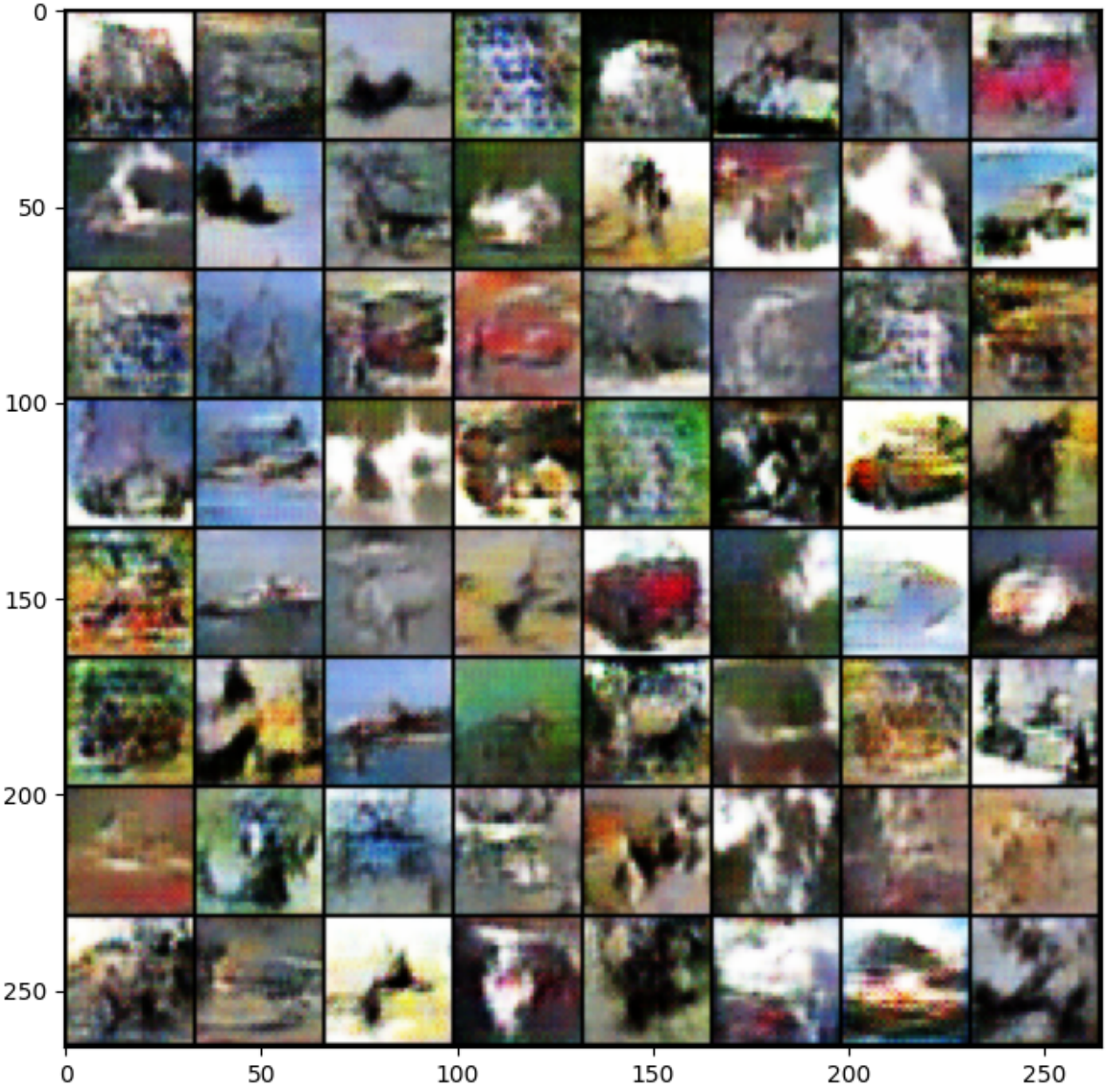


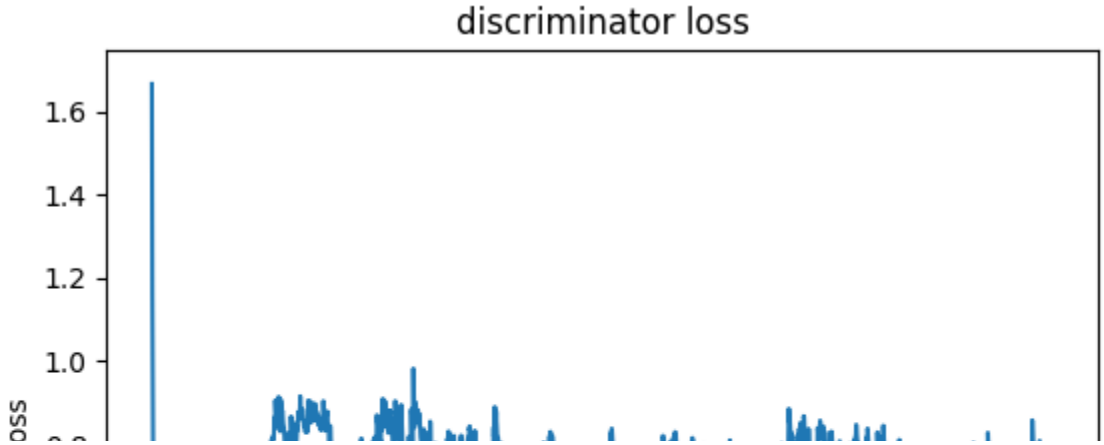


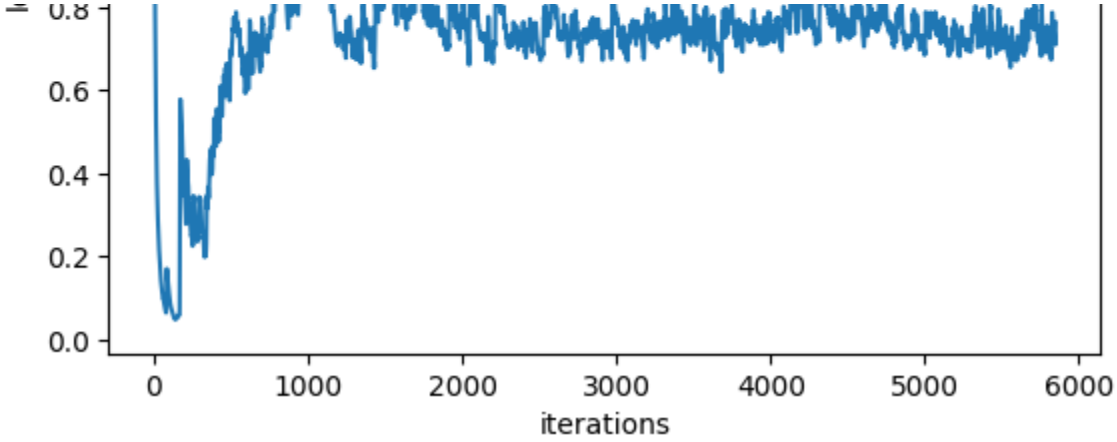


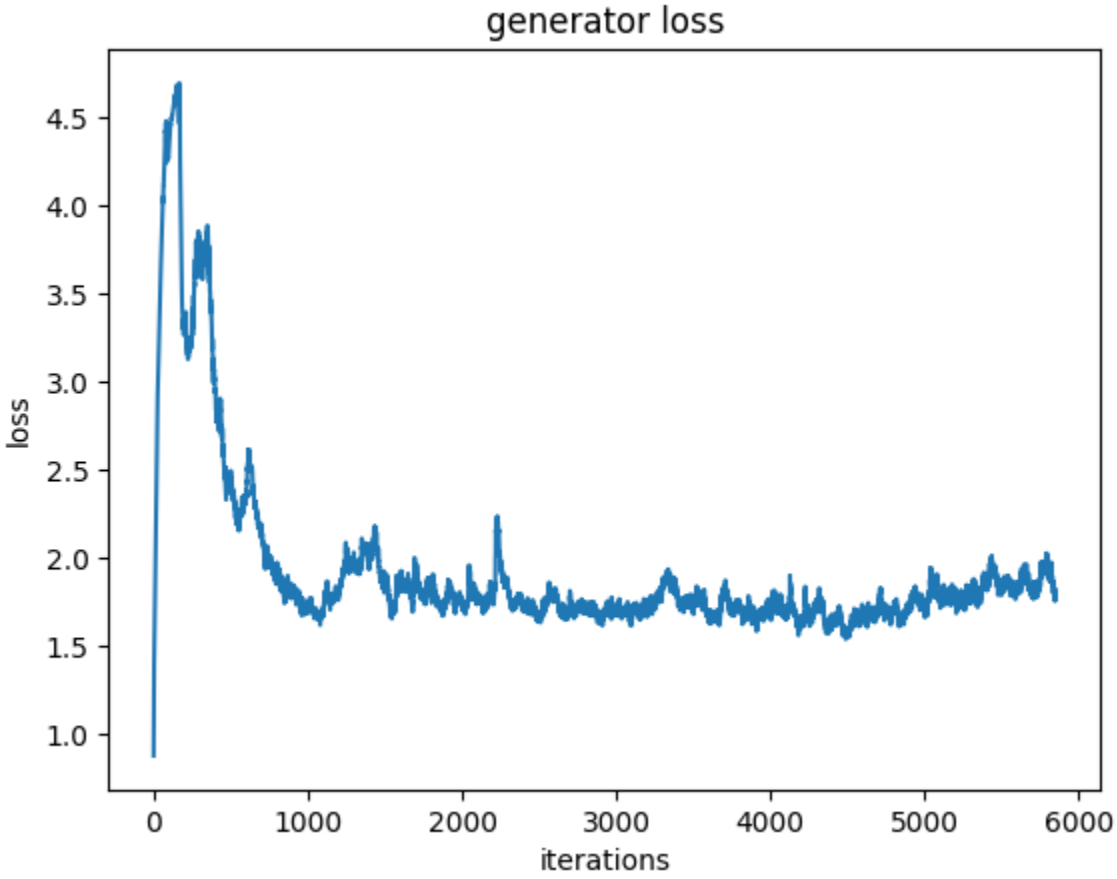

```
Iteration 5100/9750: dis loss = 0.6077, gen loss = 2.2222
Iteration 5200/9750: dis loss = 0.6156, gen loss = 0.8446
Iteration 5300/9750: dis loss = 0.9178, gen loss = 2.9327
```

```
Iteration 5400/9750: dis loss = 0.7925, gen loss = 2.6703
Iteration 5500/9750: dis loss = 0.5983, gen loss = 1.8124
Iteration 5600/9750: dis loss = 0.9145, gen loss = 2.7378
Iteration 5700/9750: dis loss = 0.9276, gen loss = 2.0779
Iteration 5800/9750: dis loss = 0.5240, gen loss = 1.9392
```

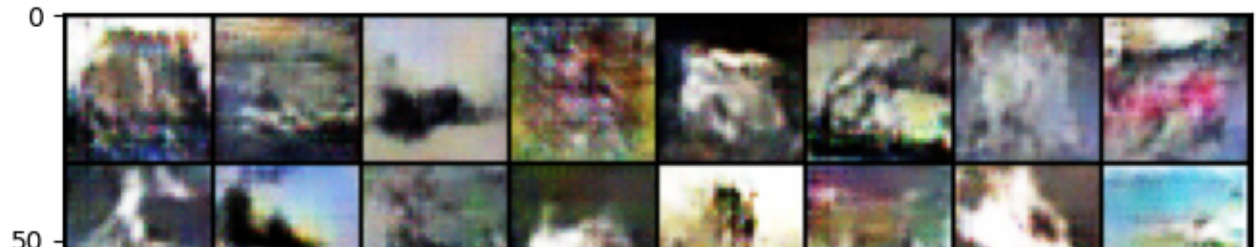


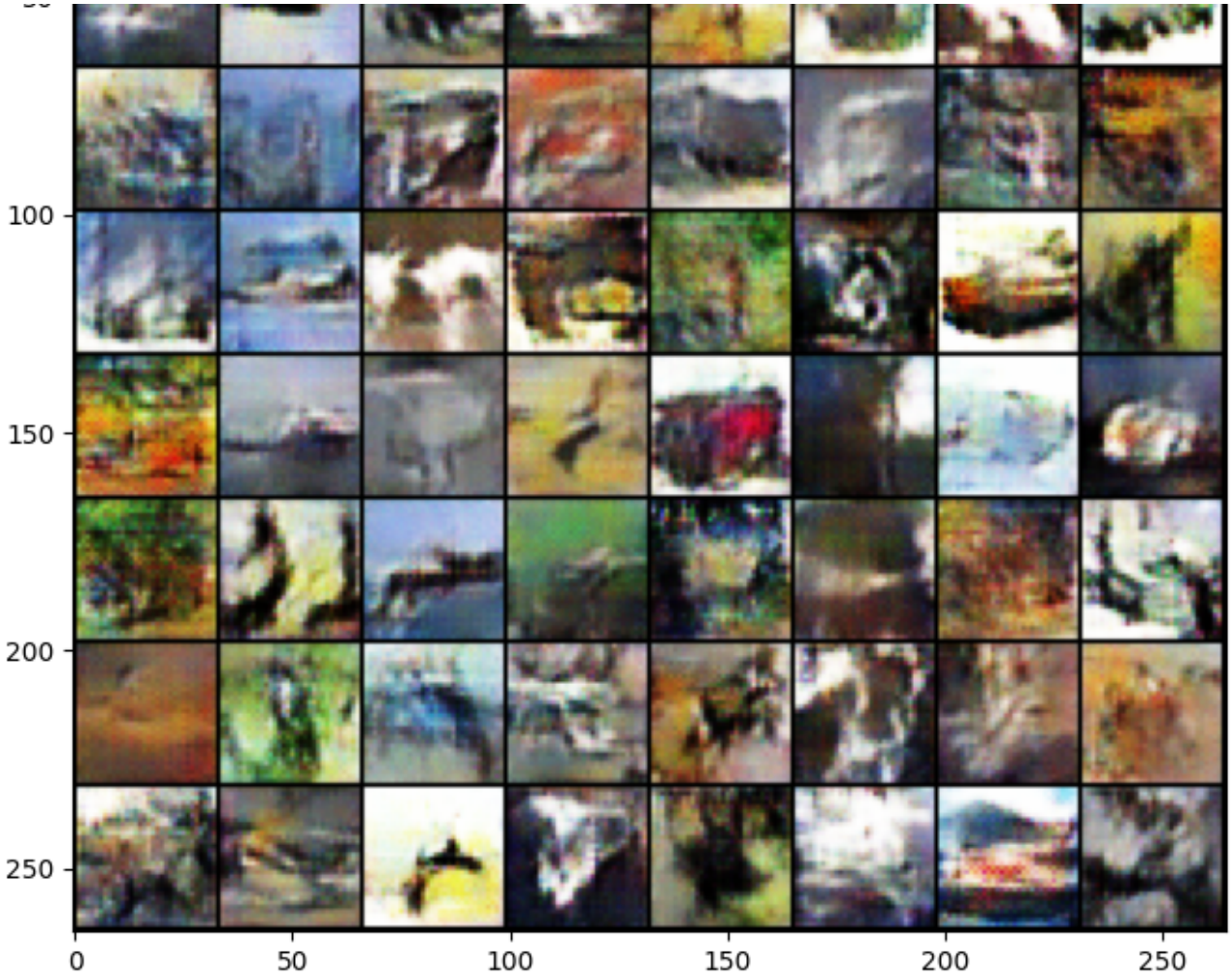


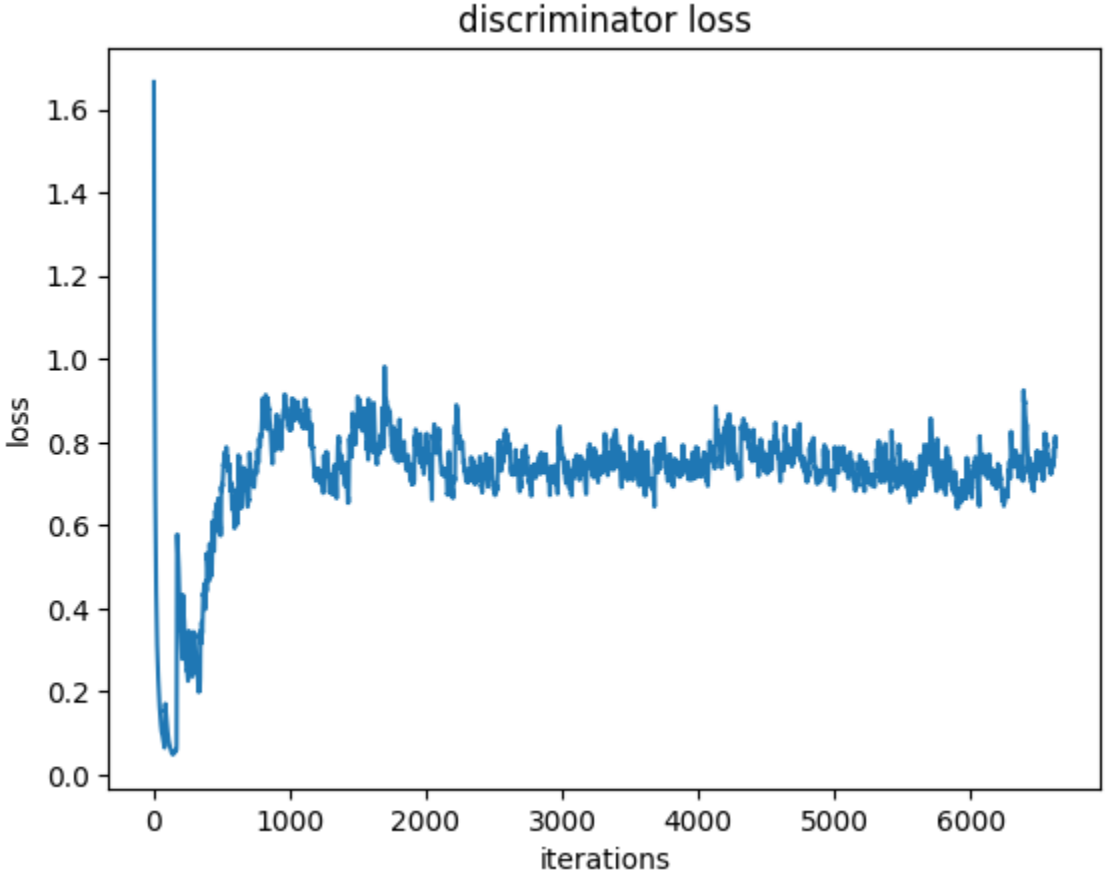


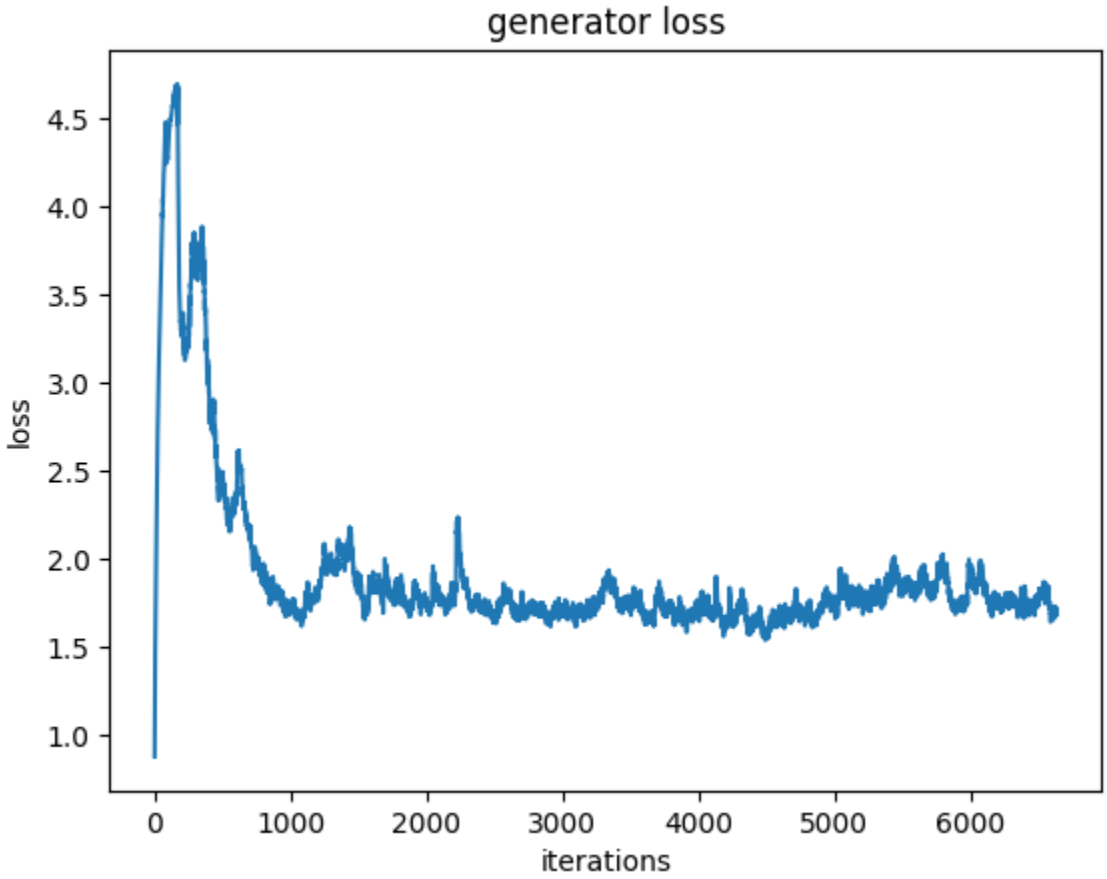



```
Iteration 5900/9750: dis loss = 0.4844, gen loss = 1.9859
Iteration 6000/9750: dis loss = 0.3467, gen loss = 3.0412
Iteration 6100/9750: dis loss = 0.9444, gen loss = 0.6962
Iteration 6200/9750: dis loss = 0.8474, gen loss = 0.9559
Iteration 6300/9750: dis loss = 1.3749, gen loss = 0.6801
Iteration 6400/9750: dis loss = 0.6859, gen loss = 2.3810
Iteration 6500/9750: dis loss = 0.7662, gen loss = 2.2620
Iteration 6600/9750: dis loss = 0.9121, gen loss = 0.9500
```

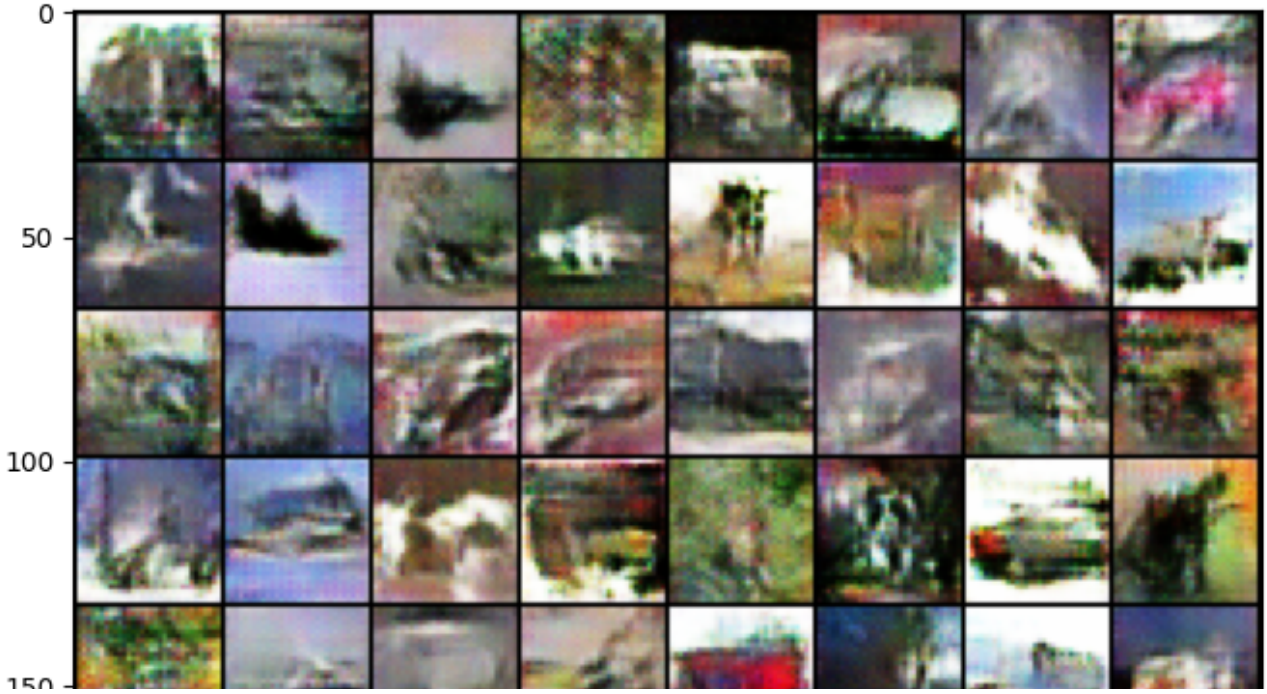


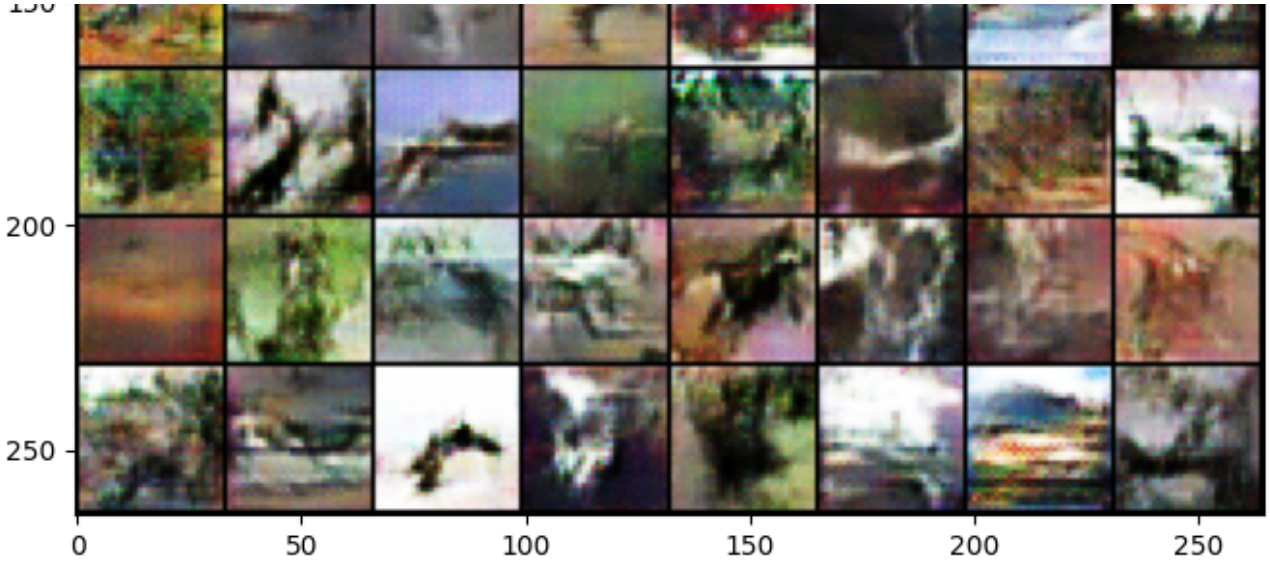

100
150
200
250
0
50
100
150
200
250

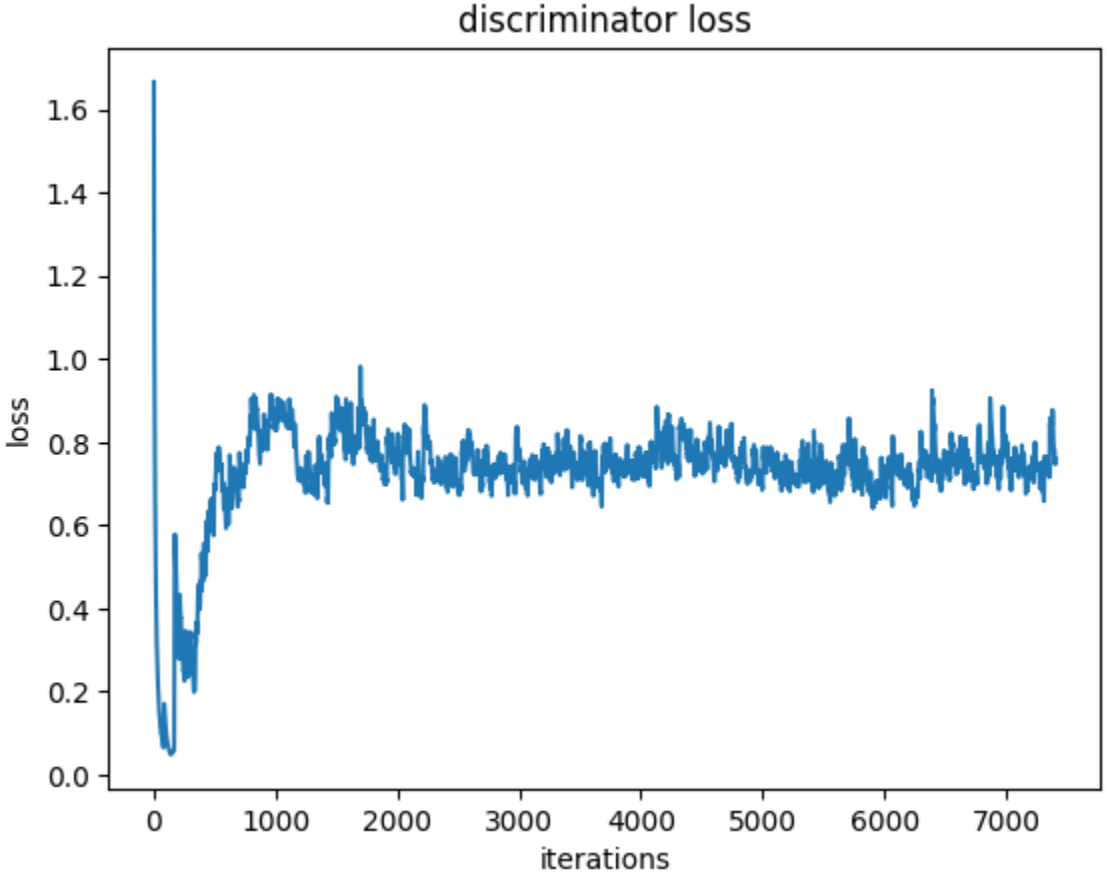

discriminator loss
loss
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
1000
2000
3000
4000
5000
6000
iterations

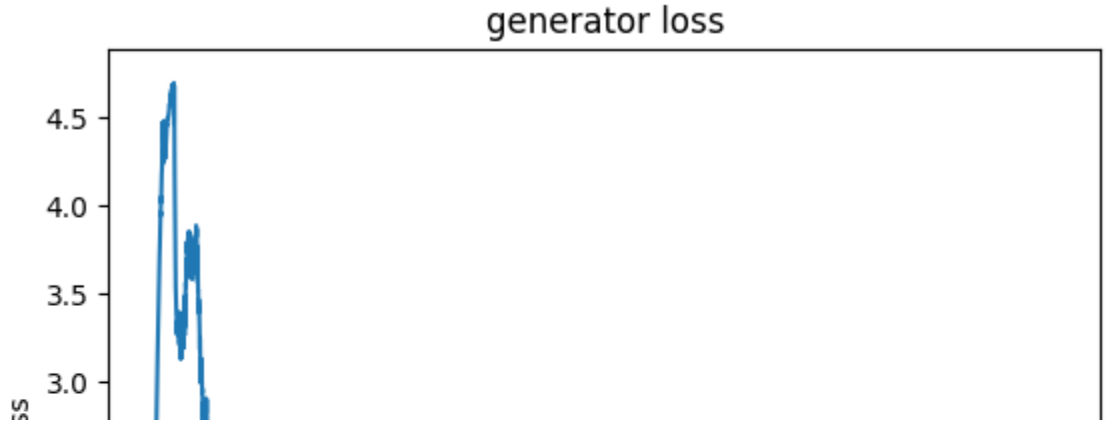


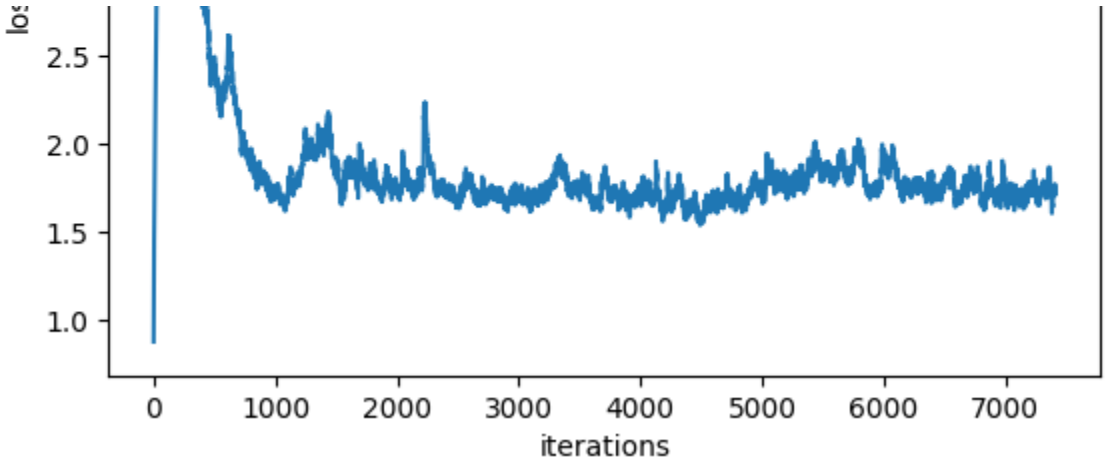

```
Iteration 6700/9750: dis loss = 1.0009, gen loss = 2.7453
Iteration 6800/9750: dis loss = 0.8027, gen loss = 1.3894
Iteration 6900/9750: dis loss = 0.6645, gen loss = 1.7960
Iteration 7000/9750: dis loss = 0.6459, gen loss = 1.9202
Iteration 7100/9750: dis loss = 0.5360, gen loss = 2.0978
Iteration 7200/9750: dis loss = 0.6935, gen loss = 1.2928
Iteration 7300/9750: dis loss = 0.8304, gen loss = 0.8284
Iteration 7400/9750: dis loss = 0.7023, gen loss = 1.1156
```



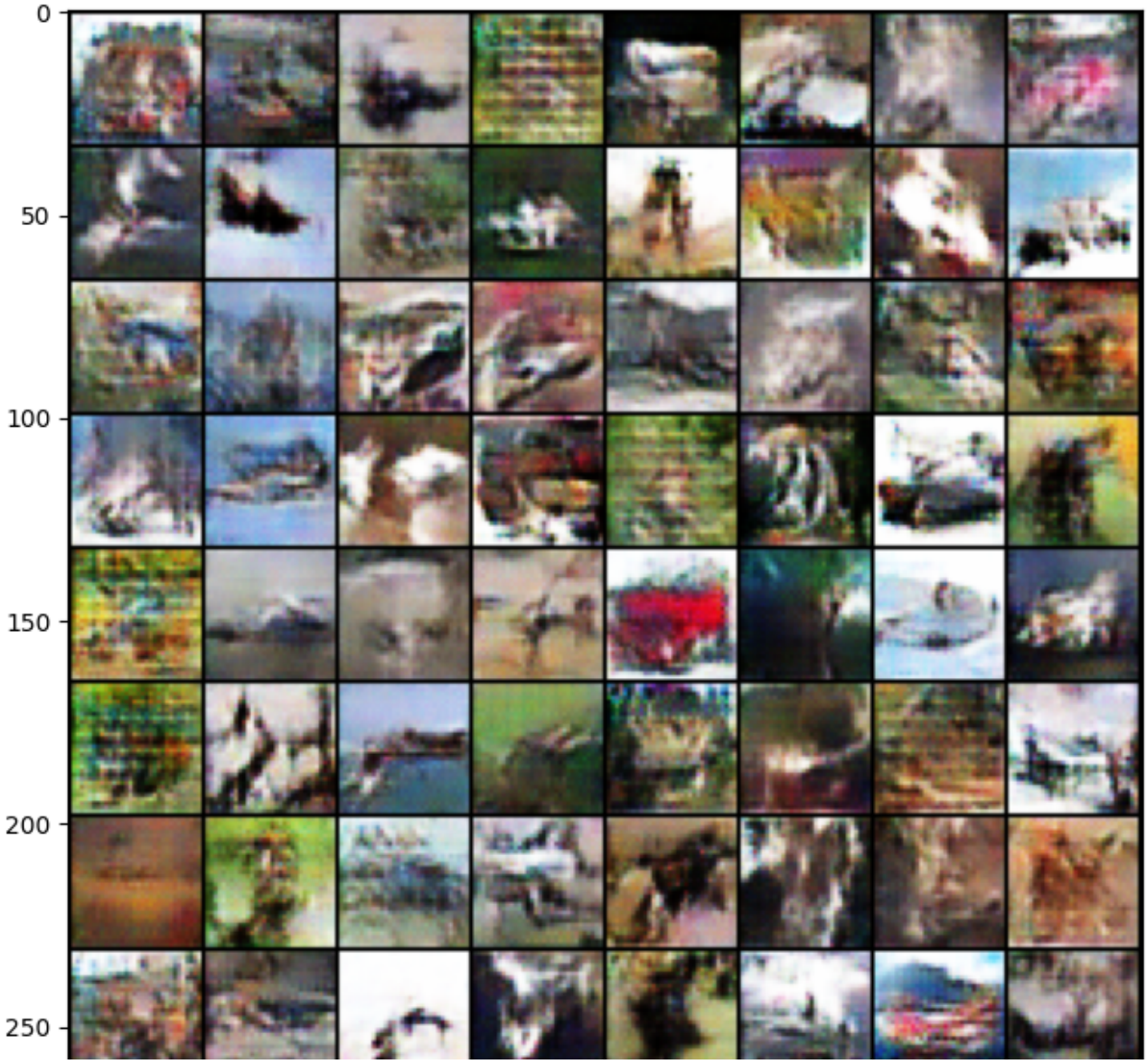

200
250
0
50
100
150
200
250


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

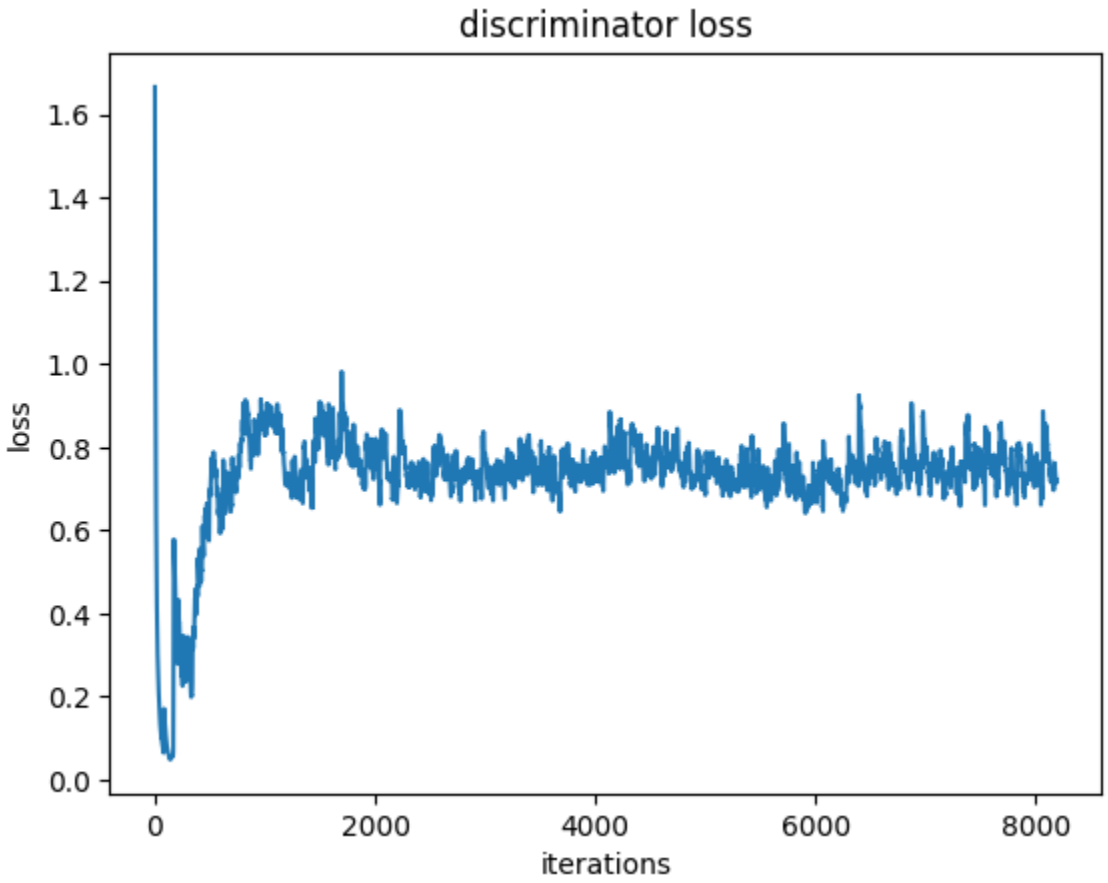

generator loss
4.5
4.0
3.5
3.0

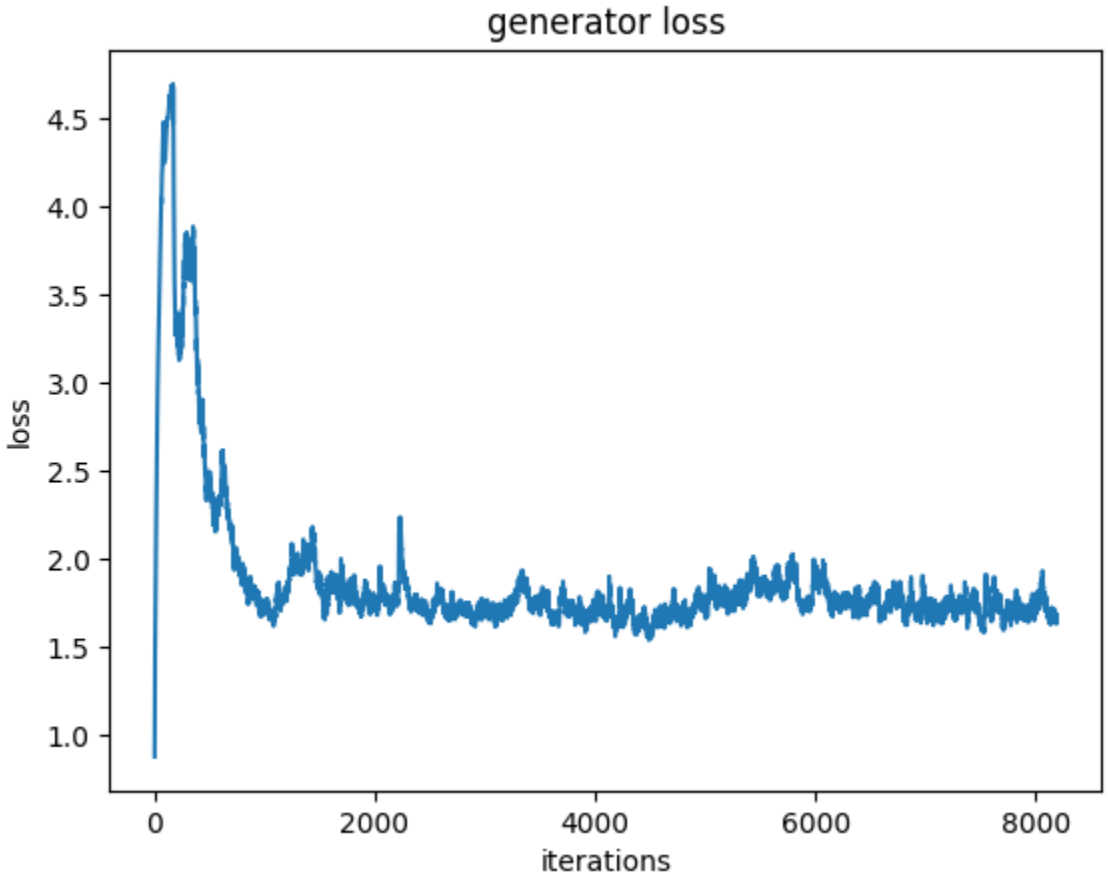



```
Iteration 7500/9750: dis loss = 0.6075, gen loss = 1.8427
Iteration 7600/9750: dis loss = 0.7655, gen loss = 1.3748
Iteration 7700/9750: dis loss = 0.7104, gen loss = 0.7177
Iteration 7800/9750: dis loss = 0.6270, gen loss = 1.7640
Iteration 7900/9750: dis loss = 0.8775, gen loss = 1.3910
Iteration 8000/9750: dis loss = 0.7321, gen loss = 2.8143
Iteration 8100/9750: dis loss = 0.6320, gen loss = 1.4916
```

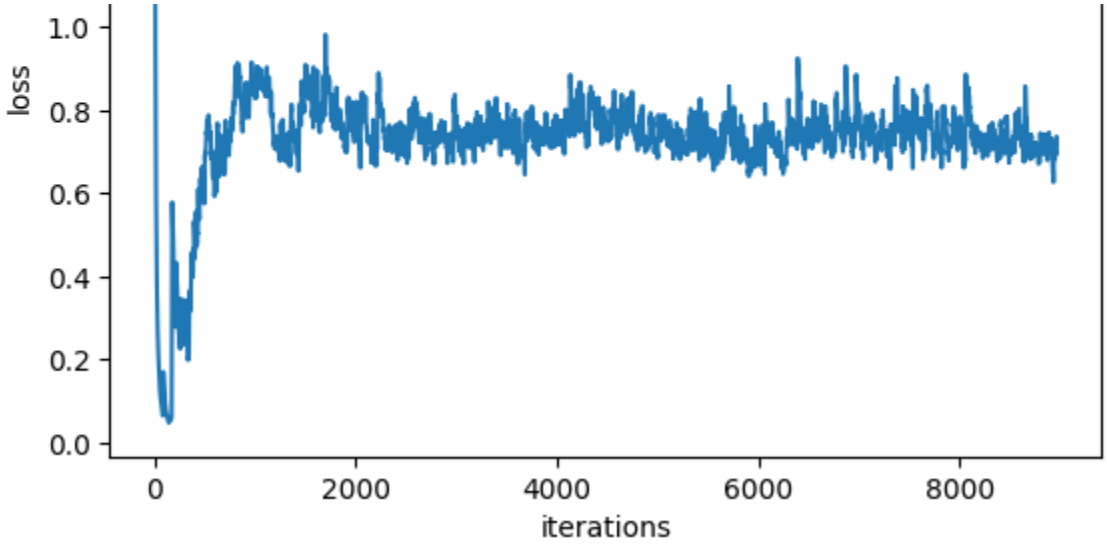


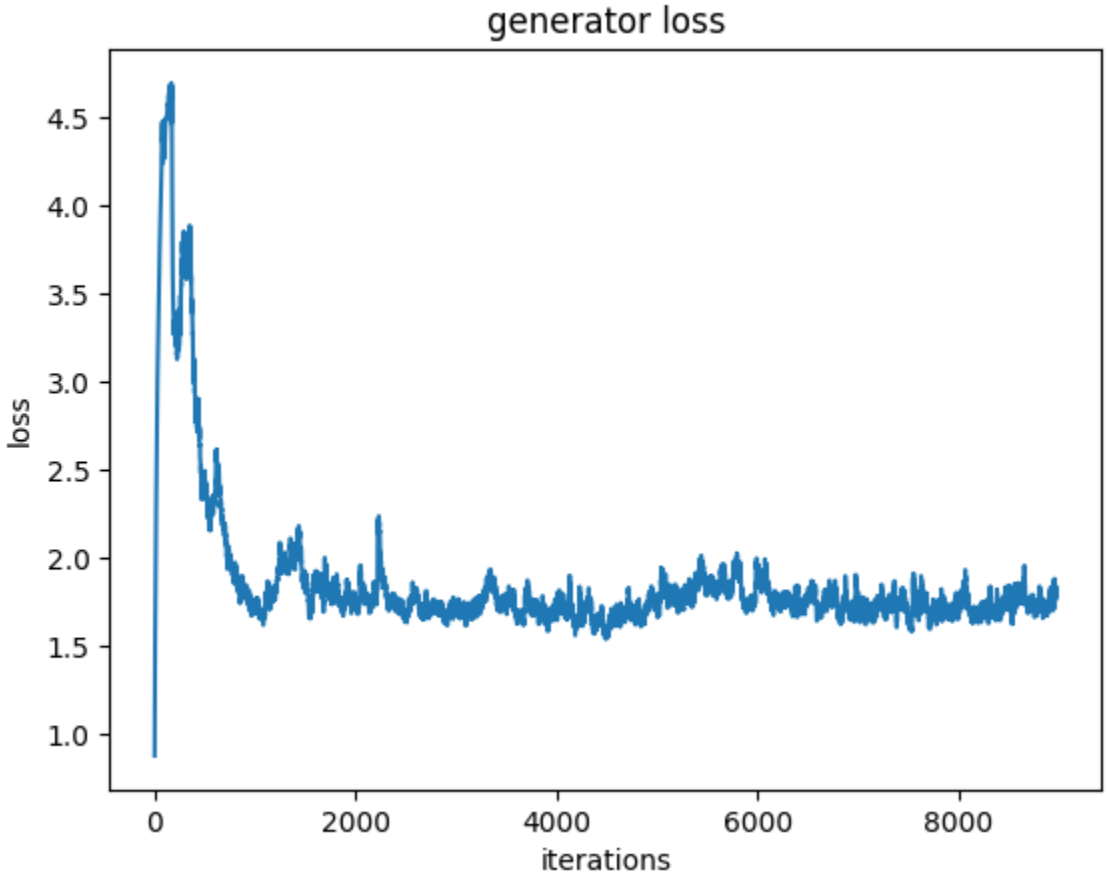

0
50
100
150
200
250


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

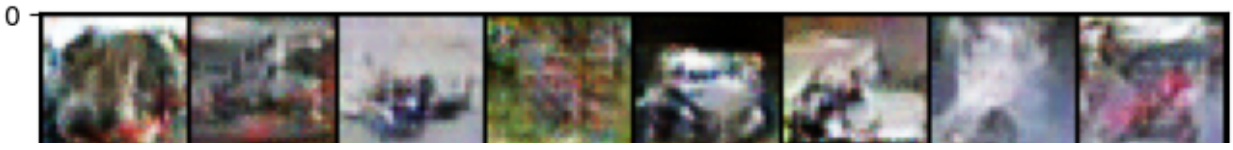

generator loss
loss
iterations
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
2000
4000
6000
8000

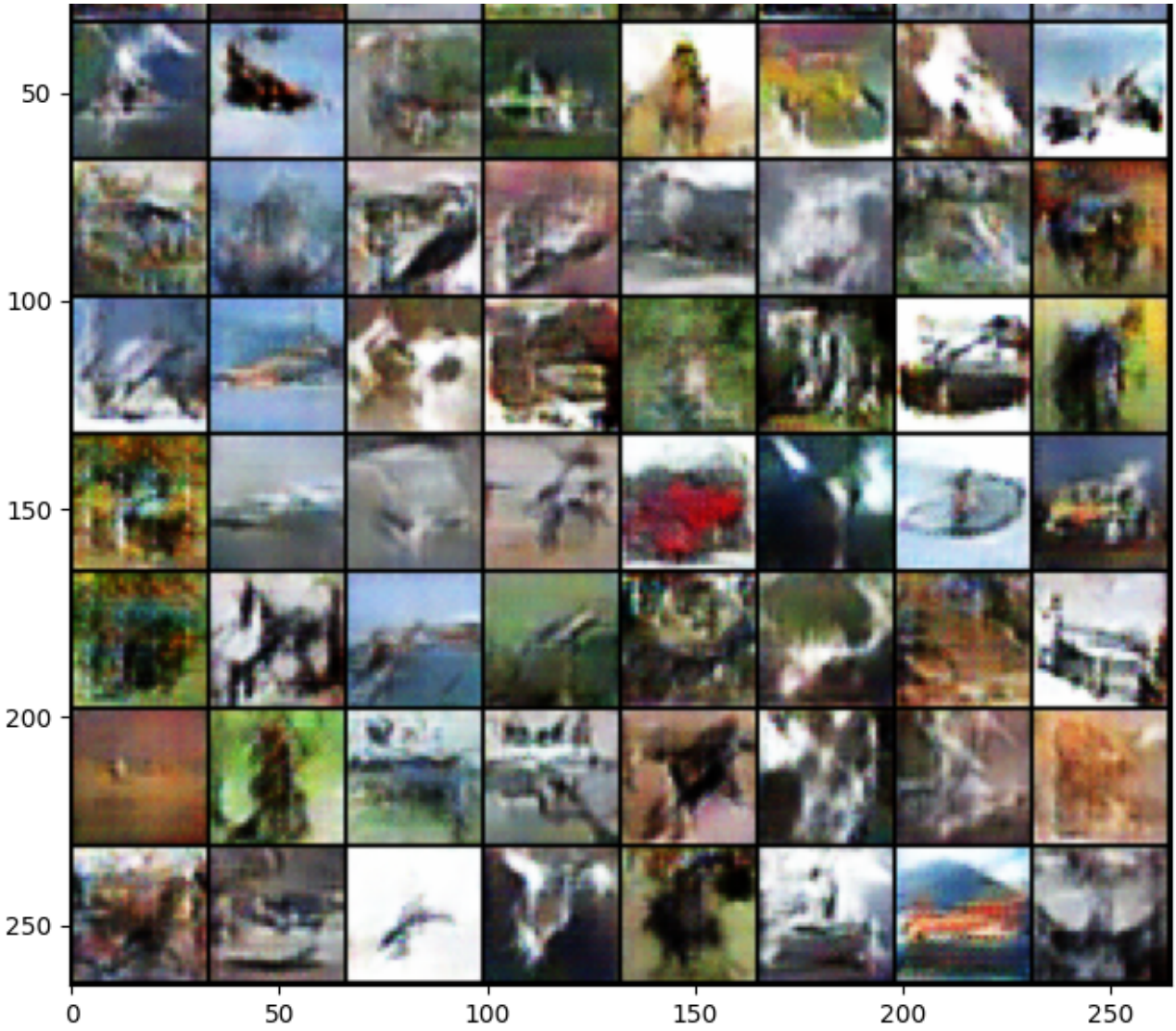
```
Iteration 8200/9750: dis loss = 0.8080, gen loss = 1.4374
Iteration 8300/9750: dis loss = 0.6036, gen loss = 2.0311
Iteration 8400/9750: dis loss = 0.7012, gen loss = 1.5088
Iteration 8500/9750: dis loss = 0.5569, gen loss = 1.6072
Iteration 8600/9750: dis loss = 0.6076, gen loss = 1.3672
Iteration 8700/9750: dis loss = 0.7972, gen loss = 1.4994
Iteration 8800/9750: dis loss = 0.9716, gen loss = 0.5957
Iteration 8900/9750: dis loss = 0.6692, gen loss = 2.4831
```

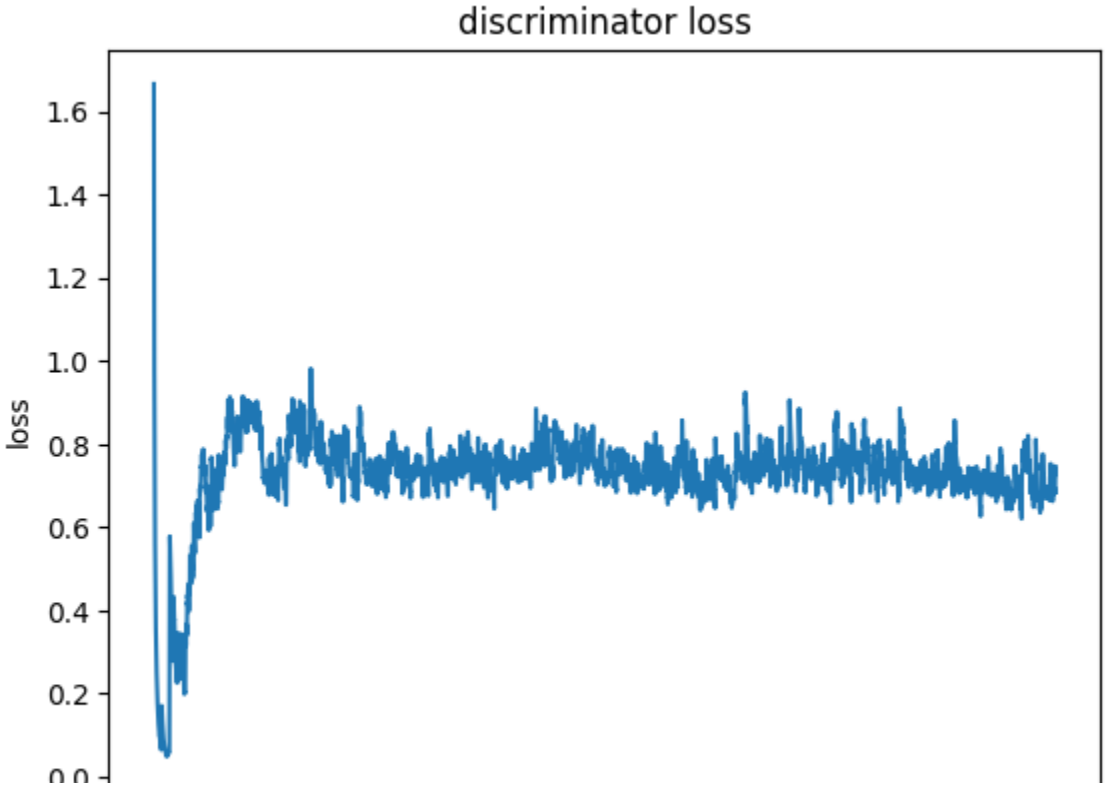


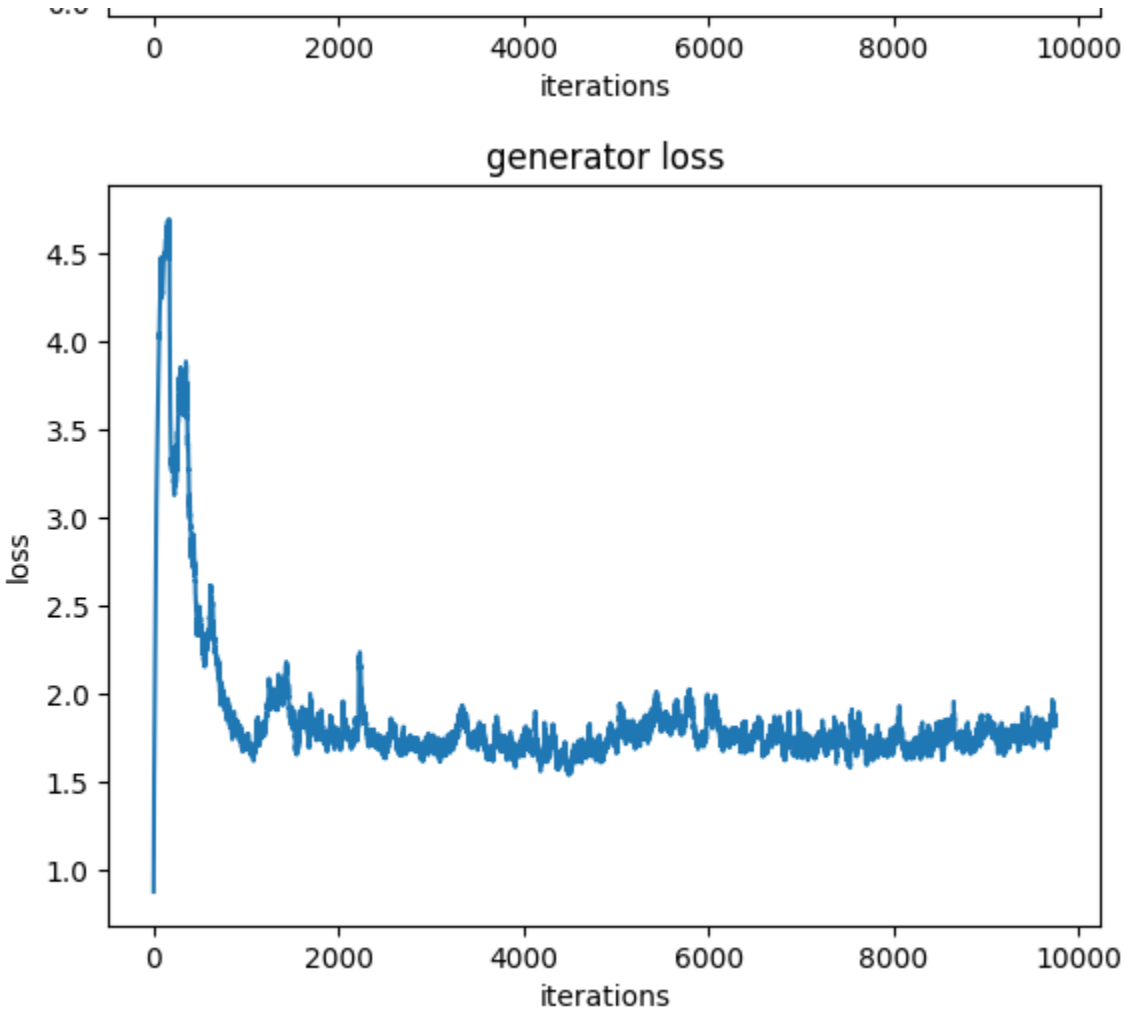



```
Iteration 9000/9750: dis loss = 0.7107, gen loss = 1.2066
Iteration 9100/9750: dis loss = 0.6807, gen loss = 2.0102
Iteration 9200/9750: dis loss = 0.5835, gen loss = 0.8215
Iteration 9300/9750: dis loss = 0.6669, gen loss = 1.8233
Iteration 9400/9750: dis loss = 0.7699, gen loss = 1.3902
Iteration 9500/9750: dis loss = 0.6008, gen loss = 1.3644
Iteration 9600/9750: dis loss = 1.0007, gen loss = 0.5519
Iteration 9700/9750: dis loss = 0.6734, gen loss = 1.7012
```




50
100
150
200
250
0
50
100
150
200
250


discriminator loss
loss
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0




```
... 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

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: In general, when we want to put real samples in the discriminator, we want the real samples to be as real as possible and come from only real data. Likewise, fake samples should actually be fake with no mixing of the two datasets. When we have a mixed batch of real and fake examples, then suddenly do BatchNorm, the normalization likely uses the mean and stddev of the mixed real/fake batch. Now instead of real examples being normalized by real_mean and real_std, the real examples are normalized with mixed_mean and mixed_std, which is not good because elements of fake data are treated as a real image in the discriminator. In the construction of the disciminator, the missing batchnorm layer could be used to prevent singular bad or fake examples in the dataset from corrupting the entire batch. By omitting this layer, the model has a chance to detect extreme samples first and pass them through an activation function before mixing means of potential outliers with the entire batch.

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)

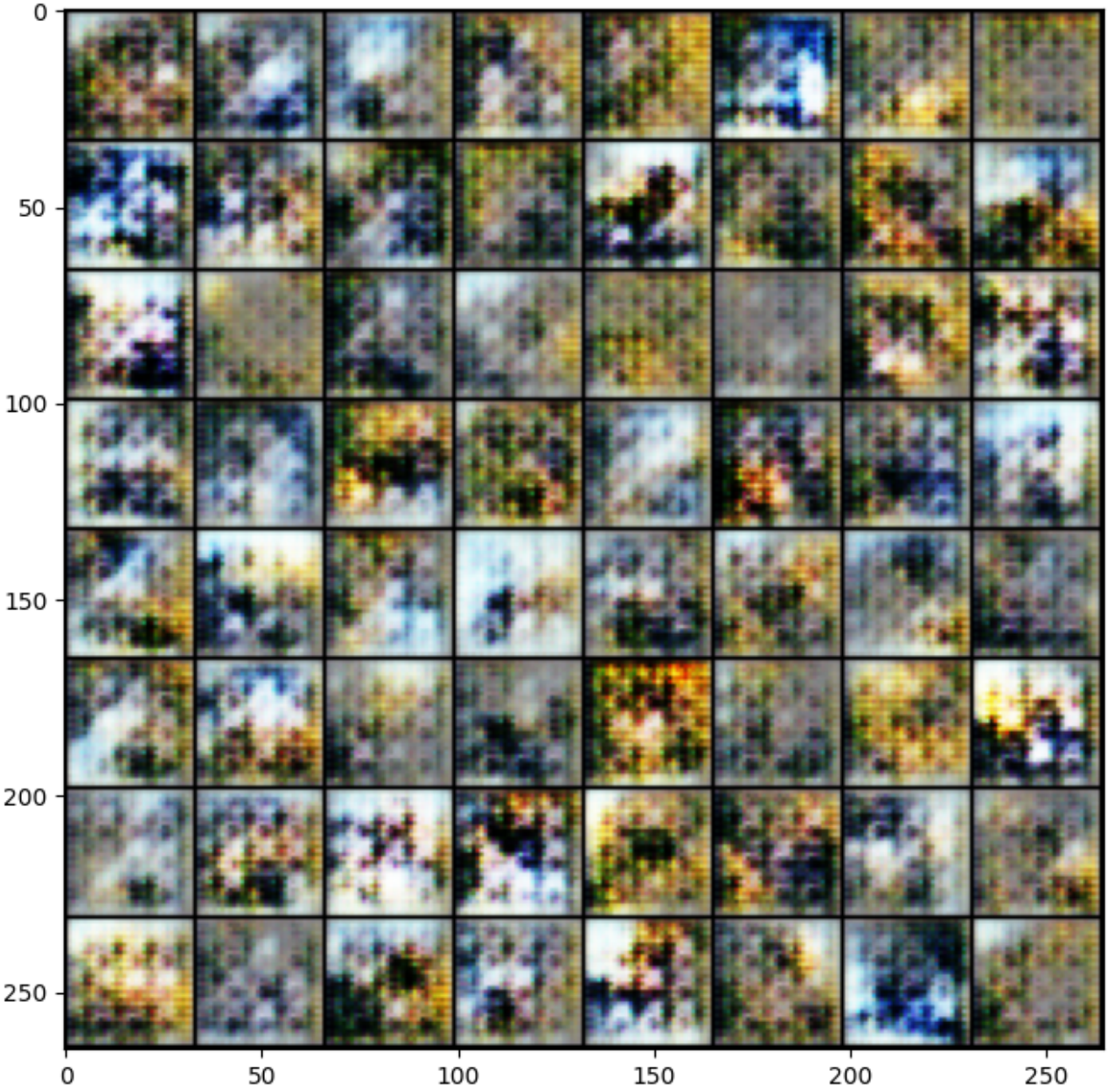
Spectral norm is a way of stabilizing the GAN training of discriminator. Please add the embedded spectral norm function in Pytorch to the Discriminator class below in order to test its effects. (see link: https://pytorch.org/docs/stable/generated/torch.nn.utils.spectral_norm.html)

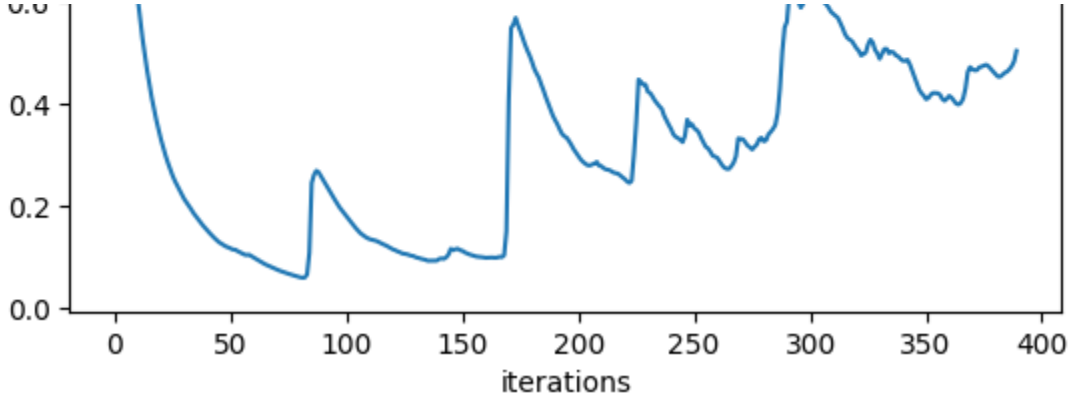
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        ##############################################################################
        # Prob 2-3:
        # adding spectral norm to the discriminator
        ##############################################################################
        self.downsample = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=4, stride=2, pad
            nn.LeakyReLU(),
            nn.utils.spectral_norm(nn.Conv2d(in_channels=32, out_channels=64, kern
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.utils.spectral_norm(nn.Conv2d(in_channels=64, out_channels=128, ker
            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
```

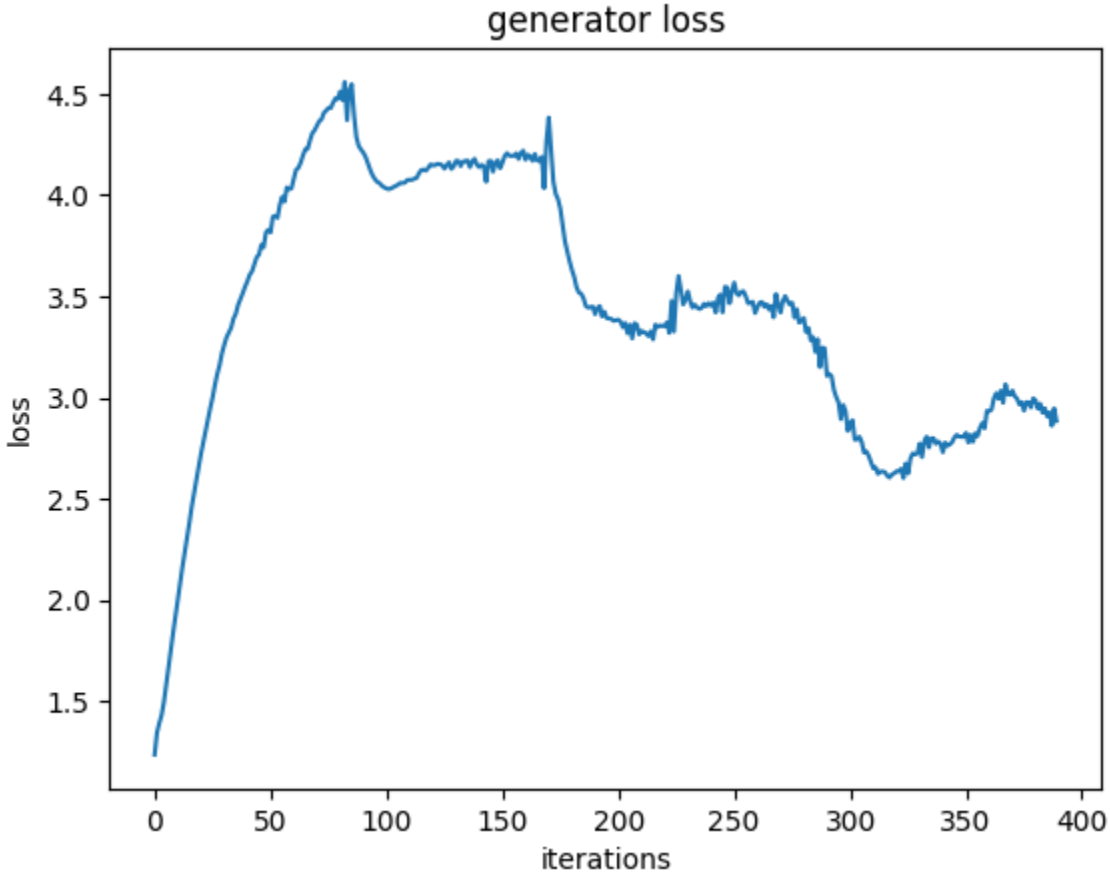
After adding the spectral norm to the discriminator, redo the training block below to see the effects.

```
set_seed(42)

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

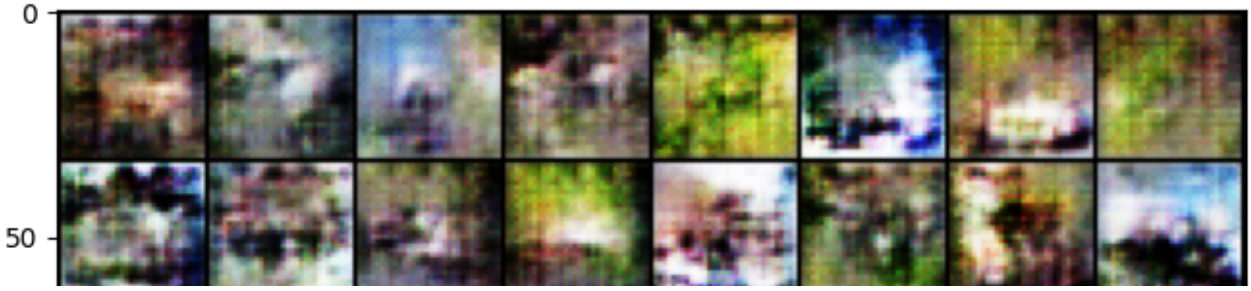
```
Start training ...
Iteration 100/9750: dis loss = 0.0661, gen loss = 3.8945
Iteration 200/9750: dis loss = 0.1765, gen loss = 3.4375
Iteration 300/9750: dis loss = 0.4384, gen loss = 0.9722
```

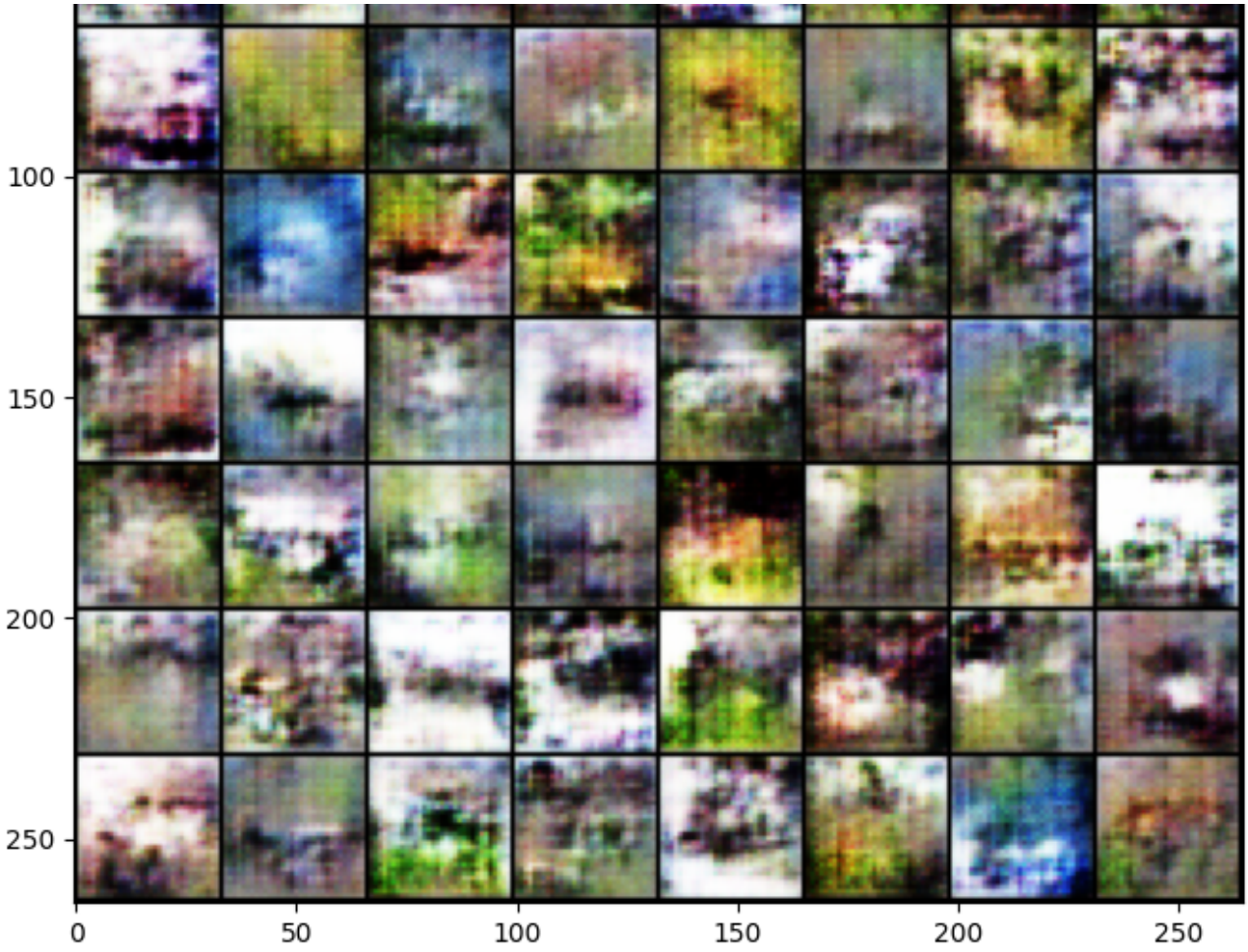


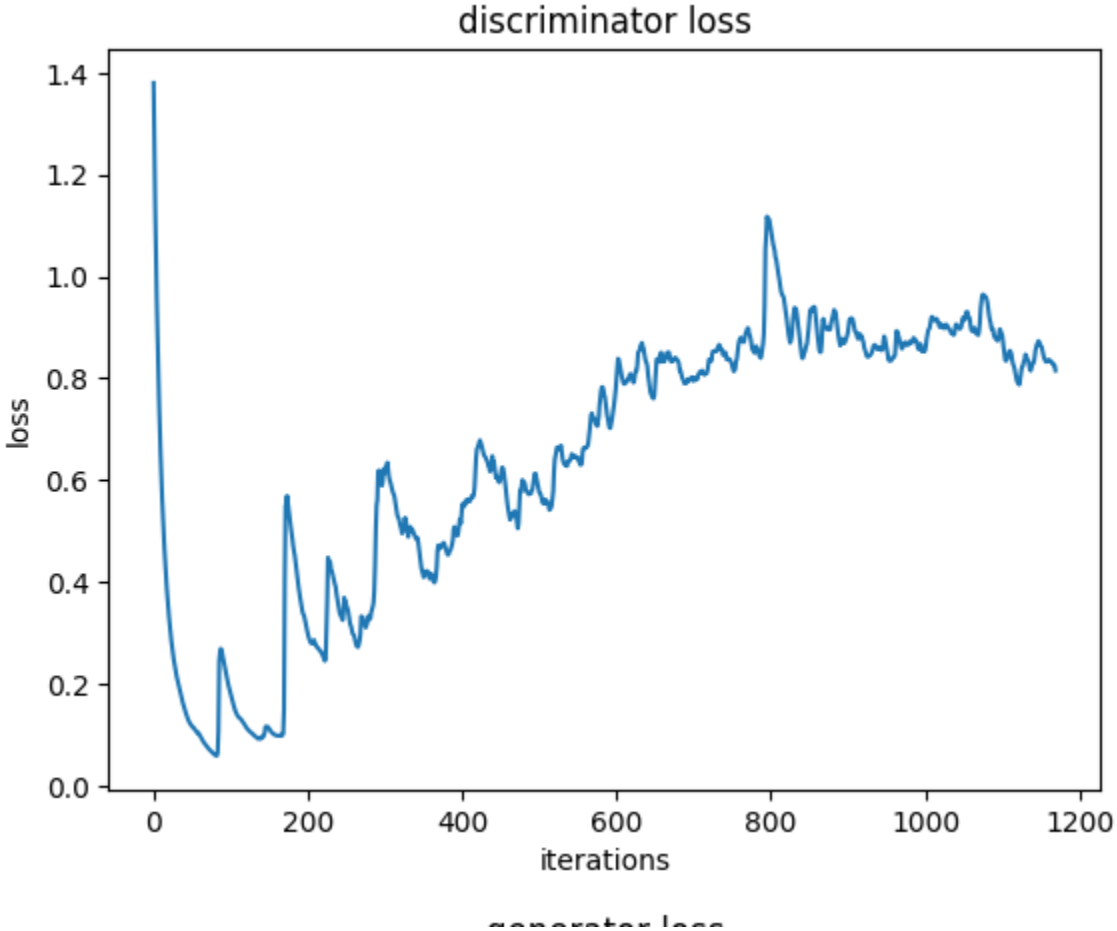


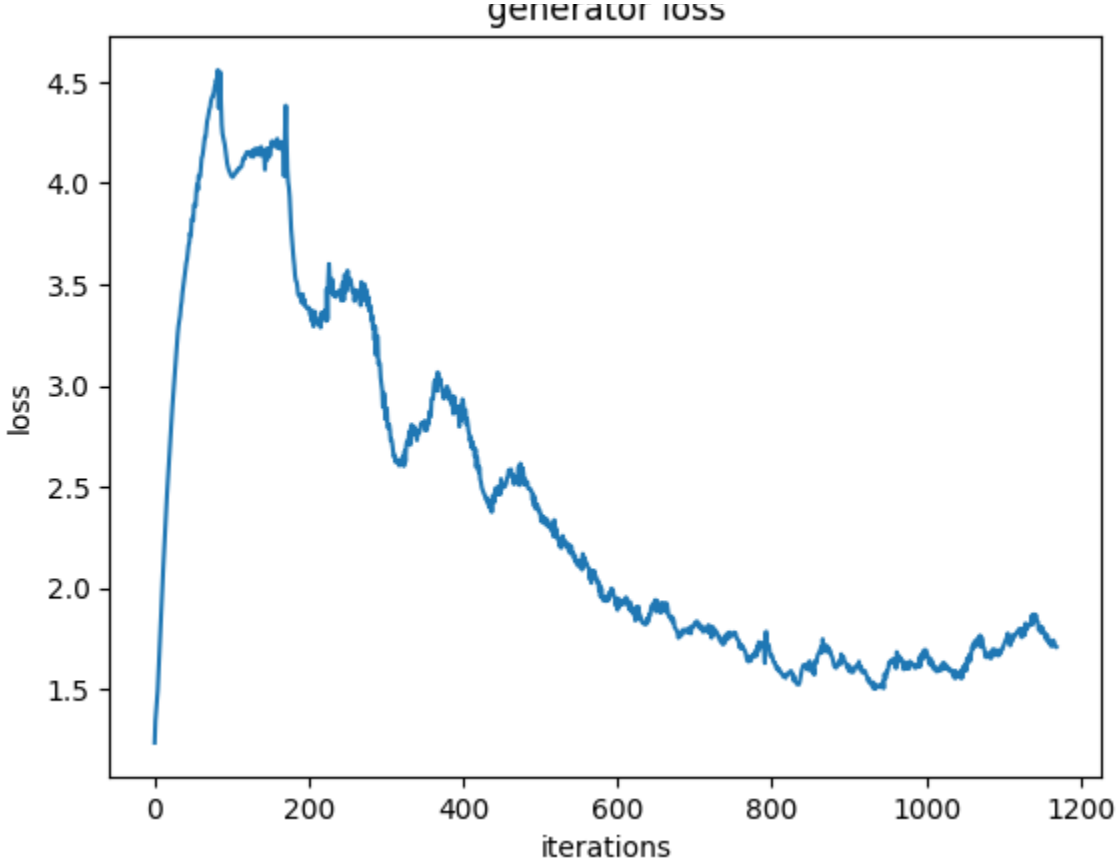



```
Iteration 400/9750: dis loss = 0.8193, gen loss = 4.8613
Iteration 500/9750: dis loss = 0.4032, gen loss = 2.5094
Iteration 600/9750: dis loss = 1.0558, gen loss = 2.5816
Iteration 700/9750: dis loss = 0.6783, gen loss = 1.7784
Iteration 800/9750: dis loss = 0.9444, gen loss = 1.6955
Iteration 900/9750: dis loss = 0.9644, gen loss = 1.8800
Iteration 1000/9750: dis loss = 0.7954, gen loss = 1.5486
Iteration 1100/9750: dis loss = 0.8126, gen loss = 1.7134
```

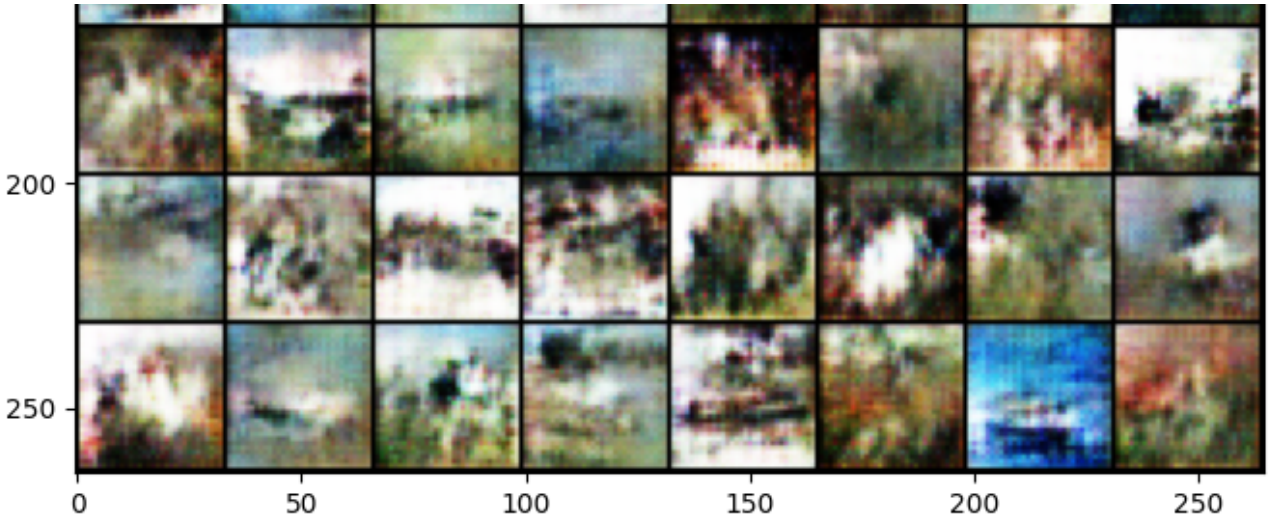


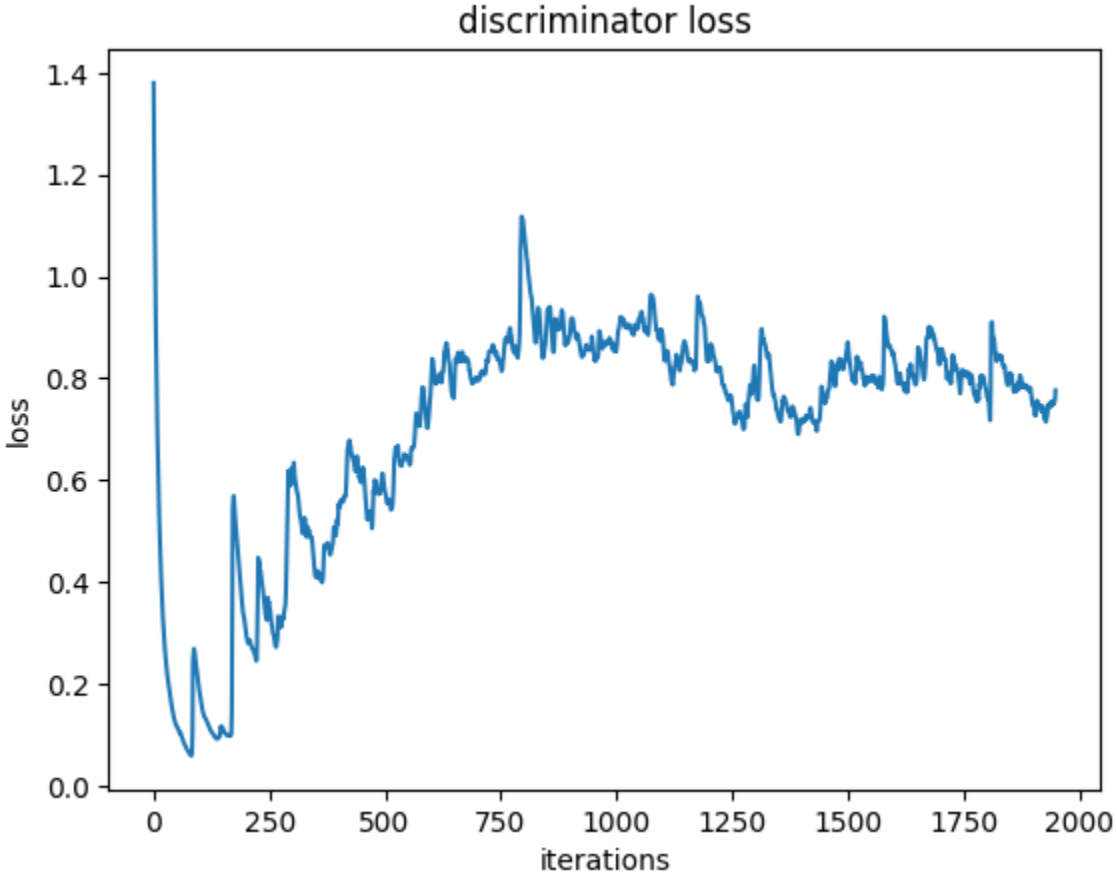

100
150
200
250
0
50
100
150
200
250

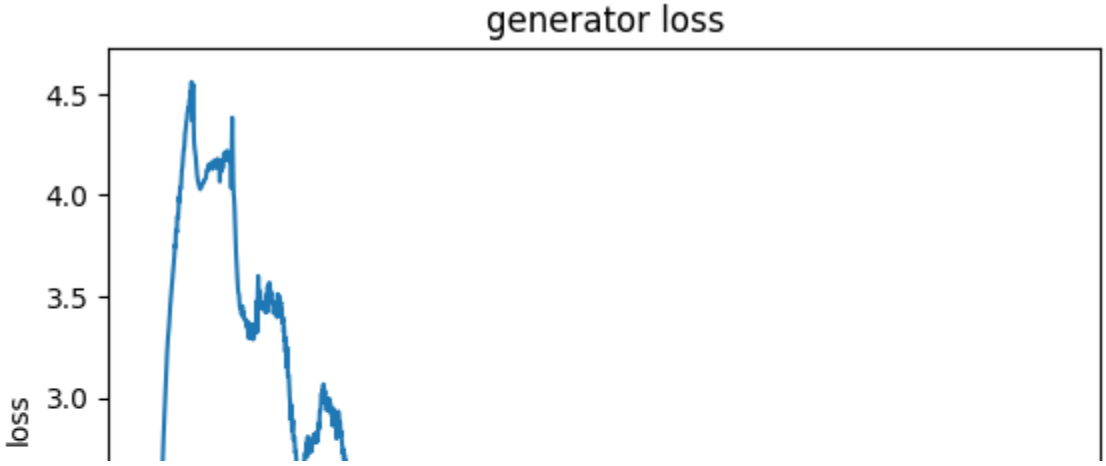

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
200
400
600
800
1000
1200

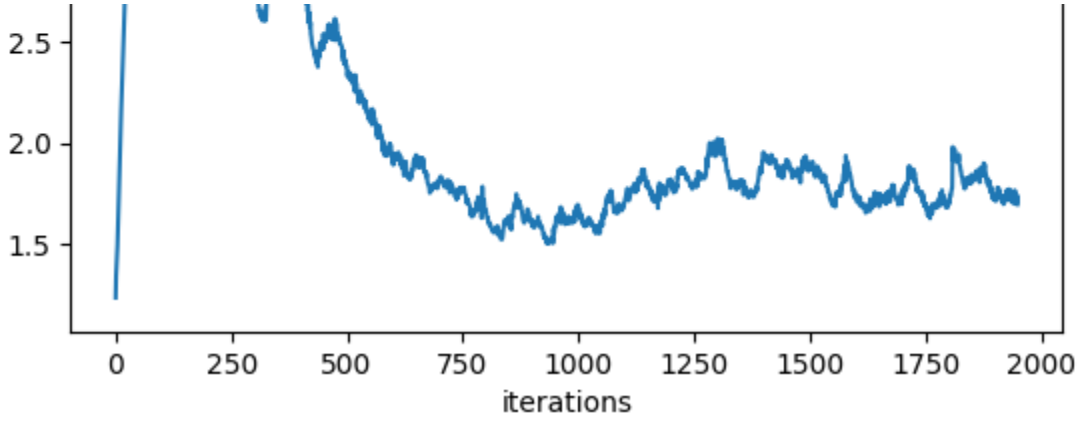



```
Iteration 1200/9750: dis loss = 0.9051, gen loss = 1.9256
Iteration 1300/9750: dis loss = 0.6597, gen loss = 2.1814
Iteration 1400/9750: dis loss = 0.6993, gen loss = 2.4673
Iteration 1500/9750: dis loss = 1.0963, gen loss = 2.2057
Iteration 1600/9750: dis loss = 0.8488, gen loss = 1.8809
Iteration 1700/9750: dis loss = 1.1010, gen loss = 0.6307
Iteration 1800/9750: dis loss = 0.6943, gen loss = 2.1552
Iteration 1900/9750: dis loss = 0.6097, gen loss = 2.0339
```

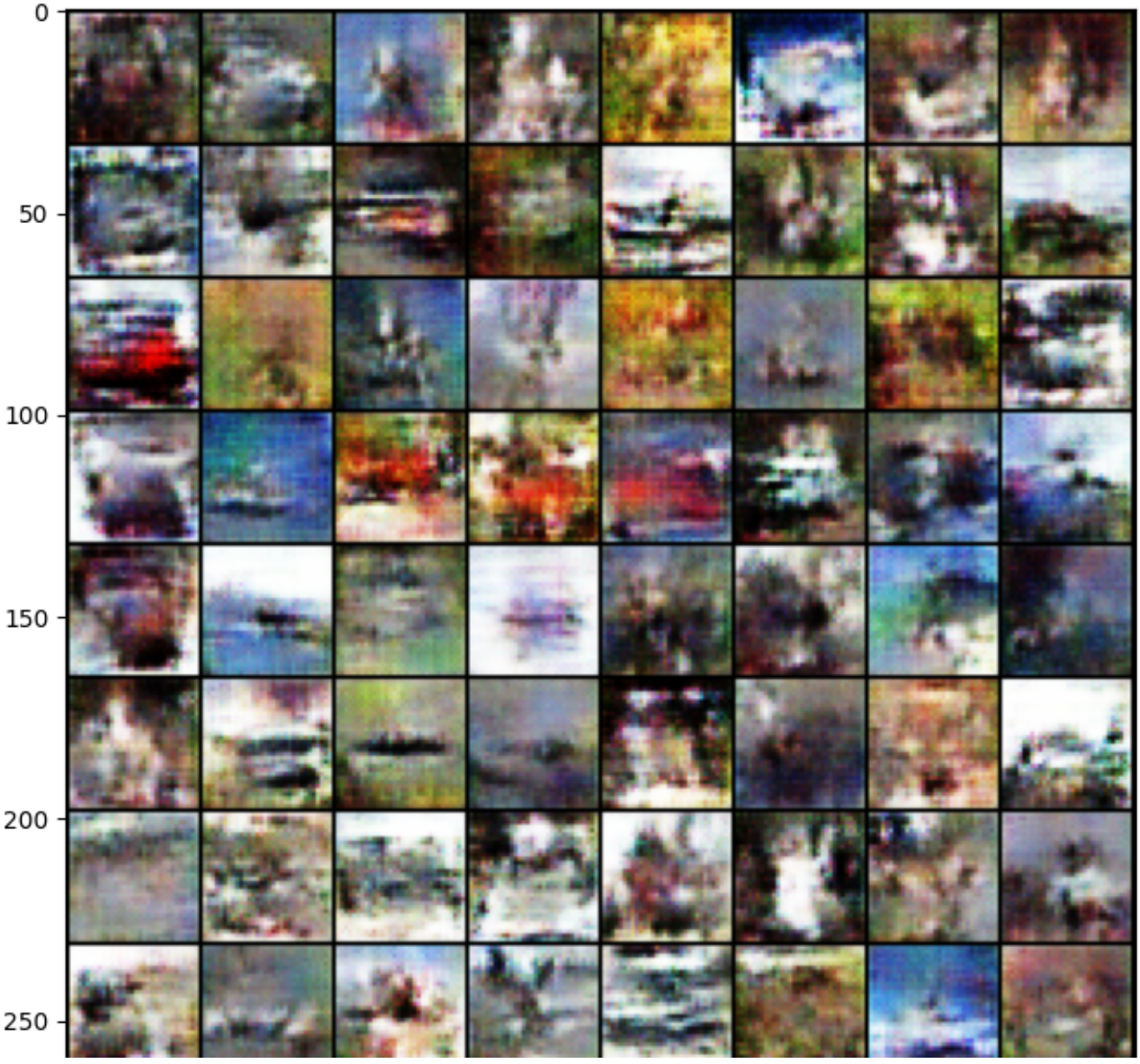

200
250
0
50
100
150
200
250


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

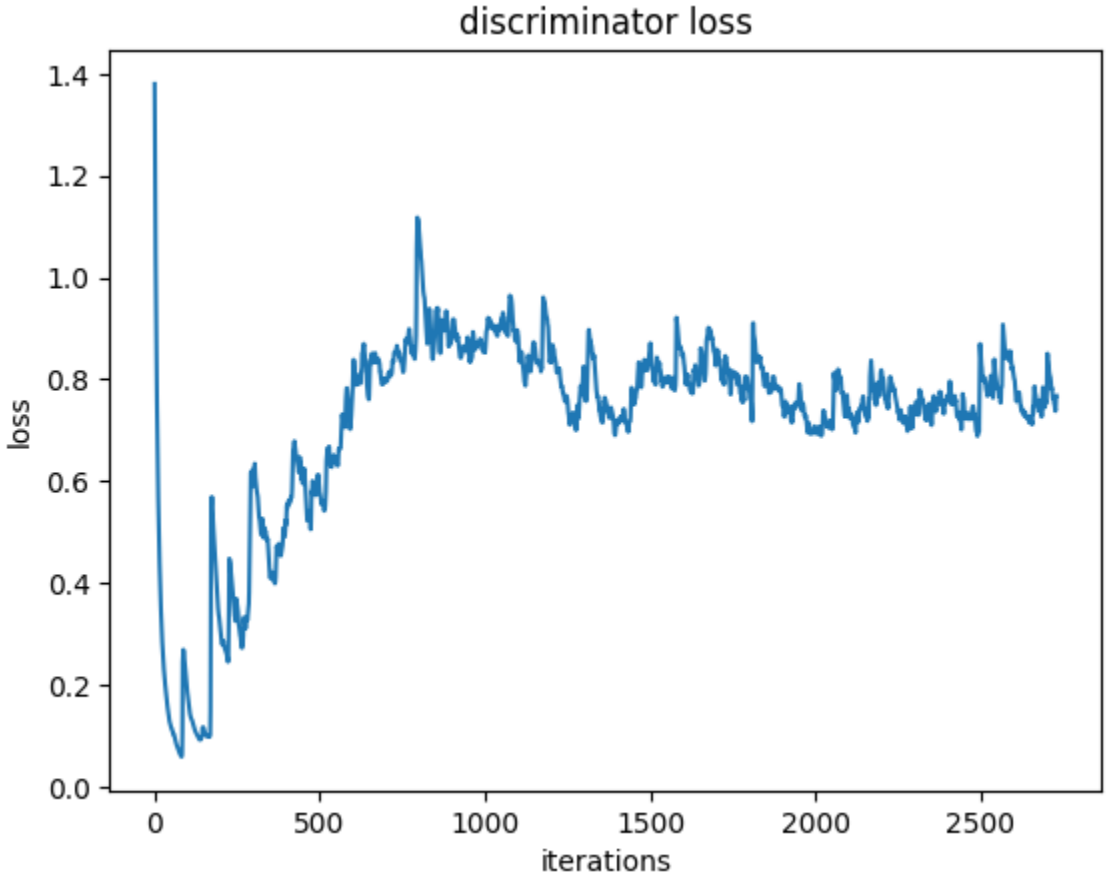

generator loss
loss
3.0
3.5
4.0
4.5

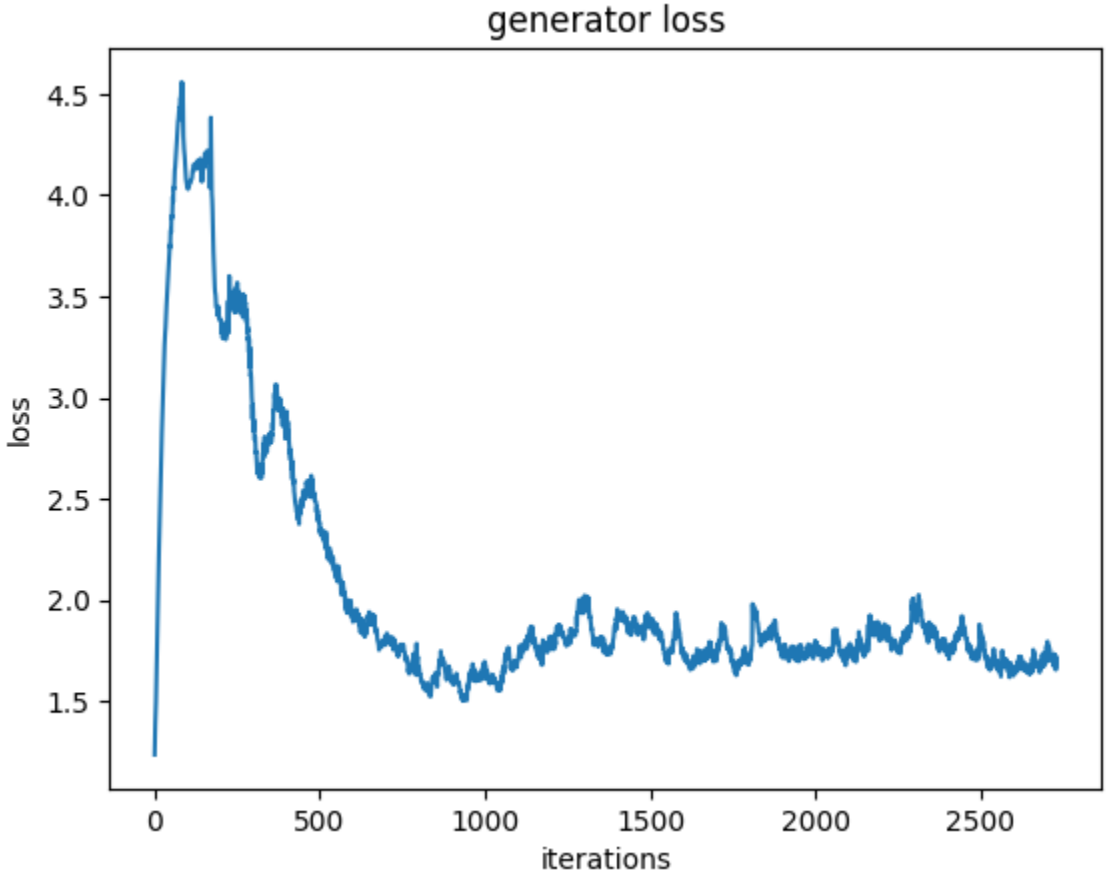



```
Iteration 2000/9750: dis loss = 0.5401, gen loss = 2.4279
Iteration 2100/9750: dis loss = 0.7781, gen loss = 1.7250
Iteration 2200/9750: dis loss = 0.8428, gen loss = 1.5921
Iteration 2300/9750: dis loss = 0.4999, gen loss = 1.9840
Iteration 2400/9750: dis loss = 0.7213, gen loss = 1.4386
Iteration 2500/9750: dis loss = 0.7220, gen loss = 2.3598
Iteration 2600/9750: dis loss = 0.7855, gen loss = 1.1507
Iteration 2700/9750: dis loss = 0.8498, gen loss = 3.3145
```

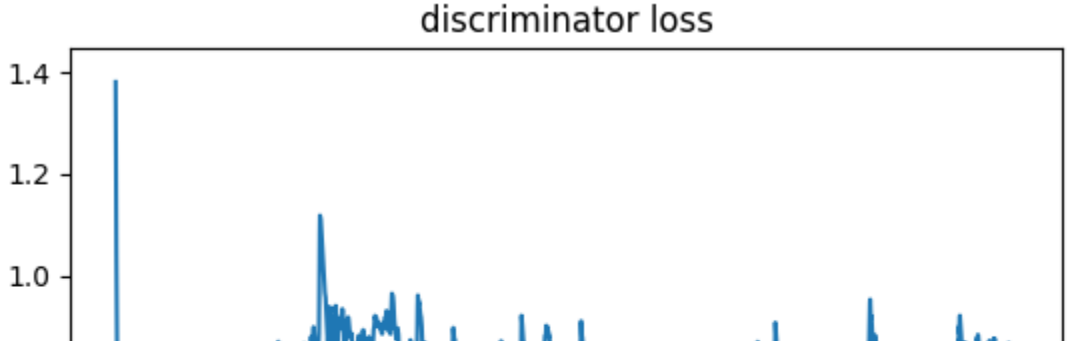


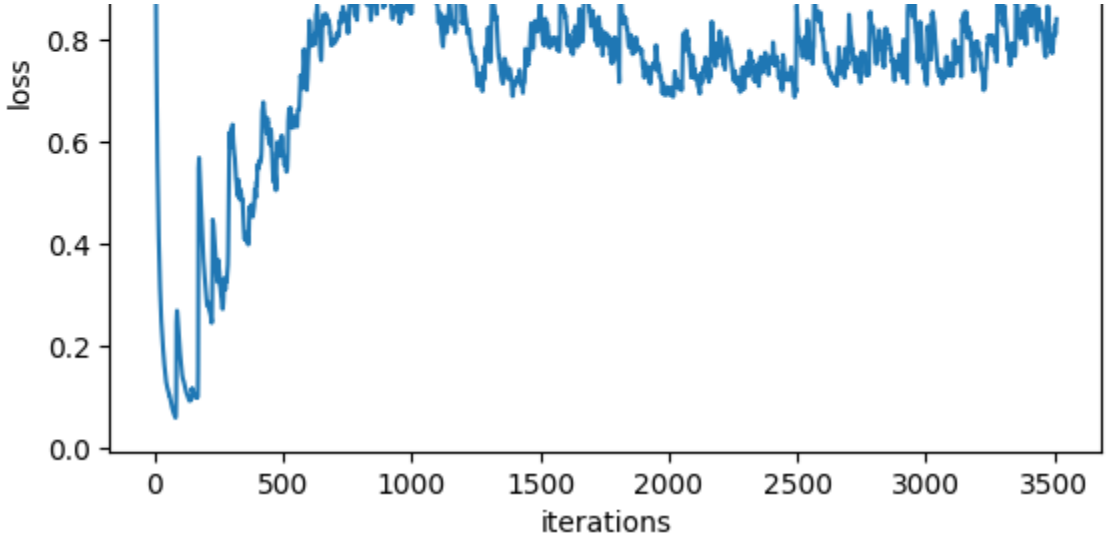

0
50
100
150
200
250

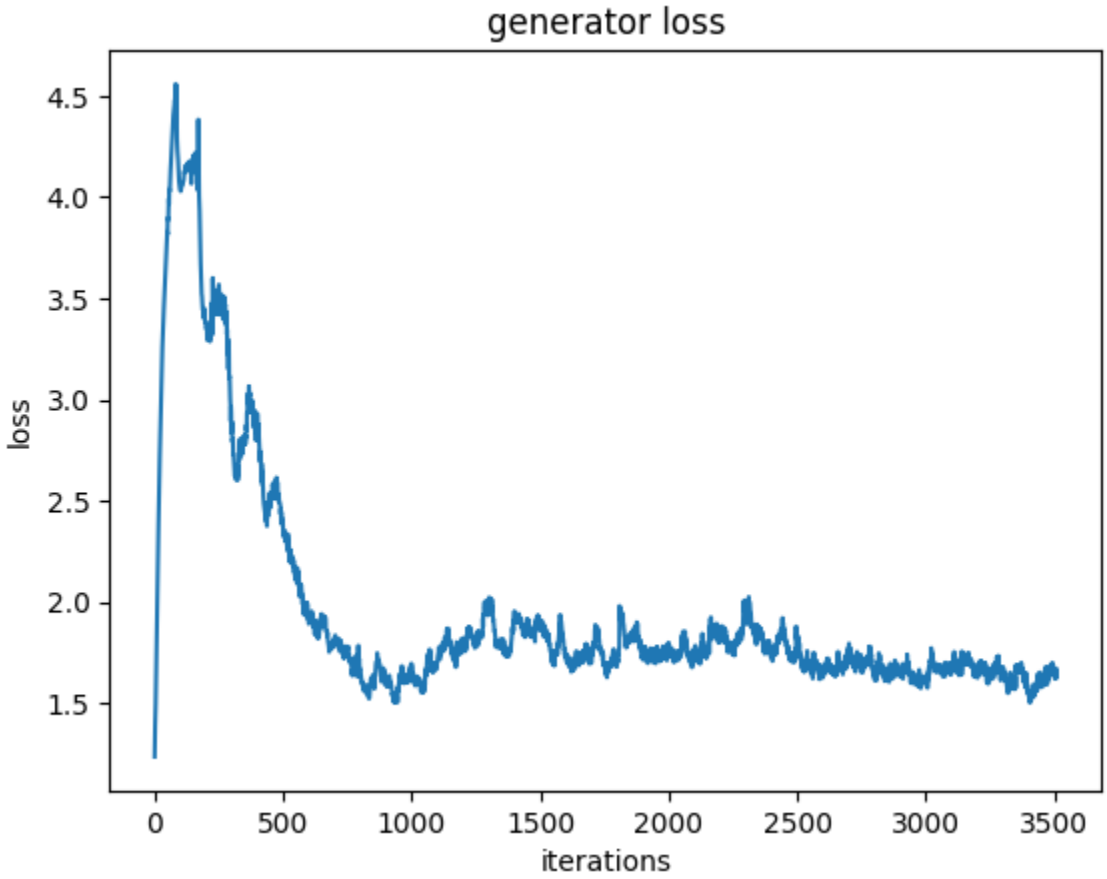

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
500
1000
1500
2000
2500


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

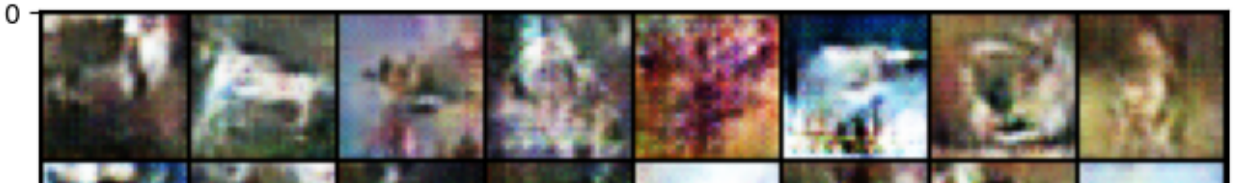
```
Iteration 2800/9750: dis loss = 0.7138, gen loss = 1.3507
Iteration 2900/9750: dis loss = 0.6854, gen loss = 1.4614
Iteration 3000/9750: dis loss = 0.6024, gen loss = 1.2162
Iteration 3100/9750: dis loss = 0.7366, gen loss = 1.2641
Iteration 3200/9750: dis loss = 0.8968, gen loss = 0.9496
Iteration 3300/9750: dis loss = 0.9134, gen loss = 1.2230
Iteration 3400/9750: dis loss = 0.9328, gen loss = 1.8153
Iteration 3500/9750: dis loss = 0.8698, gen loss = 1.2932
```

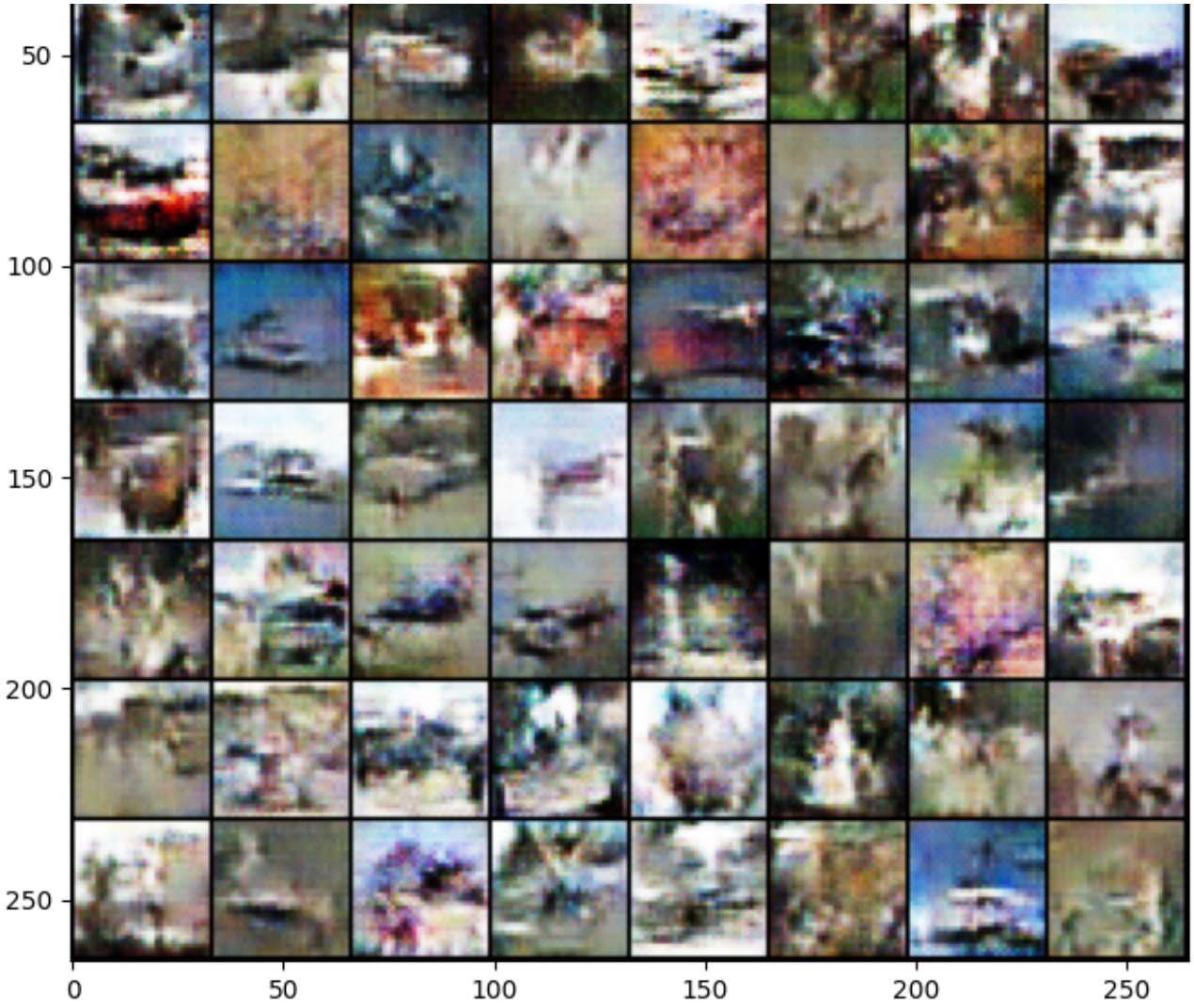


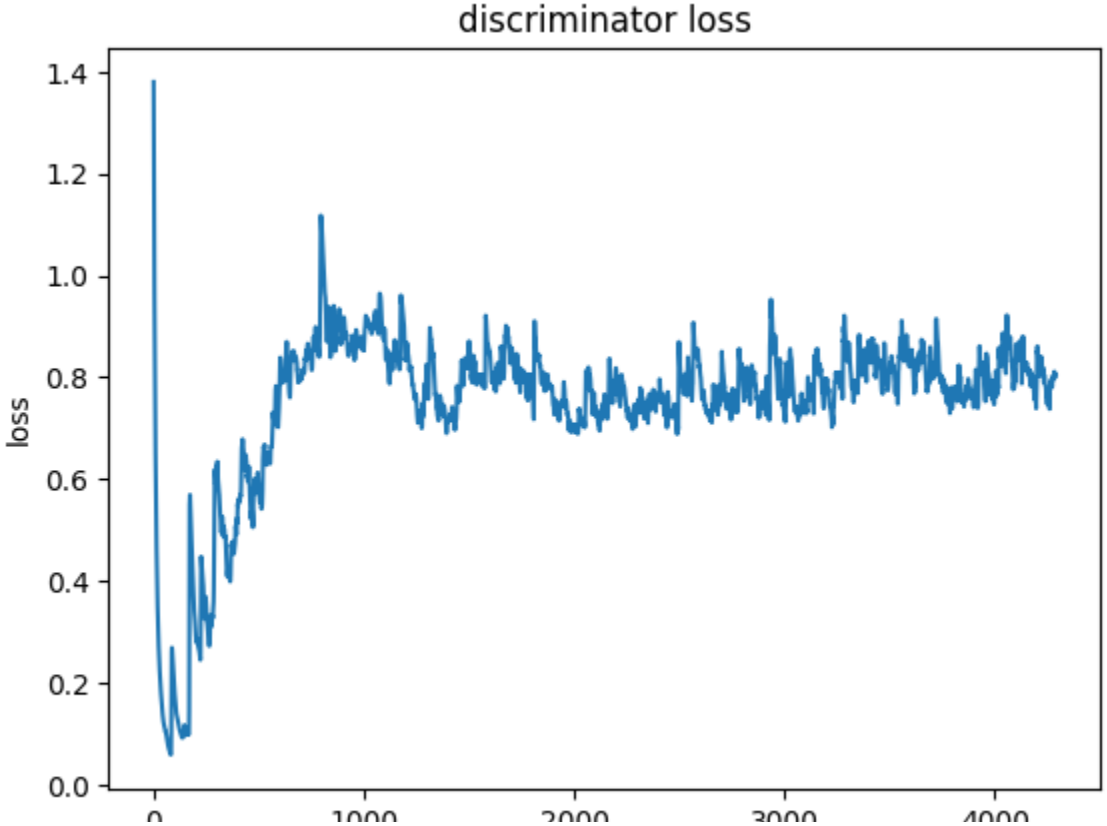


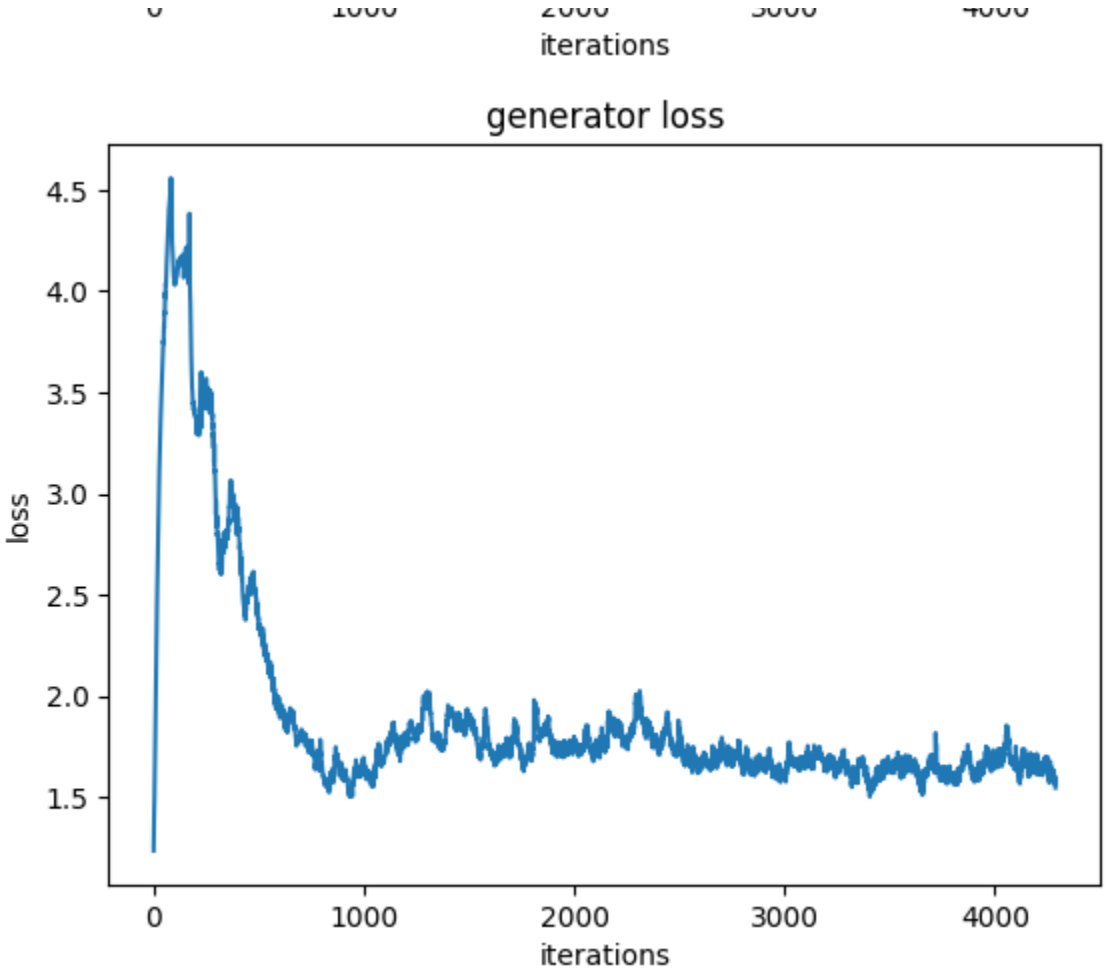



```
Iteration 3600/9750: dis loss = 1.0222, gen loss = 1.4478
Iteration 3700/9750: dis loss = 0.7173, gen loss = 1.3499
Iteration 3800/9750: dis loss = 0.7253, gen loss = 1.3995
Iteration 3900/9750: dis loss = 0.6342, gen loss = 2.1046
Iteration 4000/9750: dis loss = 0.9978, gen loss = 0.9103
Iteration 4100/9750: dis loss = 1.0856, gen loss = 1.0049
Iteration 4200/9750: dis loss = 1.2339, gen loss = 1.4068
```

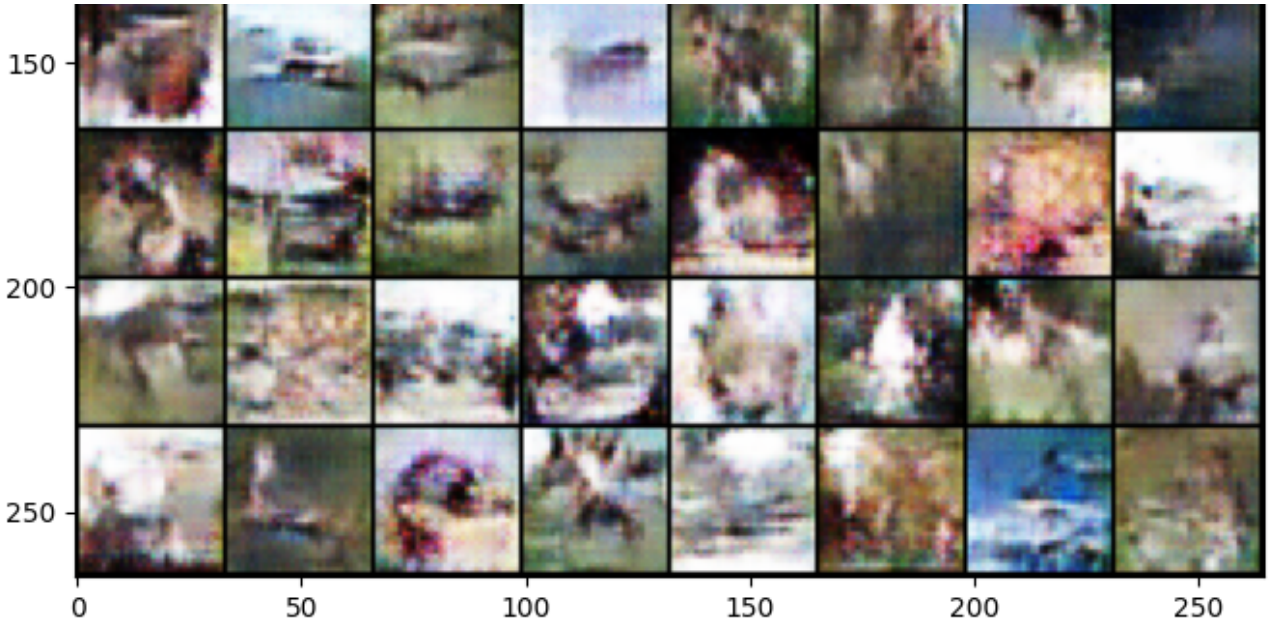


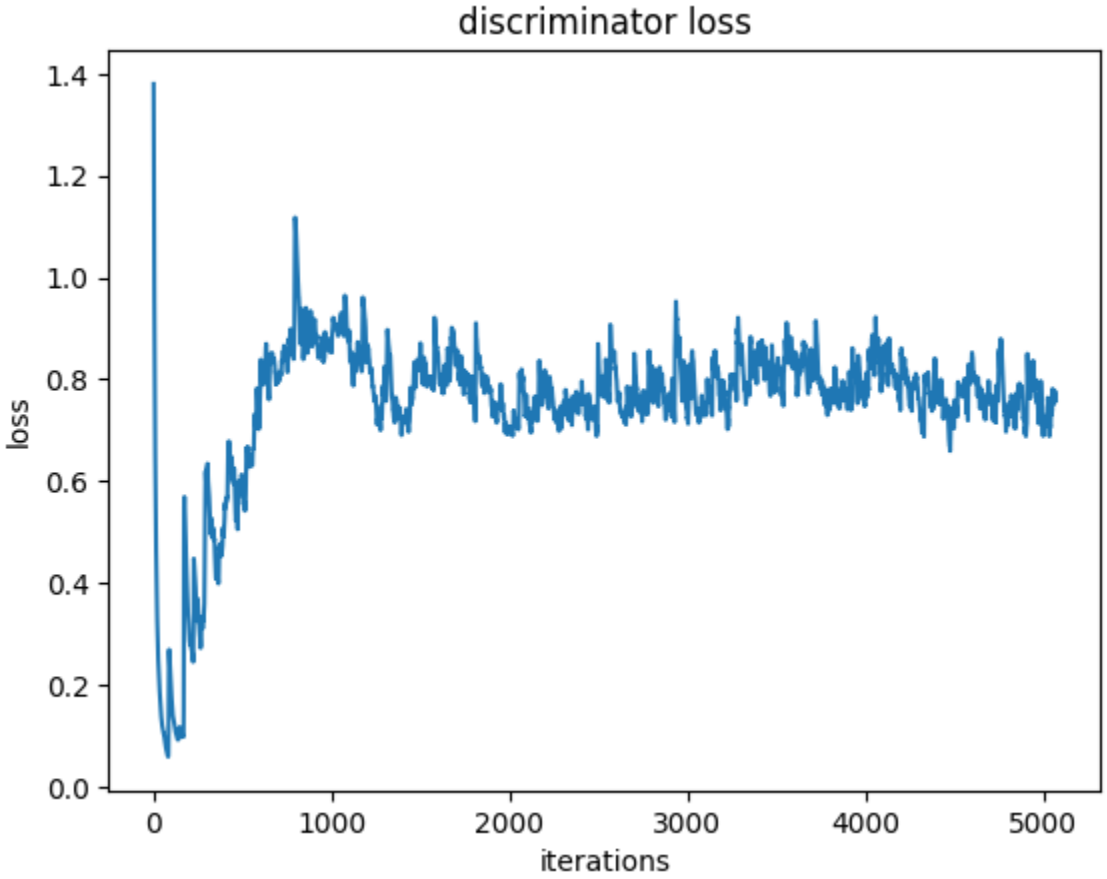

50
100
150
200
250
0
50
100
150
200
250

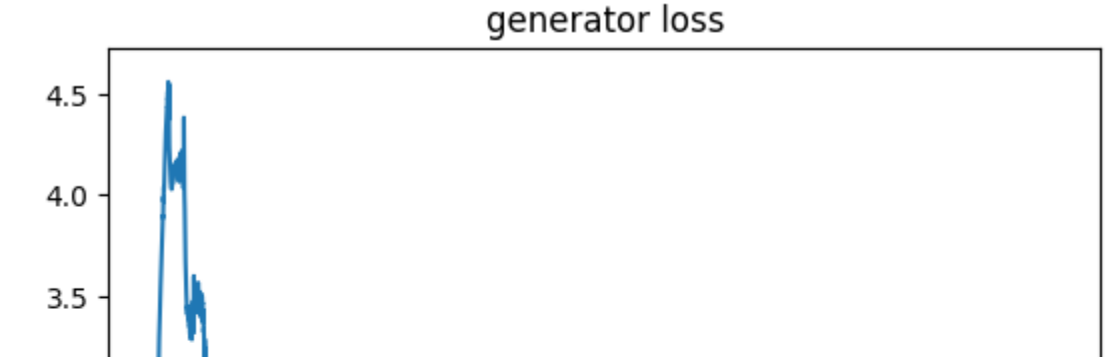

discriminator loss
loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
1000
2000
3000
4000

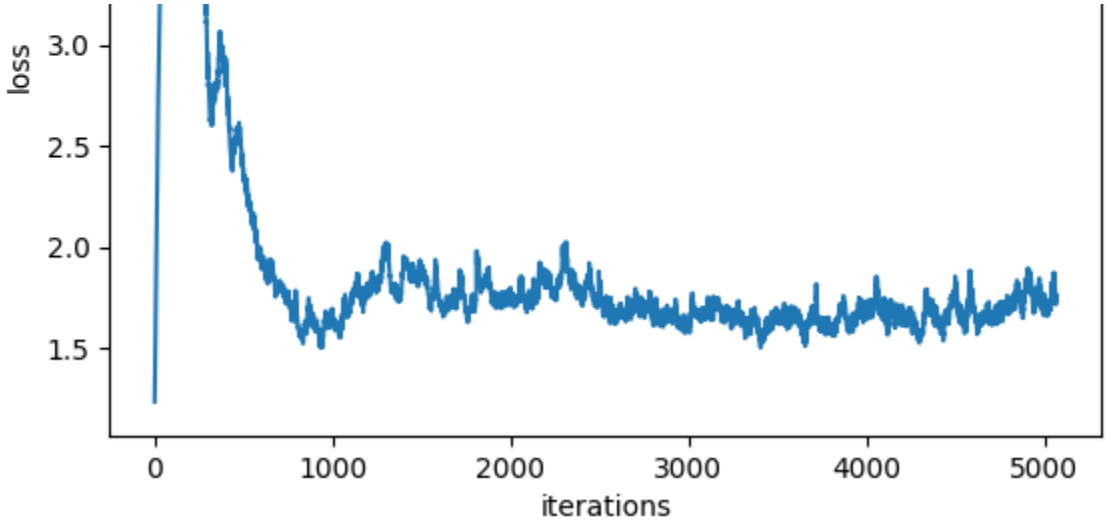



```
Iteration 4300/9750: dis loss = 0.7705, gen loss = 0.8942
Iteration 4400/9750: dis loss = 0.7971, gen loss = 1.4164
Iteration 4500/9750: dis loss = 0.5909, gen loss = 1.1290
Iteration 4600/9750: dis loss = 1.0911, gen loss = 2.1221
Iteration 4700/9750: dis loss = 0.6626, gen loss = 1.5287
Iteration 4800/9750: dis loss = 0.6340, gen loss = 1.4792
Iteration 4900/9750: dis loss = 0.4678, gen loss = 3.2987
Iteration 5000/9750: dis loss = 0.4883, gen loss = 1.9980
```

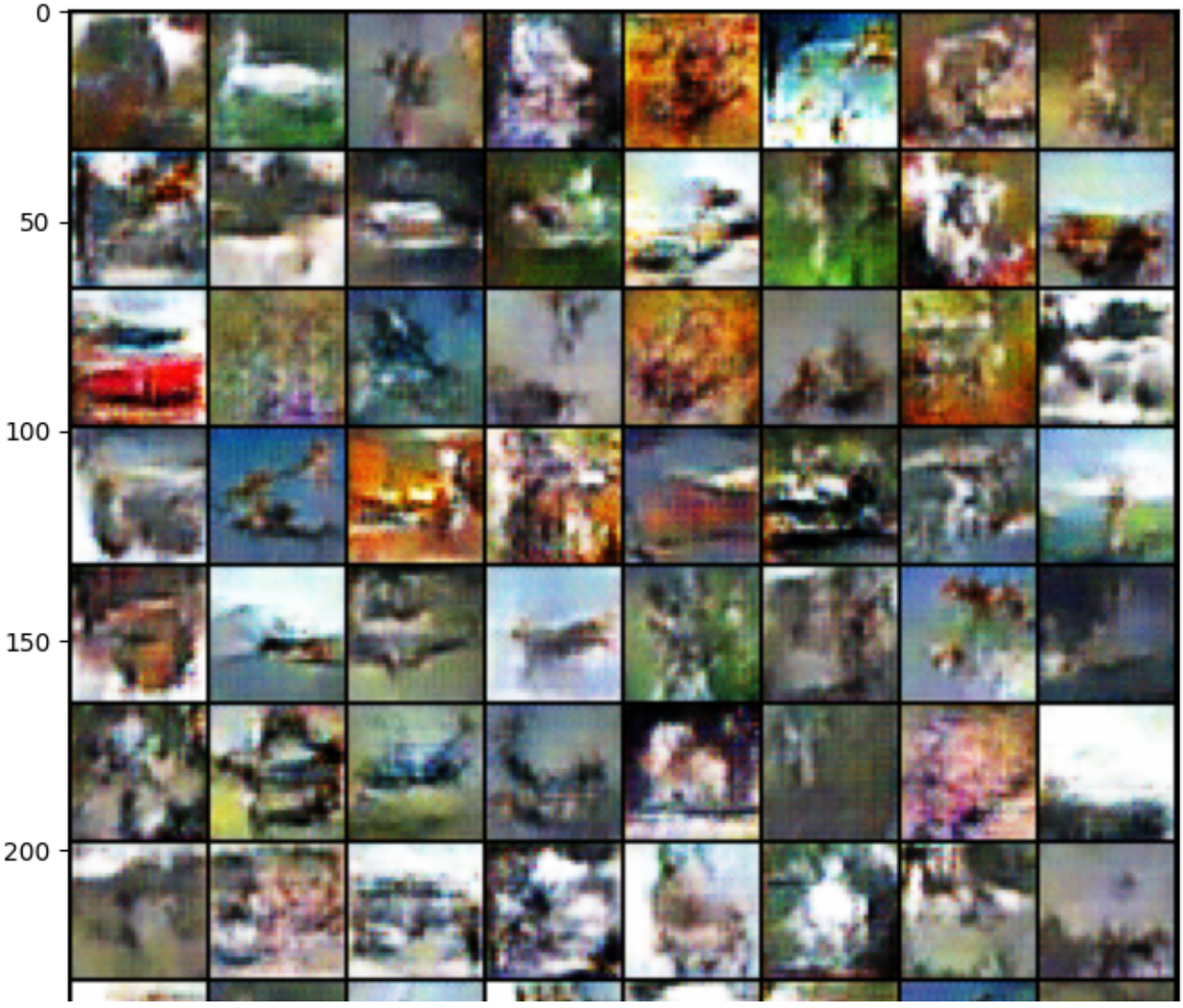

150
200
250
0
50
100
150
200
250


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

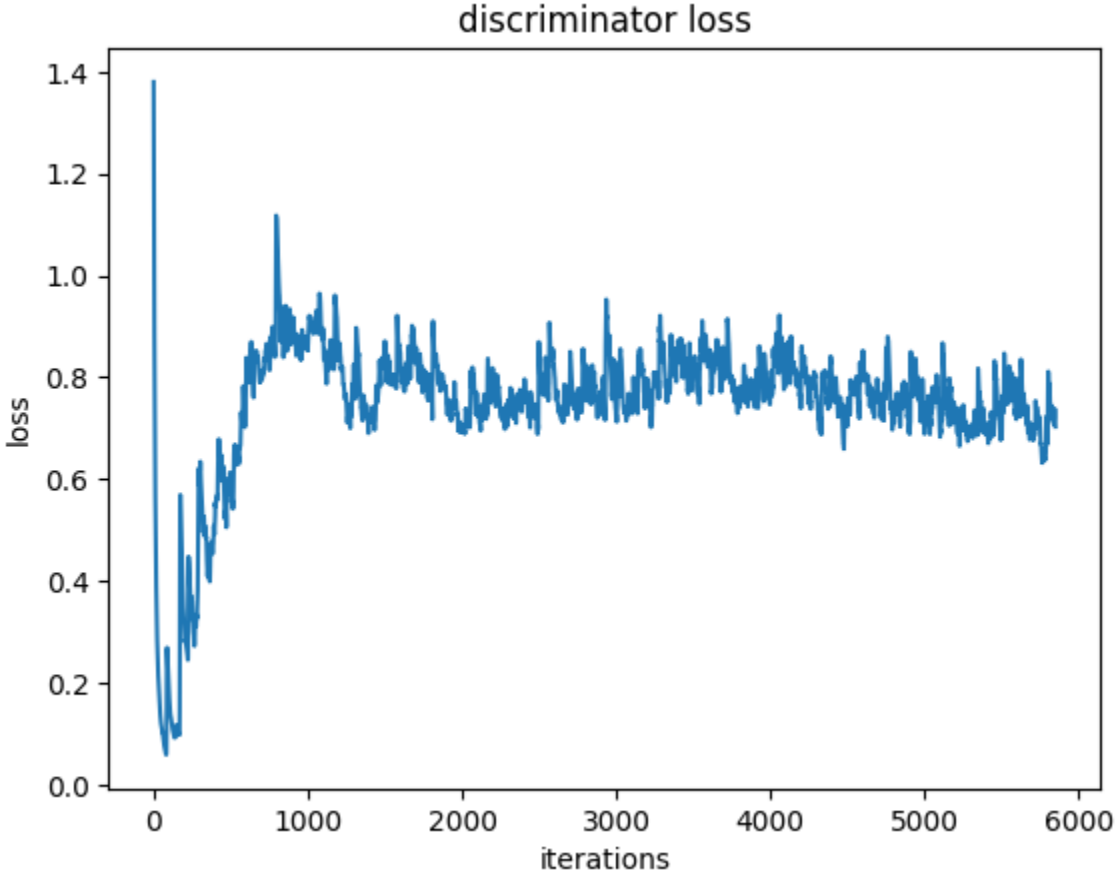

generator loss
4.5
4.0
3.5

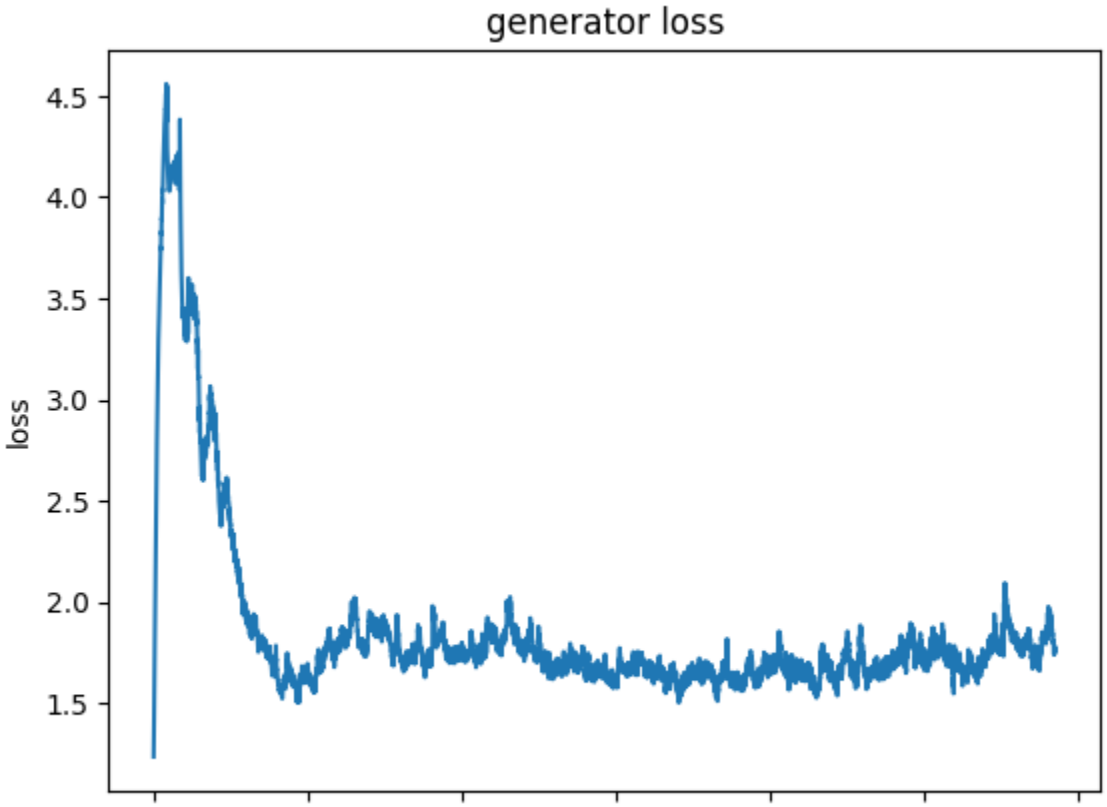


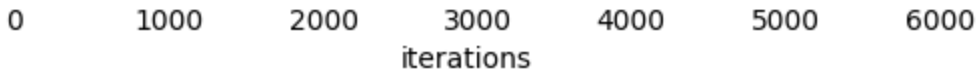

```
Iteration 5100/9750: dis loss = 0.6081, gen loss = 2.1102
Iteration 5200/9750: dis loss = 0.7095, gen loss = 1.4954
Iteration 5300/9750: dis loss = 0.8262, gen loss = 1.7638
Iteration 5400/9750: dis loss = 0.7981, gen loss = 2.3184
Iteration 5500/9750: dis loss = 0.5667, gen loss = 1.7614
Iteration 5600/9750: dis loss = 0.8490, gen loss = 1.5442
Iteration 5700/9750: dis loss = 0.5725, gen loss = 1.5310
Iteration 5800/9750: dis loss = 1.4355, gen loss = 3.2909
```



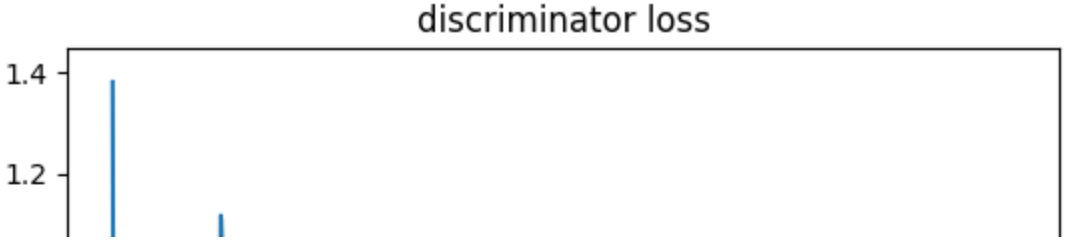

250
0
50
100
150
200
250

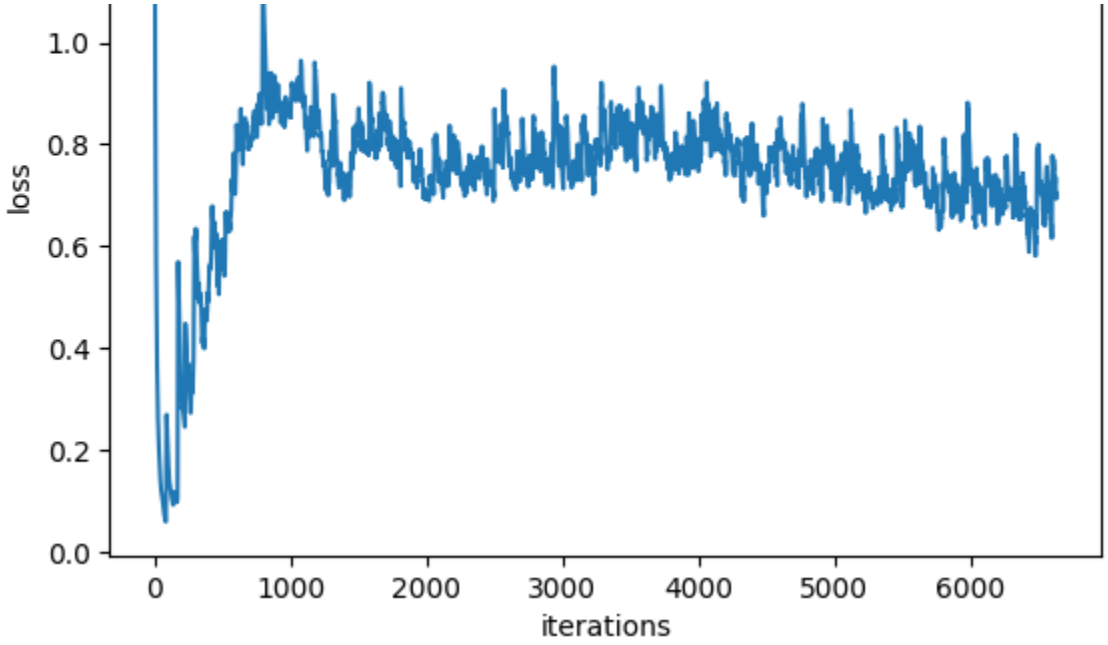

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
1000
2000
3000
4000
5000
6000

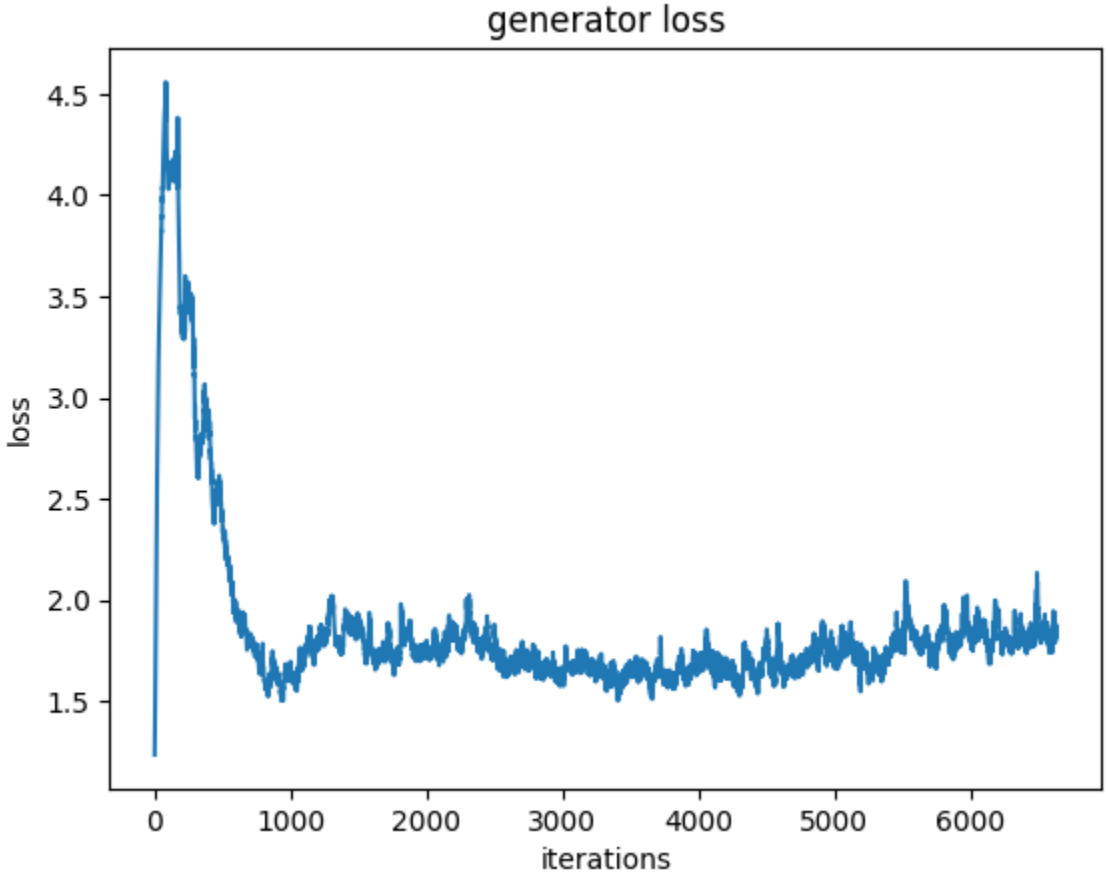

generator loss
loss
1.5
2.0
2.5
3.0
3.5
4.0
4.5




```
Iteration 5900/9750: dis loss = 0.6535, gen loss = 1.7398
Iteration 6000/9750: dis loss = 0.6553, gen loss = 1.8162
Iteration 6100/9750: dis loss = 0.5415, gen loss = 1.7912
Iteration 6200/9750: dis loss = 0.7784, gen loss = 2.1922
Iteration 6300/9750: dis loss = 0.6243, gen loss = 1.3971
Iteration 6400/9750: dis loss = 0.7577, gen loss = 1.6908
Iteration 6500/9750: dis loss = 0.9261, gen loss = 1.5339
Iteration 6600/9750: dis loss = 1.6640, gen loss = 2.5011
```

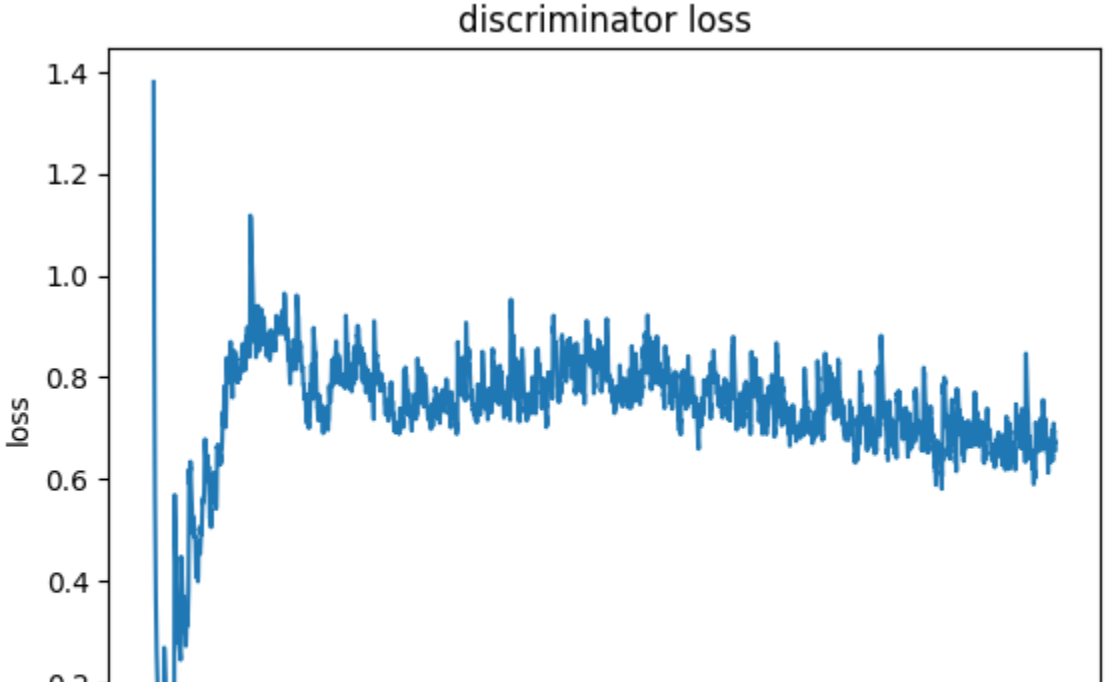


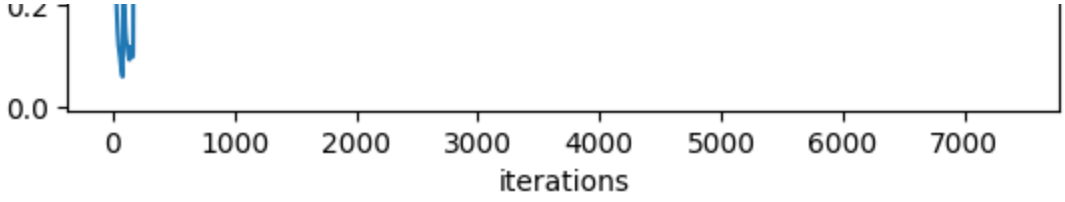


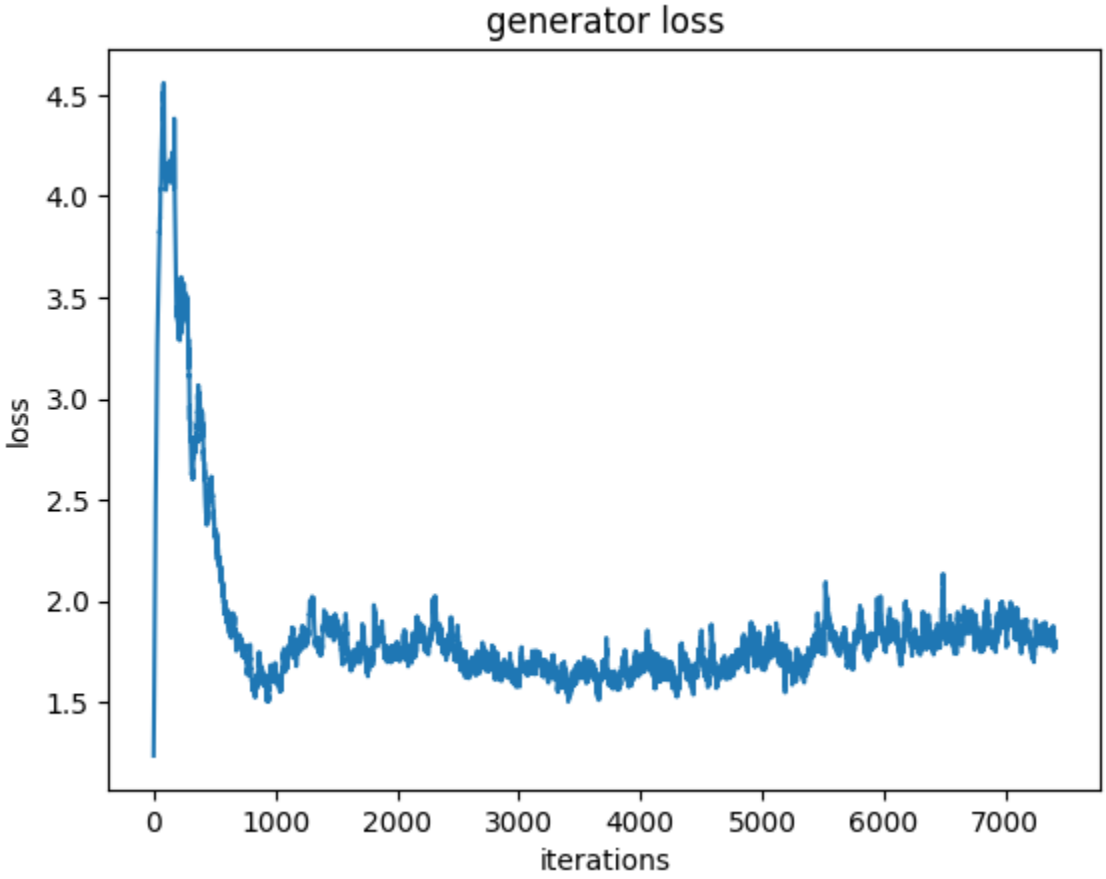



```
Iteration 6700/9750: dis loss = 0.7058, gen loss = 1.1124
Iteration 6800/9750: dis loss = 0.9041, gen loss = 1.1968
Iteration 6900/9750: dis loss = 0.4849, gen loss = 1.9438
Iteration 7000/9750: dis loss = 0.7393, gen loss = 1.1634
Iteration 7100/9750: dis loss = 0.3875, gen loss = 1.2356
Iteration 7200/9750: dis loss = 0.7012, gen loss = 1.5353
Iteration 7300/9750: dis loss = 0.5359, gen loss = 1.4898
Iteration 7400/9750: dis loss = 0.7873, gen loss = 2.4425
```

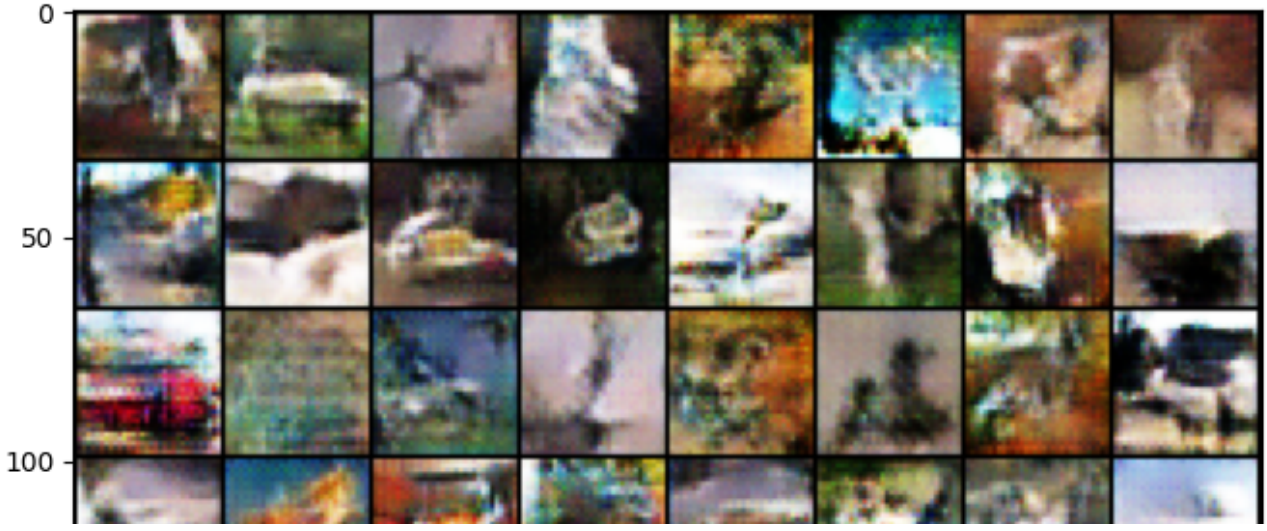

0
50
100
150
200
250

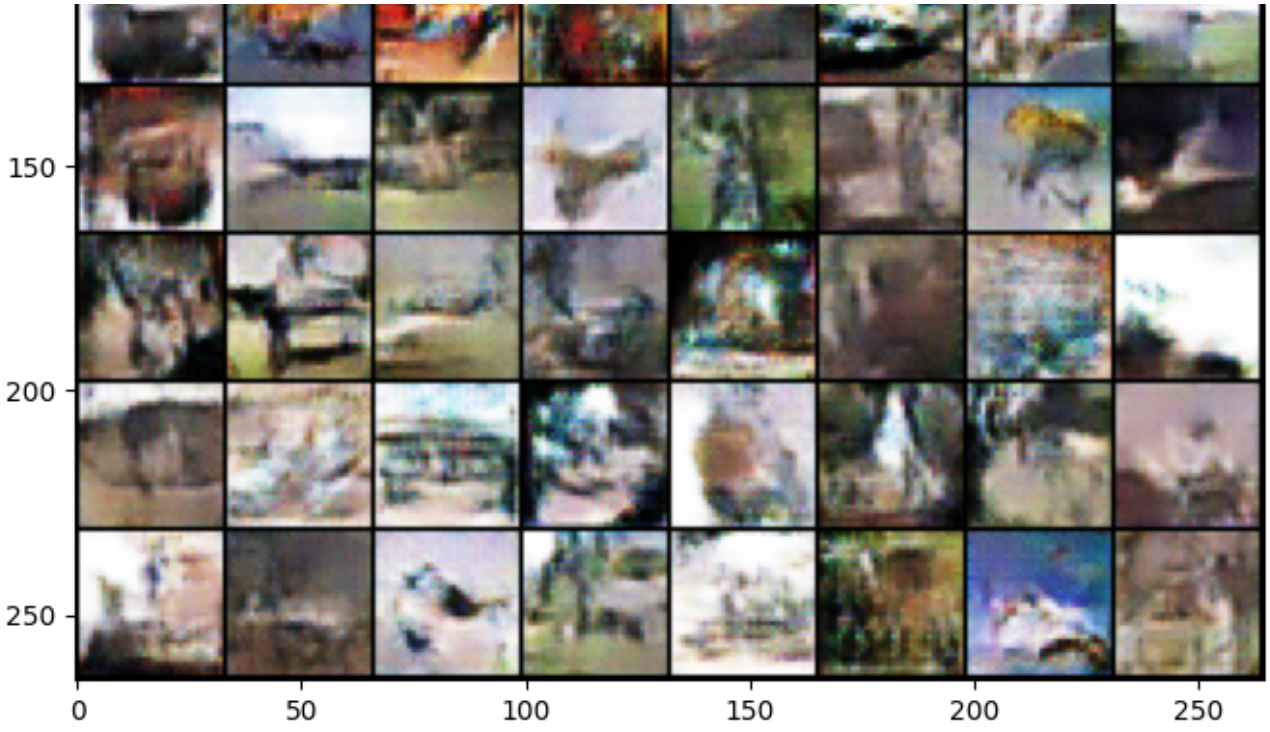

discriminator loss
loss
1.4
1.2
1.0
0.8
0.6
0.4

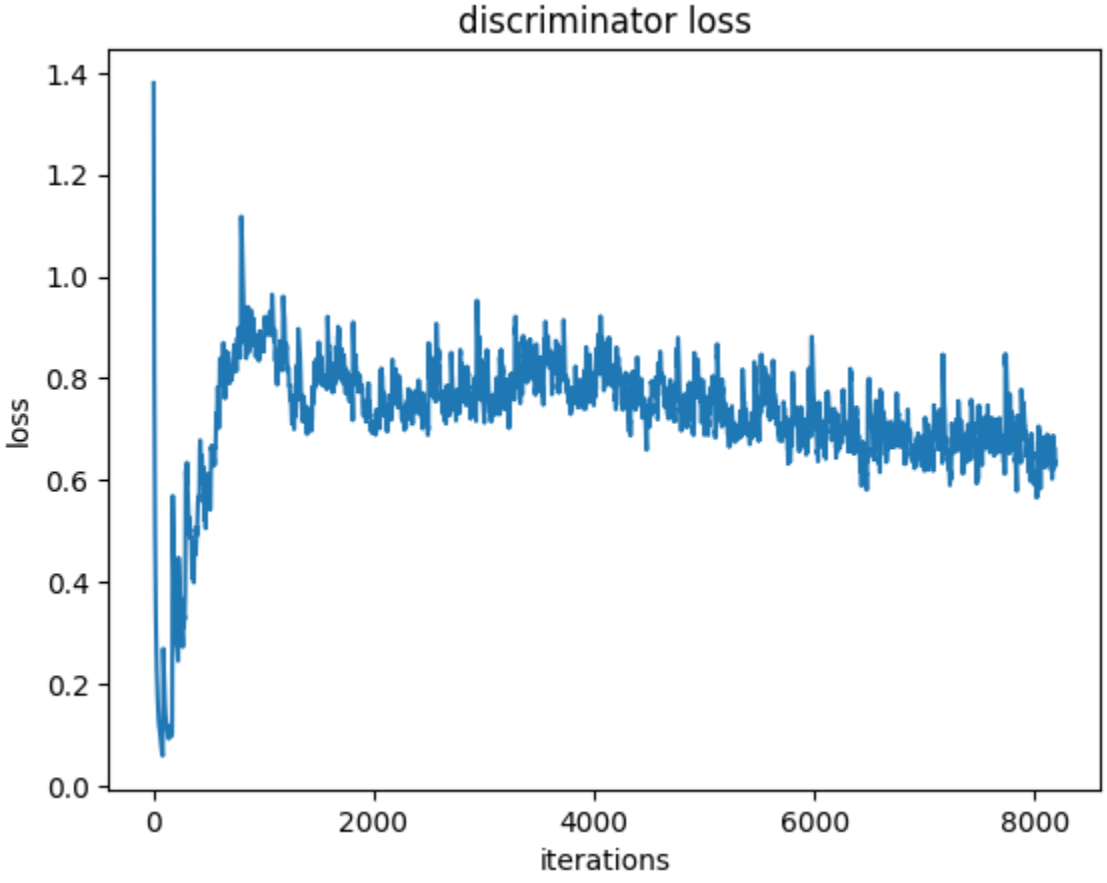


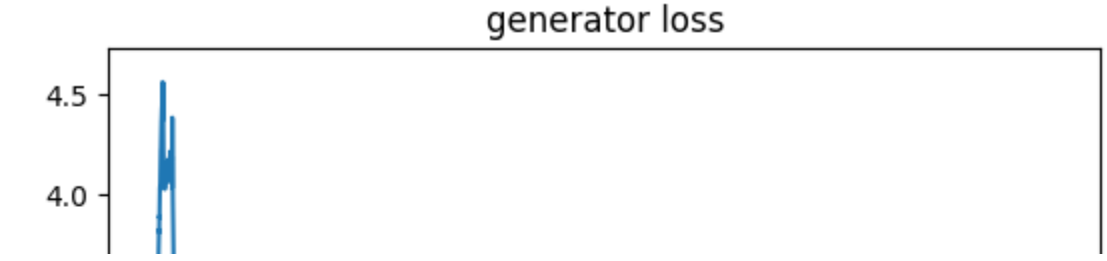


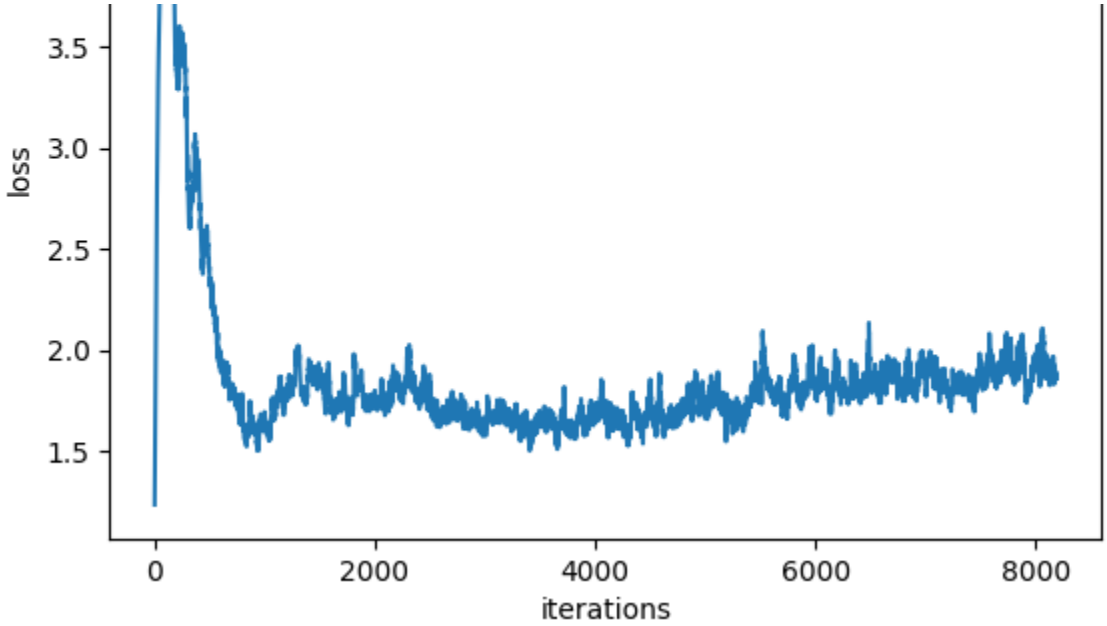

```
Iteration 7500/9750: dis loss = 0.5595, gen loss = 2.1637
Iteration 7600/9750: dis loss = 0.7655, gen loss = 1.3108
Iteration 7700/9750: dis loss = 0.7976, gen loss = 1.6350
Iteration 7800/9750: dis loss = 0.8008, gen loss = 1.2203
Iteration 7900/9750: dis loss = 1.6090, gen loss = 0.6501
Iteration 8000/9750: dis loss = 0.5009, gen loss = 1.5300
Iteration 8100/9750: dis loss = 1.1666, gen loss = 2.5836
```



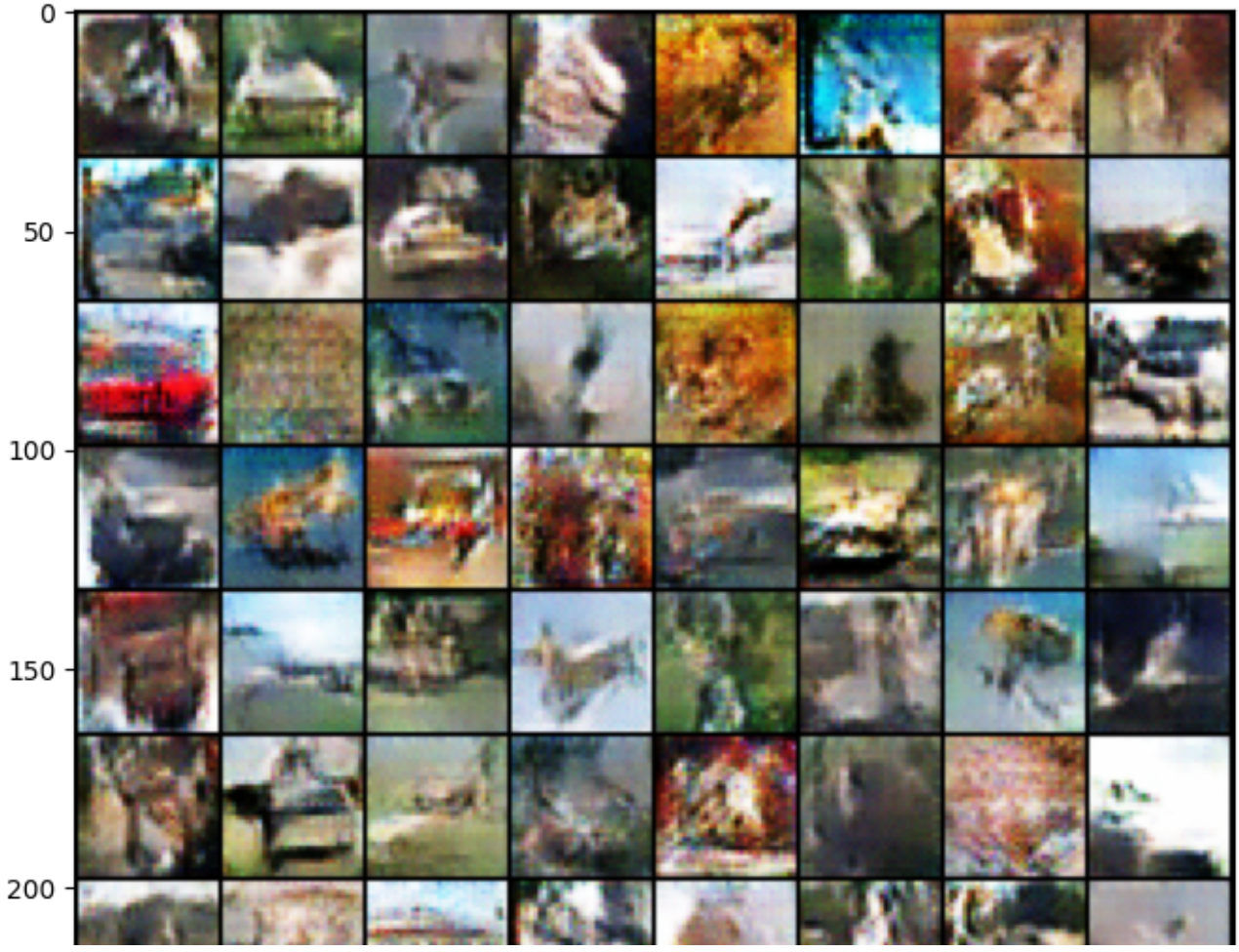

150
200
250
0
50
100
150
200
250

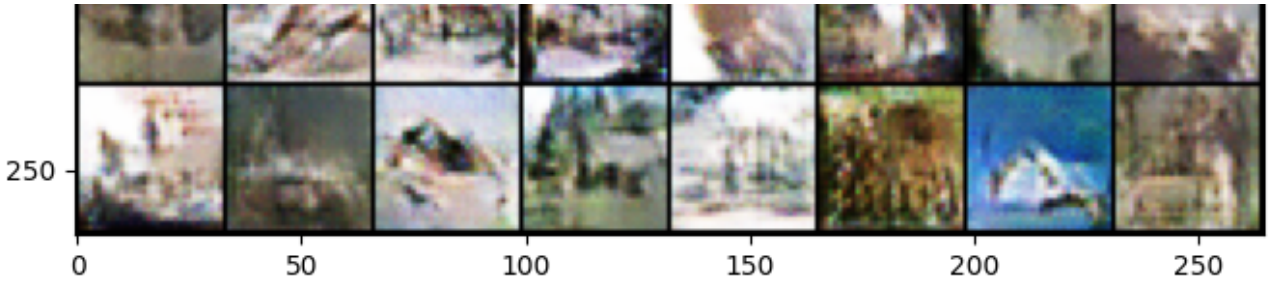

discriminator loss
loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
iterations

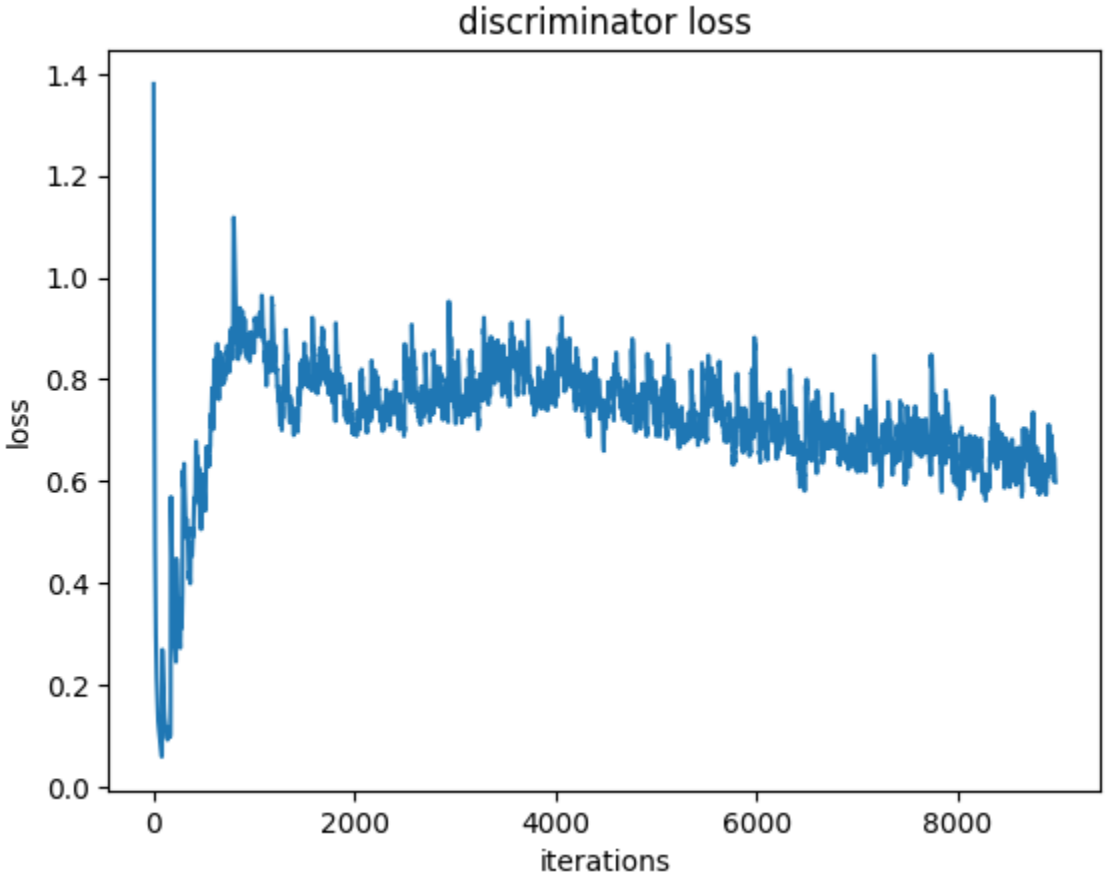

generator loss
4.5
4.0

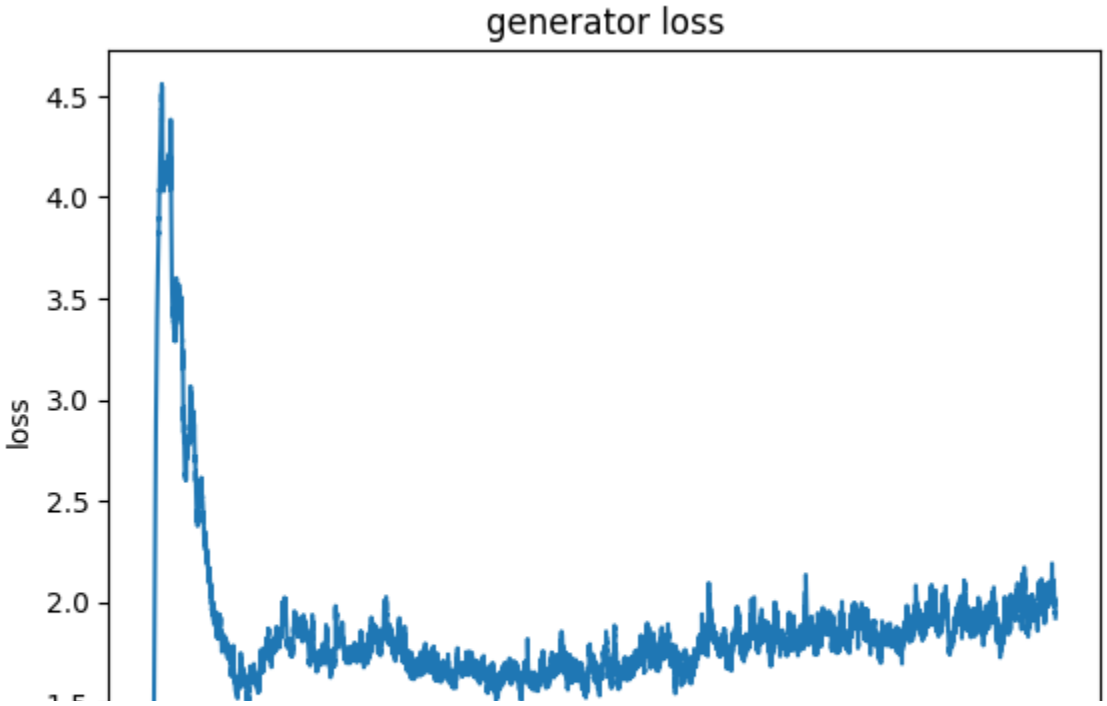


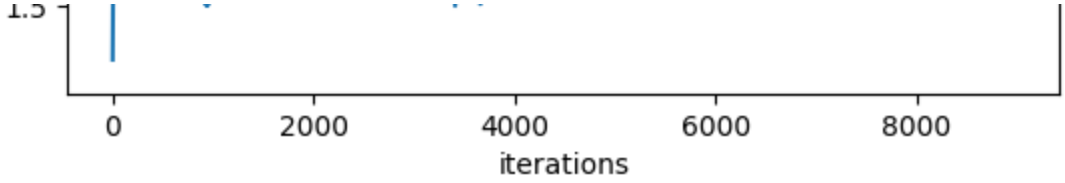

```
Iteration 8200/9750: dis loss = 0.4666, gen loss = 3.0762
Iteration 8300/9750: dis loss = 0.8024, gen loss = 1.0691
Iteration 8400/9750: dis loss = 0.5005, gen loss = 2.3425
Iteration 8500/9750: dis loss = 0.7630, gen loss = 1.8552
Iteration 8600/9750: dis loss = 0.5750, gen loss = 1.8206
Iteration 8700/9750: dis loss = 0.4592, gen loss = 3.1823
Iteration 8800/9750: dis loss = 0.5642, gen loss = 2.6651
Iteration 8900/9750: dis loss = 0.9257, gen loss = 1.2499
```



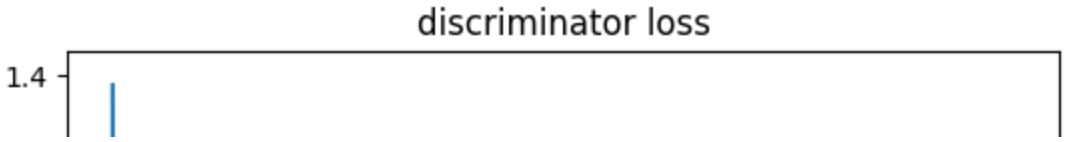

250
0
50
100
150
200
250

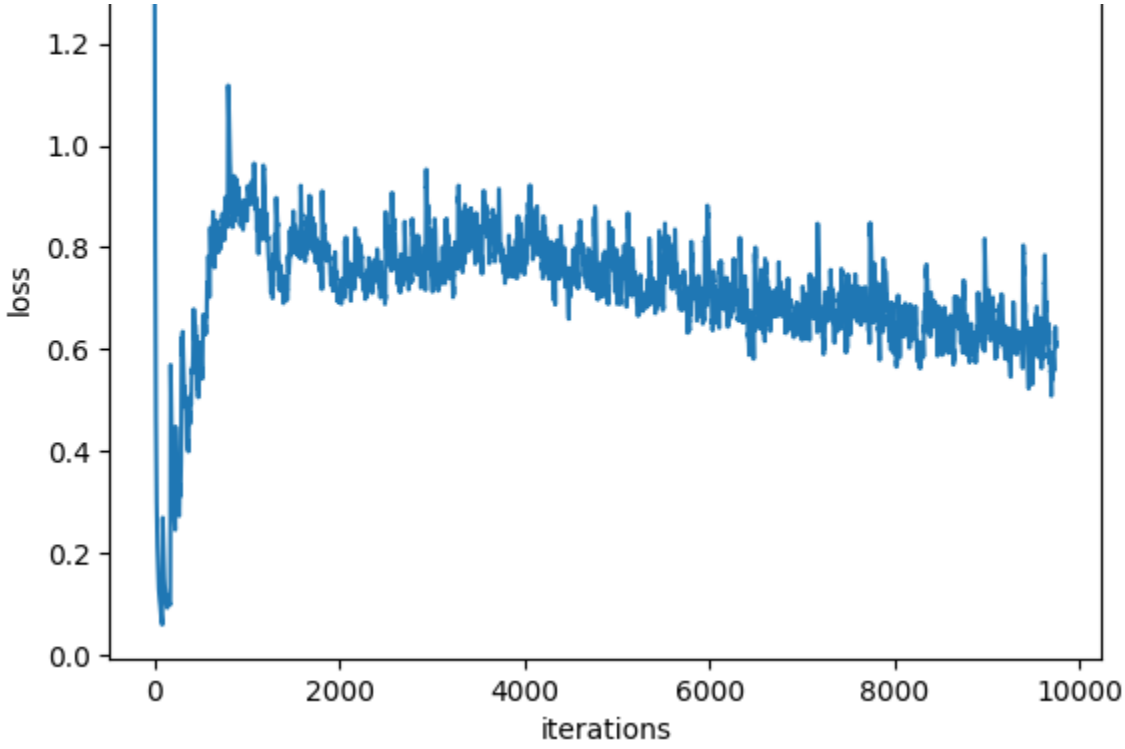

discriminator loss
loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
iterations

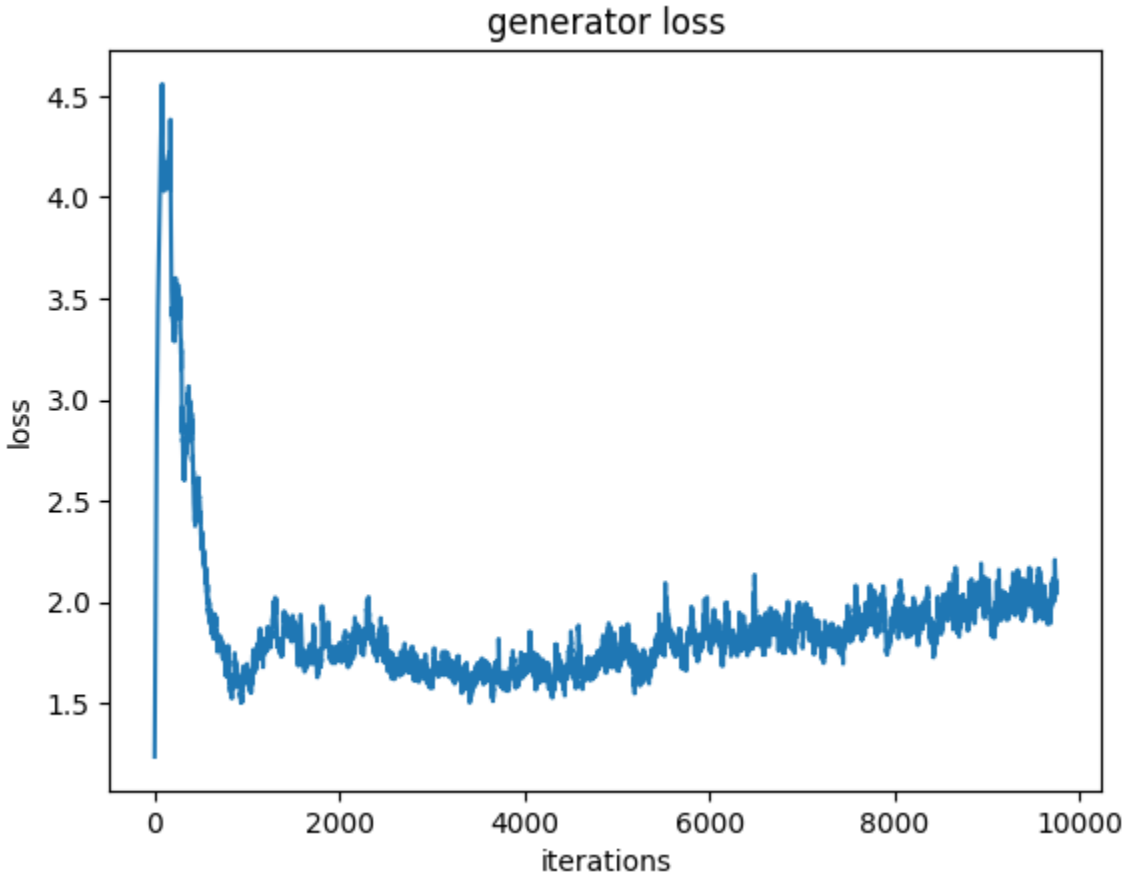

generator loss
loss
4.5
4.0
3.5
3.0
2.5
2.0




```
Iteration 9000/9750: dis loss = 0.3794, gen loss = 2.7300
Iteration 9100/9750: dis loss = 0.4358, gen loss = 2.9089
Iteration 9200/9750: dis loss = 0.5924, gen loss = 1.4115
Iteration 9300/9750: dis loss = 0.5951, gen loss = 2.3089
Iteration 9400/9750: dis loss = 0.7539, gen loss = 0.6882
Iteration 9500/9750: dis loss = 0.8511, gen loss = 1.4818
Iteration 9600/9750: dis loss = 0.8200, gen loss = 2.1135
Iteration 9700/9750: dis loss = 0.4894, gen loss = 3.2297
```




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


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

```
... 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))
```

```
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(
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:

target image:

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

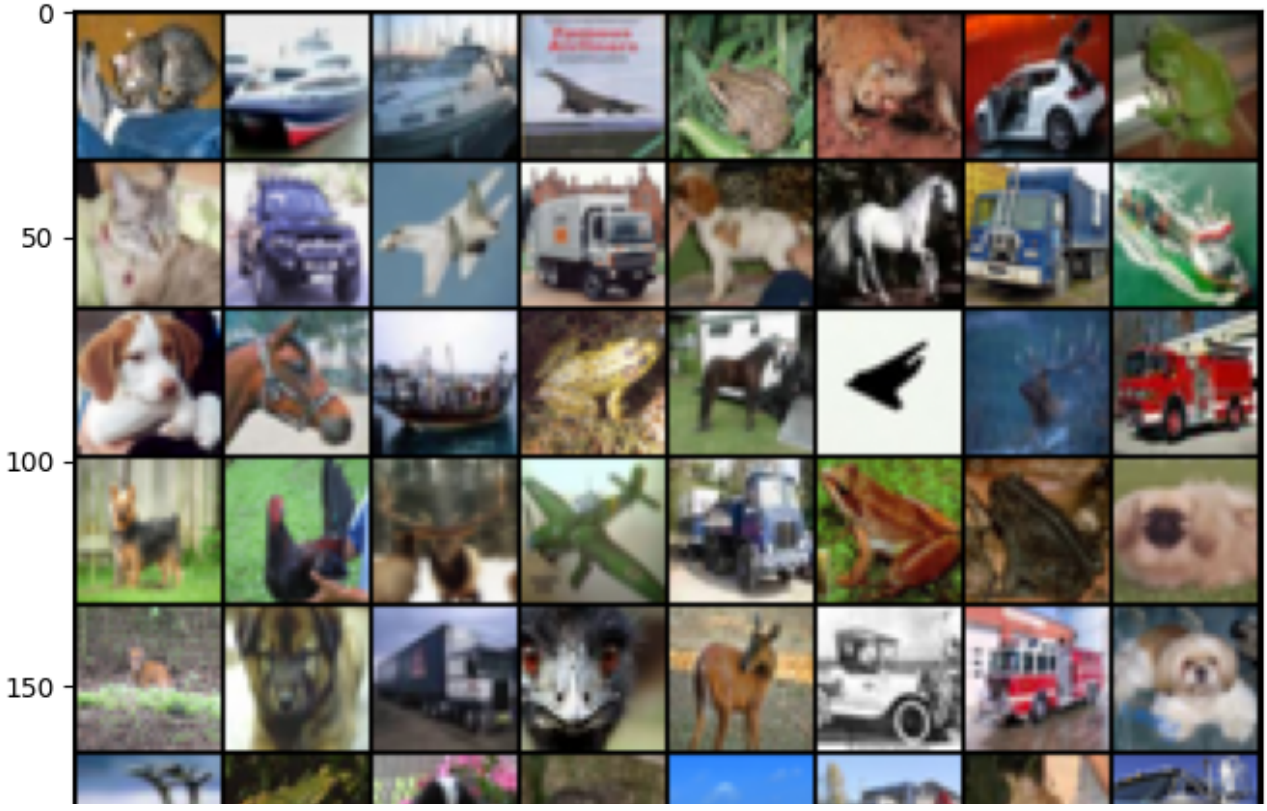
This time, we always start from a zero vector.

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

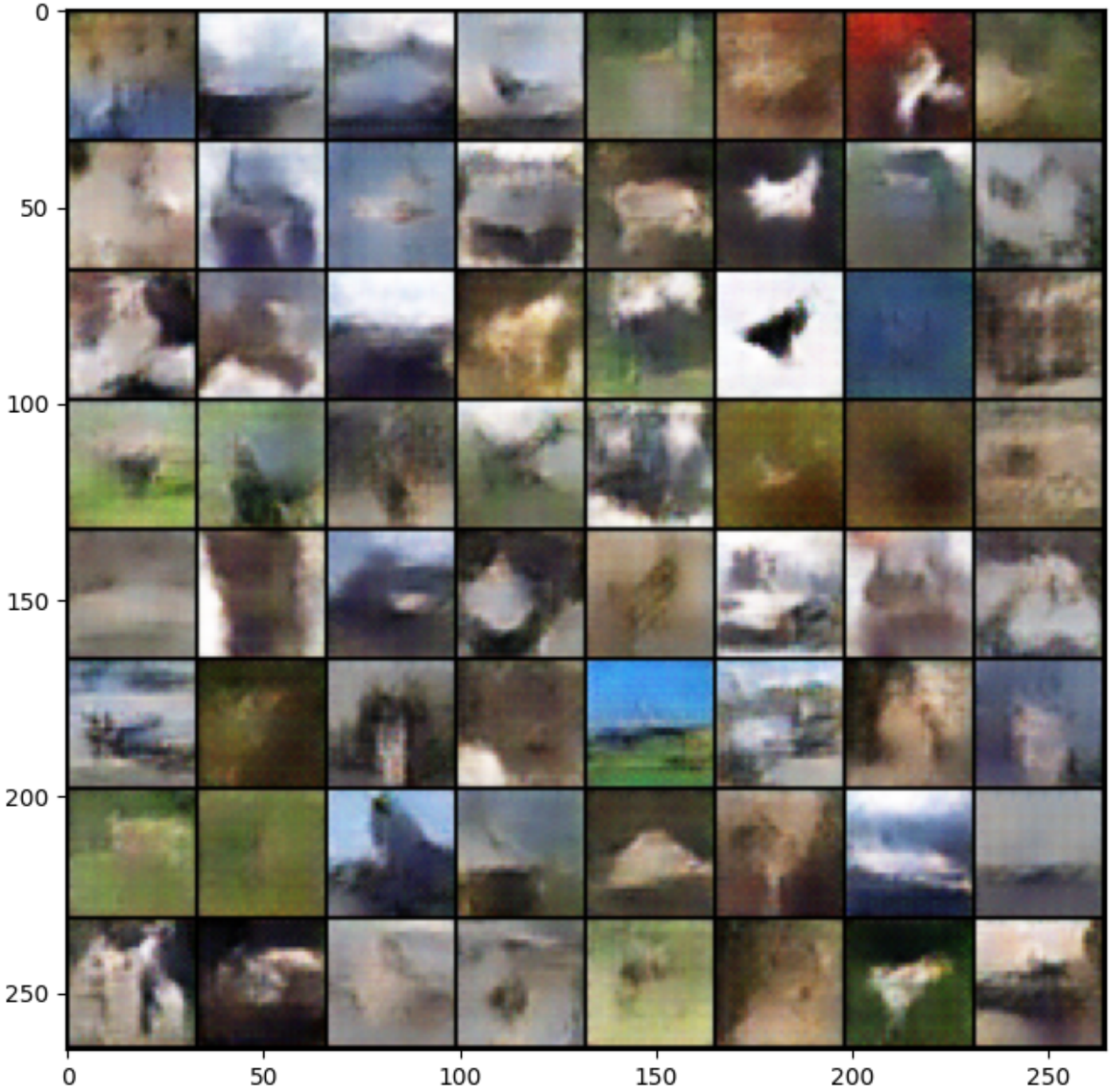
You need to achieve a reconstruction loss < 0.145. Do NOT modify anything outside of the blocks marked for you to fill in.

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

```
average reconstruction loss = 0.0139
```




200
250
0
50
100
150
200
250


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

Submission Instruction

See the pinned Piazza post for detailed instruction.