problem1-vae-v0

April 5, 2023

1 Problem 1 - Variational Auto-Encoder (VAE)

Variational Auto-Encoders (VAEs) are a widely used class of generative models. They are simple to implement and, in contrast to other generative model classes like Generative Adversarial Networks (GANs, see Problem 2), they optimize an explicit maximum likelihood objective to train the model. Finally, their architecture makes them well-suited for unsupervised representation learning, i.e., learning low-dimensional representations of high-dimenionsal inputs, like images, with only self-supervised objectives (data reconstruction in the case of VAEs).

(image source: https://mlexplained.com/2017/12/28/an-intuitive-explanation-of-variational-autoencoders-vaes-part-1)

By working on this problem you will learn and practice the following steps: 1. Set up a data loading pipeline in PyTorch. 2. Implement, train and visualize an auto-encoder architecture. 3. Extend your implementation to a variational auto-encoder. 4. Learn how to tune the critical beta parameter of your VAE. 5. Inspect the learned representation of your VAE. 6. Extend VAE's generative capabilities by conditioning it on the label you wish to generate.

Note: For faster training of the models in this assignment you can enable GPU support in this Colab. Navigate to "Runtime" –> "Change Runtime Type" and set the "Hardware Accelerator" to "GPU". However, you might hit compute limits of the colab free edition. Hence, you might want to debug locally (e.g. in a jupyter notebook) or in a CPU-only runtime on colab.

2 1. MNIST Dataset

We will perform all experiments for this problem using the MNIST dataset, a standard dataset of handwritten digits. The main benefits of this dataset are that it is small and relatively easy to model. It therefore allows for quick experimentation and serves as initial test bed in many papers.

Another benefit is that it is so widely used that PyTorch even provides functionality to automatically download it.

Let's start by downloading the data and visualizing some samples.

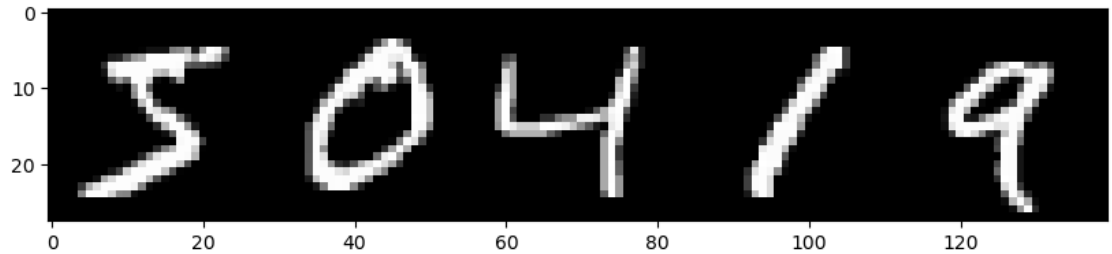
[1]:
```
import matplotlib.pyplot as plt
%matplotlib inline

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

```
%load_ext autoreload
%autoreload 2
```

[3]:
```
import torch
import torchvision
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')   # use␣
↪GPU if available
print(f"Using device: {device}")

# this will automatically download the MNIST training set
mnist_train = torchvision.datasets.MNIST(root='./data',
                                          train=True,
                                          download=True,
                                          transform=torchvision.transforms.
↪ToTensor())
print("\n Download complete! Downloaded {} training examples!".
↪format(len(mnist_train)))
```

```
Using device: cpu

 Download complete! Downloaded 60000 training examples!
```

[4]:
```
from numpy.random.mtrand import sample
import matplotlib.pyplot as plt
import numpy as np

# Let's display some of the training samples.
sample_images = []
randomize = False # set to False for debugging
num_samples = 5 # simple data sampling for now, later we will use proper␣
↪DataLoader
if randomize:
  sample_idxs = np.random.randint(low=0,high=len(mnist_train), size=num_samples)
else:
  sample_idxs = list(range(num_samples))

for idx in sample_idxs:
  sample = mnist_train[idx]
  # print(f"Tensor w/ shape {sample[0][0].detach().cpu().numpy().shape} and␣
↪label {sample[1]}")
  sample_images.append(sample[0][0].data.cpu().numpy())
  # print(sample_images[0]) # Values are in [0, 1]

fig = plt.figure(figsize = (10, 50))
ax1 = plt.subplot(111)
ax1.imshow(np.concatenate(sample_images, axis=1), cmap='gray')
plt.show()
```



3 2. Auto-Encoder

Before implementing the full VAE, we will first implement an **auto-encoder architecture**. Auto-encoders feature the same encoder-decoder architecture as VAEs and therefore also learn a low-dimensional representation of the input data without supervision. In contrast to VAEs they are **fully deterministic** models and do not employ variational inference for optimization.

The **architecture** is very simple: we will encode the input image into a low-dimensional representation using fully connected layers for the encoder. This results in a low-dimensional representation of the input image. This representation will get decoded back into the dimensionality of the input image using a decoder network that mirrors the architecture of the encoder. The whole model is trained by **minimizing a reconstruction loss** between the input and the decoded image.

Intuitively, the **auto-encoder needs to compress the information contained in the input image** into a much lower dimensional representation (e.g. 28x28=784px vs. nz embedding dimensions for our MNIST model). This is possible since the information captured in the pixels is *highly redundant*. E.g. encoding an MNIST image requires <4 bits to encode which of the 10 possible digits is displayed and a few additional bits to capture information about shape and orientation. This is much less than the $255^{28 \cdot 28}$ bits of information that could be theoretically captured in the input image.

Learning such a **compressed representation can make downstream task learning easier**. For example, learning to add two numbers based on the inferred digits is much easier than performing the task based on two piles of pixel values that depict the digits.

In the following, we will first define the architecture of encoder and decoder and then train the auto-encoder model.

3.1 Defining the Auto-Encoder Architecture [6pt]

[5]:
```
import torch.nn as nn

# Prob1-1: Let's define encoder and decoder networks
class Encoder(nn.Module):
  def __init__(self, nz, input_size):
    super().__init__()
    self.input_size = input_size
```

```
    ################################## TODO␣
↪#########################################
    # Create the network architecture using a nn.Sequential module wrapper.    ␣
↪   #
    # Encoder Architecture:                                                    ␣
↪   #
    # - input_size -> 256                                                      ␣
↪   #
    # - ReLU                                                                   ␣
↪   #
    # - 256 -> 64                                                              ␣
↪   #
    # - ReLU                                                                   ␣
↪   #
    # - 64 -> nz                                                               ␣
↪   #
    # HINT: Verify the shapes of intermediate layers by running partial␣
↪networks   #
    #        (with the next notebook cell) and visualizing the output shapes.  ␣
↪   #
  ␣
↪#################################################################################
    self.net = nn.Sequential(
      nn.Linear(self.input_size, 256),
      nn.ReLU(),
      nn.Linear(256, 64),
      nn.ReLU(),
      nn.Linear(64, nz),
      )
    ################################## END TODO␣
↪#########################################

  def forward(self, x):
    return self.net(x)


class Decoder(nn.Module):
  def __init__(self, nz, output_size):
    super().__init__()
    self.output_size = output_size
    ################################## TODO␣
↪#########################################
    # Create the network architecture using a nn.Sequential module wrapper.    ␣
↪   #
    # Decoder Architecture (mirrors encoder architecture):                     ␣
↪   #
```

```
    # - nz -> 64                                                                  ␣
↪   #
    # - ReLU                                                                      ␣
↪   #
    # - 64 -> 256                                                                 ␣
↪   #
    # - ReLU                                                                      ␣
↪   #
    # - 256 -> output_size                                                        ␣
↪   #
  ␣
↪################################################################################
    self.net = nn.Sequential(
      nn.Linear(nz, 64),
      nn.ReLU(),
      nn.Linear(64, 256),
      nn.ReLU(),
      nn.Linear(256, self.output_size)
    )
    ################################## END TODO␣
↪#######################################

  def forward(self, z):
    return self.net(z).reshape(-1, 1, self.output_size)
```

3.2 Testing the Auto-Encoder Forward Pass

[6]:
```
# To test your encoder/decoder, let's encode/decode some sample images
# first, make a PyTorch DataLoader object to sample data batches
batch_size = 64
nworkers = 2        # number of workers used for efficient data loading

################################################################################
# Create a PyTorch DataLoader object for efficiently generating training␣
↪batches.  #
# Make sure that the data loader automatically shuffles the training dataset.  ␣
↪   #
# Consider only *full* batches of data, to avoid torch errrors.              #
# The DataLoader wraps the MNIST dataset class we created earlier.           #
#       Use the given batch_size and number of data loading workers when␣
↪creating  #
#       the DataLoader. https://pytorch.org/docs/stable/data.html              ␣
↪   #
################################################################################
mnist_data_loader = torch.utils.data.DataLoader(mnist_train,
                                                batch_size=batch_size,
```

```
                                                    shuffle=True,
                                                    num_workers=nworkers,
                                                    drop_last=True)
##################################################################################

# now we can run a forward pass for encoder and decoder and check the produced␣
↪shapes
in_size = out_size = 28*28 # image size
nz = 32          # dimensionality of the learned embedding
encoder = Encoder(nz=nz, input_size=in_size)
decoder = Decoder(nz=nz, output_size=out_size)
for sample_img, sample_label in mnist_data_loader: # loads a batch of data
  input = sample_img.reshape([batch_size, in_size])
  print(f'{sample_img.shape=}, {type(sample_img)}, {input.shape=}')
  enc = encoder(input)
  print(f"Shape of encoding vector (should be [batch_size, nz]): {enc.shape}")
  dec = decoder(enc)
  print("Shape of decoded image (should be [batch_size, 1, out_size]): {}.".
↪format(dec.shape))
  break

del input, enc, dec, encoder, decoder, nworkers # remove to avoid confusion␣
↪later
```

```
sample_img.shape=torch.Size([64, 1, 28, 28]), <class 'torch.Tensor'>,
input.shape=torch.Size([64, 784])
Shape of encoding vector (should be [batch_size, nz]): torch.Size([64, 32])
Shape of decoded image (should be [batch_size, 1, out_size]): torch.Size([64, 1,
784]).
```

Now that we defined encoder and decoder network our architecture is nearly complete. However, before we start training, we can wrap encoder and decoder into an auto-encoder class for easier handling.

[7]:
```
class AutoEncoder(nn.Module):
  def __init__(self, nz):
    super().__init__()
    self.encoder = Encoder(nz=nz, input_size=in_size)
    self.decoder = Decoder(nz=nz, output_size=out_size)

  def forward(self, x):
    enc = self.encoder(x)
    return self.decoder(enc)

  def reconstruct(self, x):
    """Only used later for visualization."""
    enc = self.encoder(x)
    flattened = self.decoder(enc)
```

```
    image = flattened.reshape(-1, 28, 28)
    return image
```

3.3 Setting up the Auto-Encoder Training Loop [6pt]

After implementing the network architecture, we can now set up the training loop and run training.

[7]:
```
# Prob1-2
epochs = 10
learning_rate = 1e-3

# build AE model
print(f'Device available {device}')
ae_model = AutoEncoder(nz).to(device)    # transfer model to GPU if available
ae_model = ae_model.train()   # set model in train mode (eg batchnorm params␣
↪get updated)

# build optimizer and loss function
######################################## TODO␣
↪########################################
# Build the optimizer and loss classes. For the loss you can use a loss layer  ␣
↪    #
# from the torch.nn package. We recommend binary cross entropy.                ␣
↪    #
# HINT: We will use the Adam optimizer (learning rate given above, otherwise   ␣
↪    #
#       default parameters).                                                   ␣
↪    #
# NOTE: We could also use alternative losses like MSE and cross entropy,␣
↪depending #
#       on the assumptions we are making about the output distribution.        ␣
↪    #
##################################################################################
class NewLoss(nn.Module):
  def __init__(self) -> None:
    super().__init__()

  def forward(self, y, y_hat):
    #y_hat = torch.LongTensor(y_hat)
    criterion = nn.MSELoss()
    loss = criterion(y, y_hat)
    return loss

optimizer = torch.optim.Adam(ae_model.parameters())
criterion = NewLoss()
```

```
#################################### END TODO␣
↪#######################################

train_it = 0
for ep in range(epochs):
  print("Run Epoch {}".format(ep))
  ##################################### TODO␣
↪#######################################
  # Implement the main training loop for the auto-encoder model.                 ␣
↪     #
  # HINT: Your training loop should sample batches from the data loader, run␣
↪the     #
  #       forward pass of the AE, compute the loss, perform the backward pass␣
↪and    #
  #       perform one gradient step with the optimizer.                         ␣
↪     #
  # HINT: Don't forget to erase old gradients before performing the backward␣
↪pass.   #
 ␣
↪##########################################################################################
  loss = 0
  for indx, (sample_img, _) in enumerate(mnist_data_loader): # index and␣
↪img,label tuple
    optimizer.zero_grad()
    batch_features = sample_img.reshape(-1, 784)
    ae_output = ae_model.forward(batch_features)
    ae_output = ae_output.reshape(batch_size, 784)
    rec_loss = criterion(ae_output, batch_features)
    rec_loss.backward()
    optimizer.step()

    if train_it % 400 == 0:
      print("It {}: Reconstruction Loss: {}".format(train_it, rec_loss))
    train_it += 1
  #################################### END TODO␣
↪#######################################

print("Done!")
del epochs, learning_rate, sample_img, train_it, rec_loss #, opt
```

```
Device available cpu
Run Epoch 0
It 0: Reconstruction Loss: 0.11568797379732132
It 400: Reconstruction Loss: 0.027296032756567
It 800: Reconstruction Loss: 0.020810594782233238
Run Epoch 1
It 1200: Reconstruction Loss: 0.01801970973610878
```

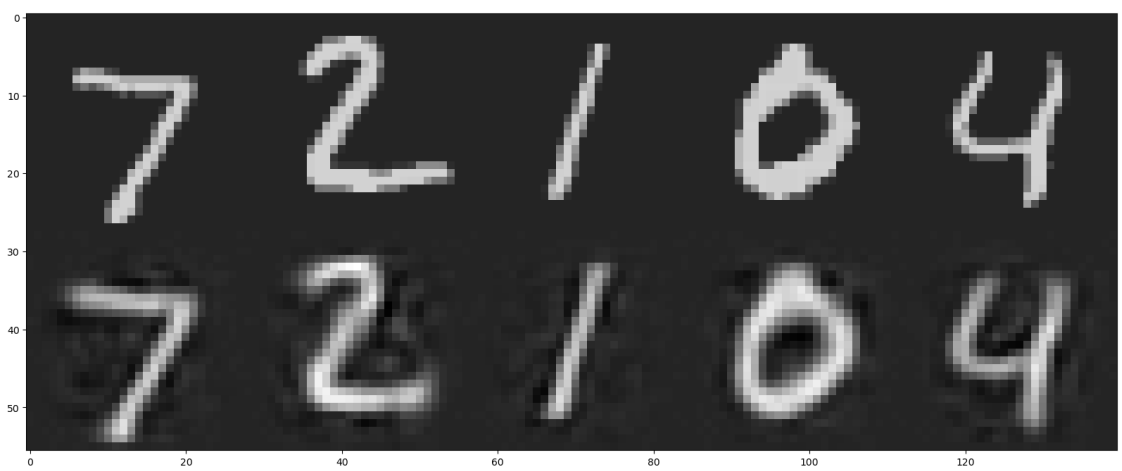
```
It 1600: Reconstruction Loss: 0.017932754009962082
Run Epoch 2
It 2000: Reconstruction Loss: 0.01710665039718151
It 2400: Reconstruction Loss: 0.015386519022285938
It 2800: Reconstruction Loss: 0.014879505150020123
Run Epoch 3
It 3200: Reconstruction Loss: 0.015120785683393478
It 3600: Reconstruction Loss: 0.013525244779884815
Run Epoch 4
It 4000: Reconstruction Loss: 0.01209457777440548
It 4400: Reconstruction Loss: 0.012310655787587166
Run Epoch 5
It 4800: Reconstruction Loss: 0.012896033003926277
It 5200: Reconstruction Loss: 0.012154248543083668
It 5600: Reconstruction Loss: 0.010981769300997257
Run Epoch 6
It 6000: Reconstruction Loss: 0.011262412182986736
It 6400: Reconstruction Loss: 0.011704298667609692
Run Epoch 7
It 6800: Reconstruction Loss: 0.011766199953854084
It 7200: Reconstruction Loss: 0.011411375366151333
Run Epoch 8
It 7600: Reconstruction Loss: 0.010474995709955692
It 8000: Reconstruction Loss: 0.010352786630392075
It 8400: Reconstruction Loss: 0.01106828823685646
Run Epoch 9
It 8800: Reconstruction Loss: 0.01062850933521986
It 9200: Reconstruction Loss: 0.011386428959667683
Done!
```

3.4 Verifying reconstructions

Now that we trained the auto-encoder we can visualize some of the reconstructions on the test set to verify that it is converged and did not overfit. **Before continuing, make sure that your auto-encoder is able to reconstruct these samples near-perfectly.**

[8]:
```
# visualize test data reconstructions
def vis_reconstruction(model, randomize=False):
  # download MNIST test set + build Dataset object
  mnist_test = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          download=True,
                                          transform=torchvision.transforms.
 ↪ToTensor())
  model.eval()      # set model in evalidation mode (eg freeze batchnorm params)
  num_samples = 5
  if randomize:
```

```
    sample_idxs = np.random.randint(low=0,high=len(mnist_test),␣
↪size=num_samples)
  else:
    sample_idxs = list(range(num_samples))

  input_imgs, test_reconstructions = [], []
  for idx in sample_idxs:
    sample = mnist_test[idx]
    input_img = np.asarray(sample[0])
    input_flat = input_img.reshape(784)
    reconstruction = model.reconstruct(torch.tensor(input_flat, device=device))

    input_imgs.append(input_img[0])
    test_reconstructions.append(reconstruction[0].data.cpu().numpy())
    # print(f'{input_img[0].shape=}\t{reconstruction.shape}')

  fig = plt.figure(figsize = (20, 50))
  ax1 = plt.subplot(111)
  ax1.imshow(np.concatenate([np.concatenate(input_imgs, axis=1),
                             np.concatenate(test_reconstructions, axis=1)],␣
↪axis=0), cmap='gray')
  plt.show()

vis_reconstruction(ae_model, randomize=False) # set randomize to False for␣
↪debugging
```



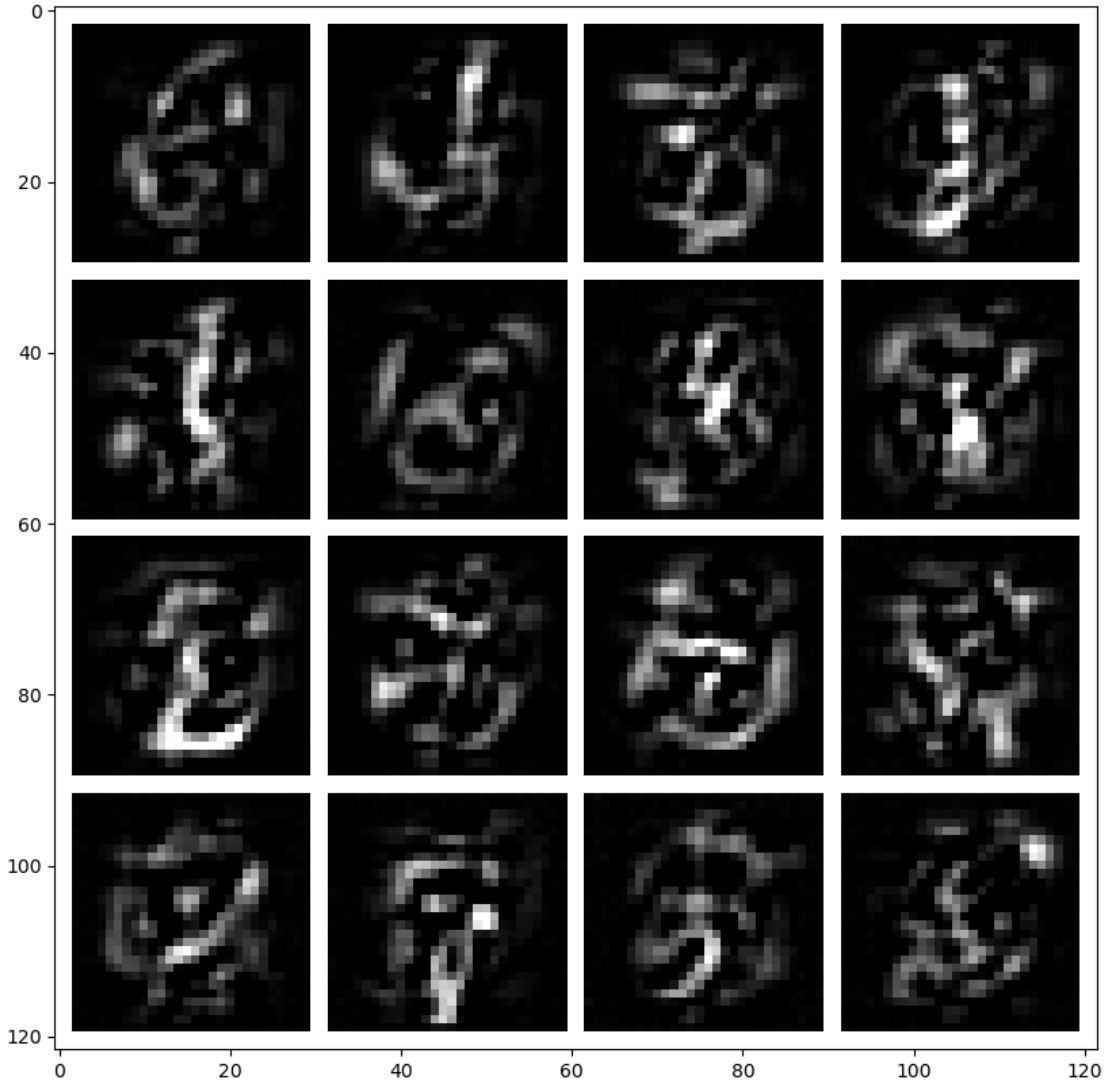
3.5 Sampling from the Auto-Encoder [2pt]

To test whether the auto-encoder is useful as a generative model, we can use it like any other generative model: draw embedding samples from a prior distribution and decode them through the decoder network. We will choose a unit Gaussian prior to allow for easy comparison to the VAE

later.

[9]:
```
# we will sample N embeddings, then decode and visualize them
def vis_samples(model):
  ##################################### TODO␣
↪#####################################
  # Prob1-3 Sample embeddings from a diagonal unit Gaussian distribution and␣
↪decode them     #
  # using the model.                                                        ␣
↪     #
  # HINT: The sampled embeddings should have shape [batch_size, nz]. Diagonal␣
↪unit   #
  #       Gaussians have mean 0 and a covariance matrix with ones on the␣
↪diagonal    #
  #       and zeros everywhere else.                                        ␣
↪     #
  # HINT: If you are unsure whether you sampled the correct distribution, you␣
↪can    #
  #       sample a large batch and compute the empirical mean and variance␣
↪using the #
  #       .mean() and .var() functions.                                     ␣
↪     #
  # HINT: You can directly use model.decoder() to decode the samples.       ␣
↪     #
 ␣
↪##################################################################################
  embeddings = torch.empty((batch_size, nz)).normal_()
  decoded_samples = model.decoder(embeddings).reshape(-1,1,28,28)
  ################################### END TODO␣
↪#####################################

  fig = plt.figure(figsize = (10, 10))
  ax1 = plt.subplot(111)
  ax1.imshow(torchvision.utils.make_grid(decoded_samples[:16], nrow=4,␣
↪pad_value=1.)\
                .data.cpu().numpy().transpose(1, 2, 0), cmap='gray')
  plt.show()

vis_samples(ae_model)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



Prob1-3 continued: Inline Question: Describe your observations, why do you think they occur? [2pt] (max 150 words)

Sampling from a latent space that isn't a guassian distribution forces you to pick points of the LS that does not directly relate to the trained data, thus reconstuction of this sampling will lead to jumbled output.

4 3. Variational Auto-Encoder (VAE)

Variational auto-encoders use a very similar architecture to deterministic auto-encoders, but are inherently storchastic models, i.e. we perform a stochastic sampling operation during the forward pass, leading to different different outputs every time we run the network for the same input. This sampling is required to optimize the VAE objective also known as the evidence lower bound (ELBO):

$$p(x) > \underbrace{\mathbb{E}_{z\sim q(z|x)} p(x|z)}_{\text{reconstruction}} - \underbrace{D_{\mathrm{KL}}(q(z|x), p(z))}_{\text{prior divergence}}$$

Here, $D_{\mathrm{KL}}(q, p)$ denotes the Kullback-Leibler (KL) divergence between the posterior distribution $q(z|x)$, i.e. the output of our encoder, and $p(z)$, the prior over the embedding variable z, which we can choose freely.

For simplicity, we will choose a unit Gaussian prior again. The first term is the reconstruction term we already know from training the auto-encoder. When assuming a Gaussian output distribution for both encoder $q(z|x)$ and decoder $p(x|z)$ the objective reduces to:

$$\mathcal{L}_{\mathrm{VAE}} = \sum_{x\sim\mathcal{D}} \mathcal{L}_{\mathrm{rec}}(x, \hat{x}) - \beta \cdot D_{\mathrm{KL}}(\mathcal{N}(\mu_q, \sigma_q), \mathcal{N}(0, I))$$

Here, $\hat{x}$ is the reconstruction output of the decoder. In comparison to the auto-encoder objective, the VAE adds a regularizing term between the output of the encoder and a chosen prior distribution, effectively forcing the encoder output to not stray too far from the prior during training. As a result the decoder gets trained with samples that look pretty similar to samples from the prior, which will hopefully allow us to generate better images when using the VAE as a generative model and actually feeding it samples from the prior (as we have done for the AE before).

The coefficient β is a scalar weighting factor that trades off between reconstruction and regularization objective. We will investigate the influence of this factor in out experiments below.

If you need a refresher on VAEs you can check out this tutorial paper: https://arxiv.org/abs/1606.05908

4.0.1 Reparametrization Trick

The sampling procedure inside the VAE's forward pass for obtaining a sample z from the posterior distribution $q(z|x)$, when implemented naively, is non-differentiable. However, since $q(z|x)$ is parametrized with a Gaussian function, there is a simple trick to obtain a differentiable sampling operator, known as the *reparametrization trick*.

Instead of directly sampling $z \sim \mathcal{N}(\mu_q, \sigma_q)$ we can "separate" the network's predictions and the random sampling by computing the sample as:

$$z = \mu_q + \sigma_q * \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Note that in this equation, the sample z is computed as a deterministic function of the network's predictions μ_q and σ_q and therefore allows to propagate gradients through the sampling procedure.

Note: While in the equations above the encoder network parametrizes the standard deviation σ_q of the Gaussian posterior distribution, in practice we usually parametrize the **logarithm of the standard deviation** $\log \sigma_q$ for numerical stability. Before sampling z we will then exponentiate the network's output to obtain σ_q.

4.1 Defining the VAE Model [7pt]

[10]:
```
def kl_divergence(mu1, log_sigma1, mu2, log_sigma2):
  """Computes KL[p||q] between two Gaussians defined by [mu, log_sigma]."""
  return (log_sigma2 - log_sigma1) + (torch.exp(log_sigma1) ** 2 + (mu1 - mu2)␣
 ↪** 2) \
               / (2 * torch.exp(log_sigma2) ** 2) - 0.5

# Prob1-4
class VAE(nn.Module):
  def __init__(self, nz, beta=1.0):
    super().__init__()
    self.beta = beta          # factor trading off between two loss components
    ######################################## TODO␣
 ↪########################################
    # Instantiate Encoder and Decoder.                                        ␣
 ↪        #
    # HINT: Remember that the encoder is now parametrizing a Gaussian␣
 ↪distribution's   #
    #       mean and log_sigma, so the dimensionality of the output needs to   ␣
 ↪        #
    #       double. The decoder works with an embedding sampled from this␣
 ↪output.  #
   ␣
 ↪##########################################################################################
    self.encoder = Encoder(2 * nz, in_size)
    self.decoder = Decoder(nz, out_size)
    self.nz = nz
    ##################################### END TODO␣
 ↪#####################################

  def forward(self, x):
    ######################################## TODO␣
 ↪########################################
    # Implement the forward pass of the VAE.                                  ␣
 ↪        #
    # HINT: Your code should implement the following steps:                   ␣
 ↪        #
    #          1. encode input x, split encoding into mean and log_sigma of␣
 ↪Gaussian   #
    #          2. sample z from inferred posterior distribution using          ␣
 ↪        #
    #             reparametrization trick                                      ␣
 ↪        #
    #          3. decode the sampled z to obtain the reconstructed image       ␣
 ↪        #
```

```
 ␣
↪##############################################################################
  q = self.encoder(x)
  mu, log_sigma = torch.chunk(q, chunks=2, dim=1)
  eps = torch.randn_like(mu)
  q = mu + torch.exp(0.5*log_sigma)*eps
  reconstruction = self.decoder(q)
  ##################################### END TODO␣
↪######################################

  return {'q': q,
          'rec': reconstruction}

 def loss(self, x, outputs):
  ######################################### TODO␣
↪########################################
  # Implement the loss computation of the VAE.                                ␣
↪       #
  # HINT: Your code should implement the following steps:                     ␣
↪       #
  #          1. compute the image reconstruction loss, similar to AE loss␣
↪above      #
  #          2. compute the KL divergence loss between the inferred posterior␣
↪       #
  #             distribution and a unit Gaussian prior; you can use the␣
↪provided     #
  #             function above for computing the KL divergence between two␣
↪Gaussians #
  #             parametrized by mean and log_sigma                            ␣
↪       #
  # HINT: Make sure to compute the KL divergence in the correct order since␣
↪it is    #
  #       not symmetric!!  ie. KL(p, q) != KL(q, p)                           ␣
↪       #
 ␣
↪##############################################################################
  rec_loss = nn.functional.mse_loss(outputs['rec'].squeeze(), x)
  mu, log_sigma = torch.chunk(outputs['q'], chunks=2, dim=-1)
  kl_loss = kl_divergence(mu, log_sigma, torch.zeros_like(mu), torch.
↪zeros_like(log_sigma)).mean()

  ##################################### END TODO␣
↪######################################

  # return weighted objective
  return rec_loss + self.beta * kl_loss, \
```

```
            {'rec_loss': rec_loss, 'kl_loss': kl_loss}

    def reconstruct(self, x):
        """Use mean of posterior estimate for visualization reconstruction."""
        ######################################## TODO␣
↪########################################
        # This function is used for visualizing reconstructions of our VAE model.␣
↪To        #
        # obtain the maximum likelihood estimate we bypass the sampling procedure␣
↪of the    #
        # inferred latent and instead directly use the mean of the inferred␣
↪posterior.      #
        # HINT: encode the input image and then decode the mean of the posterior to␣
↪obtain #
        #       the reconstruction.                                                 ␣
↪       #
        ␣
↪##########################################################################################
        encoding = self.encoder(x)
        mean, _ = torch.chunk(encoding, chunks=2, dim=-1)
        image = self.decoder(mean).reshape(-1,28,28)

        ##################################### END TODO␣
↪#####################################
        return image
```

4.2 Setting up the VAE Training Loop [4pt]

Let's start training the VAE model! We will first verify our implementation by setting $\beta = 0$.

[11]:
```
# Prob1-5 VAE training loop
learning_rate = 1e-3
nz = 32
beta = 0

######################################## TODO␣
↪########################################
epochs = 20       # recommended 5-20 epochs
##################################### END TODO␣
↪########################################

# build VAE model
vae_model = VAE(nz, beta).to(device)     # transfer model to GPU if available
vae_model = vae_model.train()    # set model in train mode (eg batchnorm params␣
↪get updated)
```

```
# build optimizer and loss function
##################################### TODO␣
↪#####################################
# Build the optimizer for the vae_model. We will again use the Adam optimizer␣
↪with #
# the given learning rate and otherwise default parameters.                  ␣
↪    #
##################################################################################
# same as AE
from torch.optim import Adam
optimizer = Adam(vae_model.parameters(), lr=learning_rate)

################################### END TODO␣
↪#####################################

train_it = 0
rec_loss, kl_loss = [], []
print(f"Running {epochs} epochs with {beta=}")
for ep in range(epochs):
  print("Run Epoch {}".format(ep))
  ##################################### TODO␣
↪#####################################
  # Implement the main training loop for the VAE model.                       ␣
↪      #
  # HINT: Your training loop should sample batches from the data loader, run␣
↪the      #
  #       forward pass of the VAE, compute the loss, perform the backward pass␣
↪and    #
  #       perform one gradient step with the optimizer.                       ␣
↪      #
  # HINT: Don't forget to erase old gradients before performing the backward␣
↪pass.    #
  # HINT: This time we will use the loss() function of our model for computing␣
↪the    #
  #       training loss. It outputs the total training loss and a dict␣
↪containing     #
  #       the breakdown of reconstruction and KL loss.                        ␣
↪      #
 ␣
↪##################################################################################
  for indx, (img, _) in enumerate(mnist_data_loader): # index and img,label␣
↪tuple
    optimizer.zero_grad()
    batch_features = torch.reshape(img, (-1, 784) ).to(device)
    vae_output = vae_model(batch_features)
    total_loss, losses = vae_model.loss(batch_features, outputs=vae_output)
```

```
    rec_loss.append(losses['rec_loss']); kl_loss.append(losses['kl_loss'])
    total_loss.backward()
    optimizer.step()

    if train_it % 100 == 0:
      print("It {}: Total Loss: {}, \t Rec Loss: {},\t KL Loss: {}"\
            .format(train_it, total_loss, losses['rec_loss'],␣
 ↪losses['kl_loss']))
    train_it += 1
  ##################################### END TODO␣
 ↪#####################################

print("Done!")

rec_loss_plotdata = [foo.detach().cpu() for foo in rec_loss]
kl_loss_plotdata = [foo.detach().cpu() for foo in kl_loss]

# log the loss training curves
fig = plt.figure(figsize = (10, 5))
ax1 = plt.subplot(121)
ax1.plot(rec_loss_plotdata)
ax1.title.set_text("Reconstruction Loss")
ax2 = plt.subplot(122)
ax2.plot(kl_loss_plotdata)
ax2.title.set_text("KL Loss")
plt.show()
```

```
Running 20 epochs with beta=0
Run Epoch 0
It 0: Total Loss: 0.1242644190788269,    Rec Loss: 0.1242644190788269,   KL
Loss: 4.744263172149658
It 100: Total Loss: 0.049549590796232224,        Rec Loss: 0.049549590796232224,
KL Loss: 978.352294921875
It 200: Total Loss: 0.038692645728588104,        Rec Loss: 0.038692645728588104,
KL Loss: 47654.13671875
It 300: Total Loss: 0.0327465683221817,          Rec Loss: 0.0327465683221817,
KL Loss: 694396.875
It 400: Total Loss: 0.030155351385474205,        Rec Loss: 0.030155351385474205,
KL Loss: 6488852.0
It 500: Total Loss: 0.026437075808644295,        Rec Loss: 0.026437075808644295,
KL Loss: 17573908.0
It 600: Total Loss: 0.02729414589703083,         Rec Loss: 0.02729414589703083,
KL Loss: 5866895.0
It 700: Total Loss: 0.02271331660449505,         Rec Loss: 0.02271331660449505,
KL Loss: 2222515.75
It 800: Total Loss: 0.021837059408426285,        Rec Loss: 0.021837059408426285,
KL Loss: 812256.1875
```

```
It 900: Total Loss: 0.020509490743279457,        Rec Loss: 0.020509490743279457,
KL Loss: 711745.5625
Run Epoch 1
It 1000: Total Loss: 0.02058345265686512,        Rec Loss: 0.02058345265686512,
KL Loss: 3583062.25
It 1100: Total Loss: 0.022461935877799988,       Rec Loss: 0.022461935877799988,
KL Loss: 2719116.25
It 1200: Total Loss: 0.021202847361564636,       Rec Loss: 0.021202847361564636,
KL Loss: 960982.9375
It 1300: Total Loss: 0.022298794239759445,       Rec Loss: 0.022298794239759445,
KL Loss: 725065.125
It 1400: Total Loss: 0.020536018535494804,       Rec Loss: 0.020536018535494804,
KL Loss: 621433.0625
It 1500: Total Loss: 0.019685322418808937,       Rec Loss: 0.019685322418808937,
KL Loss: 1344120.75
It 1600: Total Loss: 0.019523246213793755,       Rec Loss: 0.019523246213793755,
KL Loss: 2480935.75
It 1700: Total Loss: 0.018986279144883156,       Rec Loss: 0.018986279144883156,
KL Loss: 888752.9375
It 1800: Total Loss: 0.018453802913427353,       Rec Loss: 0.018453802913427353,
KL Loss: 8266831.5
Run Epoch 2
It 1900: Total Loss: 0.018476692959666252,       Rec Loss: 0.018476692959666252,
KL Loss: 552084.8125
It 2000: Total Loss: 0.01819927804172039,        Rec Loss: 0.01819927804172039,
KL Loss: 728702.5
It 2100: Total Loss: 0.01886877417564392,        Rec Loss: 0.01886877417564392,
KL Loss: 1155146.875
It 2200: Total Loss: 0.016008200123906136,       Rec Loss: 0.016008200123906136,
KL Loss: 748378.0625
It 2300: Total Loss: 0.016814954578876495,       Rec Loss: 0.016814954578876495,
KL Loss: 49302.71484375
It 2400: Total Loss: 0.016183925792574883,       Rec Loss: 0.016183925792574883,
KL Loss: 340830.5625
It 2500: Total Loss: 0.017701776698231697,       Rec Loss: 0.017701776698231697,
KL Loss: 299363.625
It 2600: Total Loss: 0.01706886850297451,        Rec Loss: 0.01706886850297451,
KL Loss: 182544.390625
It 2700: Total Loss: 0.01602717489004135,        Rec Loss: 0.01602717489004135,
KL Loss: 69007.1953125
It 2800: Total Loss: 0.017048509791493416,       Rec Loss: 0.017048509791493416,
KL Loss: 244752.625
Run Epoch 3
It 2900: Total Loss: 0.016732823103666306,       Rec Loss: 0.016732823103666306,
KL Loss: 80967.6640625
It 3000: Total Loss: 0.014889220707118511,       Rec Loss: 0.014889220707118511,
KL Loss: 38852.625
It 3100: Total Loss: 0.01722986064851284,        Rec Loss: 0.01722986064851284,
```

```
KL Loss: 11479.51171875
It 3200: Total Loss: 0.01715623028576374,        Rec Loss: 0.01715623028576374,
KL Loss: 564055.9375
It 3300: Total Loss: 0.017447685822844505,       Rec Loss: 0.017447685822844505,
KL Loss: 111306.015625
It 3400: Total Loss: 0.015615120530128479,       Rec Loss: 0.015615120530128479,
KL Loss: 23300.72265625
It 3500: Total Loss: 0.015373713336884975,       Rec Loss: 0.015373713336884975,
KL Loss: 251236.46875
It 3600: Total Loss: 0.013711587525904179,       Rec Loss: 0.013711587525904179,
KL Loss: 15808.10546875
It 3700: Total Loss: 0.014622035436332226,       Rec Loss: 0.014622035436332226,
KL Loss: 80929.1953125
Run Epoch 4
It 3800: Total Loss: 0.015135692432522774,       Rec Loss: 0.015135692432522774,
KL Loss: 4003.593994140625
It 3900: Total Loss: 0.014481551945209503,       Rec Loss: 0.014481551945209503,
KL Loss: 1952.867431640625
It 4000: Total Loss: 0.013704518787562847,       Rec Loss: 0.013704518787562847,
KL Loss: 44729.32421875
It 4100: Total Loss: 0.015234052203595638,       Rec Loss: 0.015234052203595638,
KL Loss: 17014.849609375
It 4200: Total Loss: 0.014988962560892105,       Rec Loss: 0.014988962560892105,
KL Loss: 950.2340087890625
It 4300: Total Loss: 0.015215558931231499,       Rec Loss: 0.015215558931231499,
KL Loss: 26190.6015625
It 4400: Total Loss: 0.014683406800031662,       Rec Loss: 0.014683406800031662,
KL Loss: 16294.732421875
It 4500: Total Loss: 0.014447521418333054,       Rec Loss: 0.014447521418333054,
KL Loss: 5388.63623046875
It 4600: Total Loss: 0.01350090466439724,        Rec Loss: 0.01350090466439724,
KL Loss: 17717.23046875
Run Epoch 5
It 4700: Total Loss: 0.014414806850254536,       Rec Loss: 0.014414806850254536,
KL Loss: 3600.187255859375
It 4800: Total Loss: 0.01456838846206665,        Rec Loss: 0.01456838846206665,
KL Loss: 4282.962890625
It 4900: Total Loss: 0.013125188648700714,       Rec Loss: 0.013125188648700714,
KL Loss: 1495.1343994140625
It 5000: Total Loss: 0.013626646250486374,       Rec Loss: 0.013626646250486374,
KL Loss: 7453.5419921875
It 5100: Total Loss: 0.015001221559941769,       Rec Loss: 0.015001221559941769,
KL Loss: 8415.4453125
It 5200: Total Loss: 0.013772460632026196,       Rec Loss: 0.013772460632026196,
KL Loss: 5286.09716796875
It 5300: Total Loss: 0.01325035747140646,        Rec Loss: 0.01325035747140646,
KL Loss: 1938.2479248046875
It 5400: Total Loss: 0.014489388093352318,       Rec Loss: 0.014489388093352318,
```

```
KL Loss: 1496.65966796875
It 5500: Total Loss: 0.013161592185497284,       Rec Loss: 0.013161592185497284,
KL Loss: 1022.85693359375
It 5600: Total Loss: 0.012591633014380932,       Rec Loss: 0.012591633014380932,
KL Loss: 1012.8363647460938
Run Epoch 6
It 5700: Total Loss: 0.013203306123614311,       Rec Loss: 0.013203306123614311,
KL Loss: 27374.7421875
It 5800: Total Loss: 0.013353077694773674,       Rec Loss: 0.013353077694773674,
KL Loss: 5988.39208984375
It 5900: Total Loss: 0.012875284068286419,       Rec Loss: 0.012875284068286419,
KL Loss: 4671.3486328125
It 6000: Total Loss: 0.012168851681053638,       Rec Loss: 0.012168851681053638,
KL Loss: 1670.0750732421875
It 6100: Total Loss: 0.012912971898913383,       Rec Loss: 0.012912971898913383,
KL Loss: 3513.13427734375
It 6200: Total Loss: 0.012767568230628967,       Rec Loss: 0.012767568230628967,
KL Loss: 3466.465576171875
It 6300: Total Loss: 0.01216390635818243,        Rec Loss: 0.01216390635818243,
KL Loss: 3677.28515625
It 6400: Total Loss: 0.012717334553599358,       Rec Loss: 0.012717334553599358,
KL Loss: 2083.630615234375
It 6500: Total Loss: 0.011712240986526012,       Rec Loss: 0.011712240986526012,
KL Loss: 2617.13623046875
Run Epoch 7
It 6600: Total Loss: 0.012775642797350883,       Rec Loss: 0.012775642797350883,
KL Loss: 2507.7646484375
It 6700: Total Loss: 0.012645890936255455,       Rec Loss: 0.012645890936255455,
KL Loss: 821.348876953125
It 6800: Total Loss: 0.013292786665260792,       Rec Loss: 0.013292786665260792,
KL Loss: 994.18212890625
It 6900: Total Loss: 0.012208187952637672,       Rec Loss: 0.012208187952637672,
KL Loss: 791.3875122070312
It 7000: Total Loss: 0.013235379010438919,       Rec Loss: 0.013235379010438919,
KL Loss: 984.2493896484375
It 7100: Total Loss: 0.012620492838323116,       Rec Loss: 0.012620492838323116,
KL Loss: 3074.705322265625
It 7200: Total Loss: 0.012726779095828533,       Rec Loss: 0.012726779095828533,
KL Loss: 548.4358520507812
It 7300: Total Loss: 0.01182759739458561,        Rec Loss: 0.01182759739458561,
KL Loss: 1999.9425048828125
It 7400: Total Loss: 0.012447357177734375,       Rec Loss: 0.012447357177734375,
KL Loss: 1623.9483642578125
Run Epoch 8
It 7500: Total Loss: 0.011707736179232597,       Rec Loss: 0.011707736179232597,
KL Loss: 1005.9547119140625
It 7600: Total Loss: 0.01143758837133646,        Rec Loss: 0.01143758837133646,
KL Loss: 186.90518188476562
```

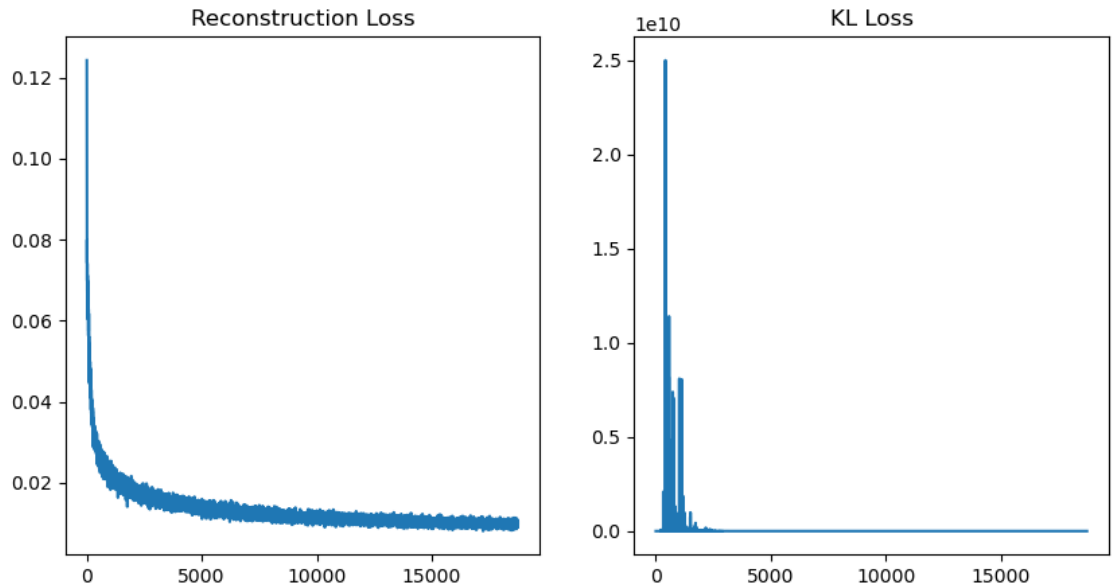
```
It 7700: Total Loss: 0.012700260616838932,       Rec Loss: 0.012700260616838932,
KL Loss: 2462.3388671875
It 7800: Total Loss: 0.010766911320388317,       Rec Loss: 0.010766911320388317,
KL Loss: 624.3251953125
It 7900: Total Loss: 0.012208609841763973,       Rec Loss: 0.012208609841763973,
KL Loss: 1562.8780517578125
It 8000: Total Loss: 0.012603365816175938,       Rec Loss: 0.012603365816175938,
KL Loss: 608.0050659179688
It 8100: Total Loss: 0.012112641707062721,       Rec Loss: 0.012112641707062721,
KL Loss: 747.2120361328125
It 8200: Total Loss: 0.011998135596513748,       Rec Loss: 0.011998135596513748,
KL Loss: 230.74508666992188
It 8300: Total Loss: 0.012499522417783737,       Rec Loss: 0.012499522417783737,
KL Loss: 2779.49462890625
It 8400: Total Loss: 0.01206131000071764,        Rec Loss: 0.01206131000071764,
KL Loss: 243.66465759277344
Run Epoch 9
It 8500: Total Loss: 0.010827641002833843,       Rec Loss: 0.010827641002833843,
KL Loss: 375.22894287109375
It 8600: Total Loss: 0.011680977419018745,       Rec Loss: 0.011680977419018745,
KL Loss: 346.2124328613281
It 8700: Total Loss: 0.010765758343040943,       Rec Loss: 0.010765758343040943,
KL Loss: 474.50640869140625
It 8800: Total Loss: 0.01298467442393303,        Rec Loss: 0.01298467442393303,
KL Loss: 395.0135498046875
It 8900: Total Loss: 0.011049458757042885,       Rec Loss: 0.011049458757042885,
KL Loss: 360.4422607421875
It 9000: Total Loss: 0.012022962793707848,       Rec Loss: 0.012022962793707848,
KL Loss: 381.046630859375
It 9100: Total Loss: 0.011050811968743801,       Rec Loss: 0.011050811968743801,
KL Loss: 496.30181884765625
It 9200: Total Loss: 0.010425316169857979,       Rec Loss: 0.010425316169857979,
KL Loss: 288.41510009765625
It 9300: Total Loss: 0.011710838414728642,       Rec Loss: 0.011710838414728642,
KL Loss: 342.2103271484375
Run Epoch 10
It 9400: Total Loss: 0.010556613095104694,       Rec Loss: 0.010556613095104694,
KL Loss: 450.9995422363281
It 9500: Total Loss: 0.011736113578081131,       Rec Loss: 0.011736113578081131,
KL Loss: 452.42169189453125
It 9600: Total Loss: 0.011320782825350761,       Rec Loss: 0.011320782825350761,
KL Loss: 136.53701782226562
It 9700: Total Loss: 0.011491059325635433,       Rec Loss: 0.011491059325635433,
KL Loss: 231.01333618164062
It 9800: Total Loss: 0.01073453575372696,        Rec Loss: 0.01073453575372696,
KL Loss: 630.264404296875
It 9900: Total Loss: 0.010879047214984894,       Rec Loss: 0.010879047214984894,
KL Loss: 144.68882751464844
```

```
It 10000: Total Loss: 0.011328116059303284,      Rec Loss: 0.011328116059303284,
KL Loss: 1955.496826171875
It 10100: Total Loss: 0.0104948366060853,        Rec Loss: 0.0104948366060853,
KL Loss: 281.75677490234375
It 10200: Total Loss: 0.01072235219180584,       Rec Loss: 0.01072235219180584,
KL Loss: 140.24566650390625
It 10300: Total Loss: 0.012446891516447067,      Rec Loss: 0.012446891516447067,
KL Loss: 134.88771057128906
Run Epoch 11
It 10400: Total Loss: 0.011912881396710873,      Rec Loss: 0.011912881396710873,
KL Loss: 230.6777801513672
It 10500: Total Loss: 0.010623126290738583,      Rec Loss: 0.010623126290738583,
KL Loss: 161.11801147460938
It 10600: Total Loss: 0.011901484802365303,      Rec Loss: 0.011901484802365303,
KL Loss: 145.68038940429688
It 10700: Total Loss: 0.011708037927746773,      Rec Loss: 0.011708037927746773,
KL Loss: 118.82600402832031
It 10800: Total Loss: 0.010651408694684505,      Rec Loss: 0.010651408694684505,
KL Loss: 68.64330291748047
It 10900: Total Loss: 0.011927112936973572,      Rec Loss: 0.011927112936973572,
KL Loss: 327.11578369140625
It 11000: Total Loss: 0.010913311503827572,      Rec Loss: 0.010913311503827572,
KL Loss: 70.44001770019531
It 11100: Total Loss: 0.010130544193089008,      Rec Loss: 0.010130544193089008,
KL Loss: 66.8099594116211
It 11200: Total Loss: 0.010741027072072029,      Rec Loss: 0.010741027072072029,
KL Loss: 233.49871826171875
Run Epoch 12
It 11300: Total Loss: 0.010328875854611397,      Rec Loss: 0.010328875854611397,
KL Loss: 198.10720825195312
It 11400: Total Loss: 0.011023894883692265,      Rec Loss: 0.011023894883692265,
KL Loss: 2006.223876953125
It 11500: Total Loss: 0.010006756521761417,      Rec Loss: 0.010006756521761417,
KL Loss: 60.140865325927734
It 11600: Total Loss: 0.011013210751116276,      Rec Loss: 0.011013210751116276,
KL Loss: 491.5958557128906
It 11700: Total Loss: 0.010638351552188396,      Rec Loss: 0.010638351552188396,
KL Loss: 233.05886840820312
It 11800: Total Loss: 0.01098856795579195,       Rec Loss: 0.01098856795579195,
KL Loss: 73.16096496582031
It 11900: Total Loss: 0.009287908673286438,      Rec Loss: 0.009287908673286438,
KL Loss: 125.66287994384766
It 12000: Total Loss: 0.01027103140950203,       Rec Loss: 0.01027103140950203,
KL Loss: 164.91531372070312
It 12100: Total Loss: 0.01093412283807993,       Rec Loss: 0.01093412283807993,
KL Loss: 84.01512145996094
Run Epoch 13
It 12200: Total Loss: 0.010413842275738716,      Rec Loss: 0.010413842275738716,
```

```
KL Loss: 156.56524658203125
It 12300: Total Loss: 0.010306953452527523,      Rec Loss: 0.010306953452527523,
KL Loss: 68.22283935546875
It 12400: Total Loss: 0.012110740877687931,      Rec Loss: 0.012110740877687931,
KL Loss: 317.0924987792969
It 12500: Total Loss: 0.011338932439684868,      Rec Loss: 0.011338932439684868,
KL Loss: 122.79104614257812
It 12600: Total Loss: 0.011398840695619583,      Rec Loss: 0.011398840695619583,
KL Loss: 172.1075439453125
It 12700: Total Loss: 0.0093923918902874,        Rec Loss: 0.0093923918902874,
KL Loss: 61.984378814697266
It 12800: Total Loss: 0.010967015288770199,      Rec Loss: 0.010967015288770199,
KL Loss: 110.63407135009766
It 12900: Total Loss: 0.01032183039933443,       Rec Loss: 0.01032183039933443,
KL Loss: 194.96311950683594
It 13000: Total Loss: 0.010111618787050247,      Rec Loss: 0.010111618787050247,
KL Loss: 83.77079010009766
It 13100: Total Loss: 0.010159964673221111,      Rec Loss: 0.010159964673221111,
KL Loss: 102.94464874267578
Run Epoch 14
It 13200: Total Loss: 0.010454501025378704,      Rec Loss: 0.010454501025378704,
KL Loss: 87.09127044677734
It 13300: Total Loss: 0.010163204744458199,      Rec Loss: 0.010163204744458199,
KL Loss: 310.96759033203125
It 13400: Total Loss: 0.009632288478314877,      Rec Loss: 0.009632288478314877,
KL Loss: 147.86651611328125
It 13500: Total Loss: 0.011122697032988071,      Rec Loss: 0.011122697032988071,
KL Loss: 75.8782730102539
It 13600: Total Loss: 0.009671667590737343,      Rec Loss: 0.009671667590737343,
KL Loss: 104.76835632324219
It 13700: Total Loss: 0.01037562545388937,       Rec Loss: 0.01037562545388937,
KL Loss: 65.12213897705078
It 13800: Total Loss: 0.008669564500451088,      Rec Loss: 0.008669564500451088,
KL Loss: 247.7252655029297
It 13900: Total Loss: 0.009548378176987171,      Rec Loss: 0.009548378176987171,
KL Loss: 129.85923767089844
It 14000: Total Loss: 0.009887968190014362,      Rec Loss: 0.009887968190014362,
KL Loss: 205.0015106201172
Run Epoch 15
It 14100: Total Loss: 0.010730631649494171,      Rec Loss: 0.010730631649494171,
KL Loss: 65.13755798339844
It 14200: Total Loss: 0.010247055441141129,      Rec Loss: 0.010247055441141129,
KL Loss: 61.31412887573242
It 14300: Total Loss: 0.010920337401330471,      Rec Loss: 0.010920337401330471,
KL Loss: 55.55073547363281
It 14400: Total Loss: 0.011107302270829678,      Rec Loss: 0.011107302270829678,
KL Loss: 67.19448852539062
It 14500: Total Loss: 0.010241374373435974,      Rec Loss: 0.010241374373435974,
```

```
KL Loss: 52.58848190307617
It 14600: Total Loss: 0.010077428072690964,      Rec Loss: 0.010077428072690964,
KL Loss: 45.95368194580078
It 14700: Total Loss: 0.010050860233604908,      Rec Loss: 0.010050860233604908,
KL Loss: 62.337806701660156
It 14800: Total Loss: 0.010510836727917194,      Rec Loss: 0.010510836727917194,
KL Loss: 104.95323181152344
It 14900: Total Loss: 0.010037313215434551,      Rec Loss: 0.010037313215434551,
KL Loss: 55.355106353759766
Run Epoch 16
It 15000: Total Loss: 0.009619276970624924,      Rec Loss: 0.009619276970624924,
KL Loss: 43.997833251953125
It 15100: Total Loss: 0.010673128999769688,      Rec Loss: 0.010673128999769688,
KL Loss: 53.14936065673828
It 15200: Total Loss: 0.011134985834360123,      Rec Loss: 0.011134985834360123,
KL Loss: 43.27973937988281
It 15300: Total Loss: 0.010780302807688713,      Rec Loss: 0.010780302807688713,
KL Loss: 42.96377182006836
It 15400: Total Loss: 0.010975439101457596,      Rec Loss: 0.010975439101457596,
KL Loss: 35.49612808227539
It 15500: Total Loss: 0.009910888969898224,      Rec Loss: 0.009910888969898224,
KL Loss: 82.68000030517578
It 15600: Total Loss: 0.009451749734580517,      Rec Loss: 0.009451749734580517,
KL Loss: 31.29865837097168
It 15700: Total Loss: 0.009434955194592476,      Rec Loss: 0.009434955194592476,
KL Loss: 52.920108795166016
It 15800: Total Loss: 0.009766067378222942,      Rec Loss: 0.009766067378222942,
KL Loss: 50.18920135498047
It 15900: Total Loss: 0.010788513347506523,      Rec Loss: 0.010788513347506523,
KL Loss: 68.78197479248047
Run Epoch 17
It 16000: Total Loss: 0.010096066631376743,      Rec Loss: 0.010096066631376743,
KL Loss: 42.66754913330078
It 16100: Total Loss: 0.010908559896051884,      Rec Loss: 0.010908559896051884,
KL Loss: 140.61428833007812
It 16200: Total Loss: 0.010372579097747803,      Rec Loss: 0.010372579097747803,
KL Loss: 58.57658386230469
It 16300: Total Loss: 0.009599416516721249,      Rec Loss: 0.009599416516721249,
KL Loss: 28.532451629638672
It 16400: Total Loss: 0.009486310184001923,      Rec Loss: 0.009486310184001923,
KL Loss: 78.7850341796875
It 16500: Total Loss: 0.01034475676715374,       Rec Loss: 0.01034475676715374,
KL Loss: 60.41432571411133
It 16600: Total Loss: 0.010204600170254707,      Rec Loss: 0.010204600170254707,
KL Loss: 62.660926818847656
It 16700: Total Loss: 0.009922019205987453,      Rec Loss: 0.009922019205987453,
KL Loss: 19.85037612915039
It 16800: Total Loss: 0.010200952179729939,      Rec Loss: 0.010200952179729939,
```

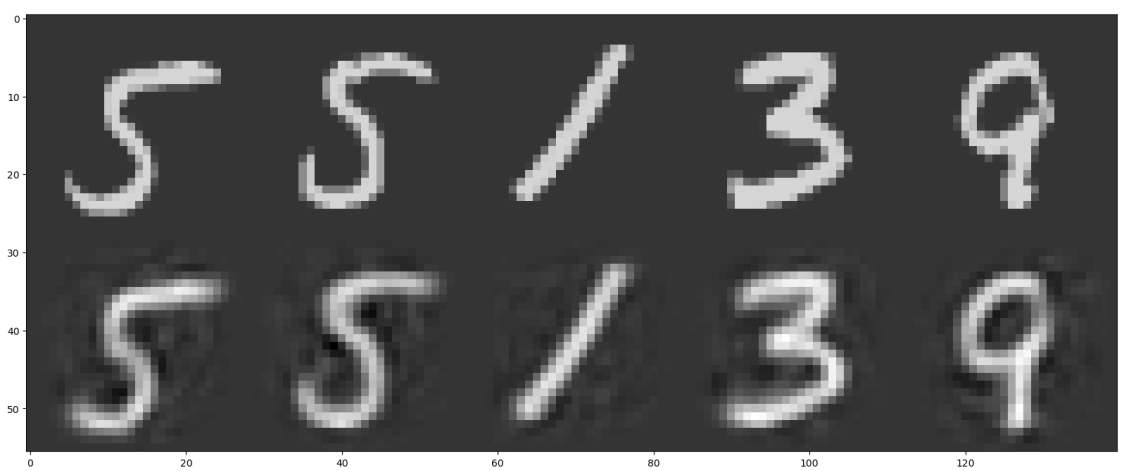
```
KL Loss: 60.67787551879883
Run Epoch 18
It 16900: Total Loss: 0.009655089117586613,      Rec Loss: 0.009655089117586613,
KL Loss: 41.48760223388672
It 17000: Total Loss: 0.009156799875199795,      Rec Loss: 0.009156799875199795,
KL Loss: 32.58353042602539
It 17100: Total Loss: 0.010141544044017792,      Rec Loss: 0.010141544044017792,
KL Loss: 196.78504943847656
It 17200: Total Loss: 0.010063869878649712,      Rec Loss: 0.010063869878649712,
KL Loss: 65.52796936035156
It 17300: Total Loss: 0.010100796818733215,      Rec Loss: 0.010100796818733215,
KL Loss: 31.571537017822266
It 17400: Total Loss: 0.010563714429736137,      Rec Loss: 0.010563714429736137,
KL Loss: 29.942523956298828
It 17500: Total Loss: 0.009741577319800854,      Rec Loss: 0.009741577319800854,
KL Loss: 54.23160171508789
It 17600: Total Loss: 0.010694781318306923,      Rec Loss: 0.010694781318306923,
KL Loss: 29.848377227783203
It 17700: Total Loss: 0.009081815369427204,      Rec Loss: 0.009081815369427204,
KL Loss: 34.175872802734375
It 17800: Total Loss: 0.009718145243823528,      Rec Loss: 0.009718145243823528,
KL Loss: 26.585445404052734
Run Epoch 19
It 17900: Total Loss: 0.008789455518126488,      Rec Loss: 0.008789455518126488,
KL Loss: 44.54269027709961
It 18000: Total Loss: 0.009797447361052036,      Rec Loss: 0.009797447361052036,
KL Loss: 27.510513305664062
It 18100: Total Loss: 0.00962933897972107,       Rec Loss: 0.00962933897972107,
KL Loss: 29.057113647460938
It 18200: Total Loss: 0.009163320995867252,      Rec Loss: 0.009163320995867252,
KL Loss: 34.0850944519043
It 18300: Total Loss: 0.01011430099606514,       Rec Loss: 0.01011430099606514,
KL Loss: 54.34074020385742
It 18400: Total Loss: 0.008692123927175999,      Rec Loss: 0.008692123927175999,
KL Loss: 12.67809009552002
It 18500: Total Loss: 0.01001347042620182,       Rec Loss: 0.01001347042620182,
KL Loss: 37.933692932128906
It 18600: Total Loss: 0.009504429064691067,      Rec Loss: 0.009504429064691067,
KL Loss: 86.50745391845703
It 18700: Total Loss: 0.010487492196261883,      Rec Loss: 0.010487492196261883,
KL Loss: 16.639896392822266
Done!
```



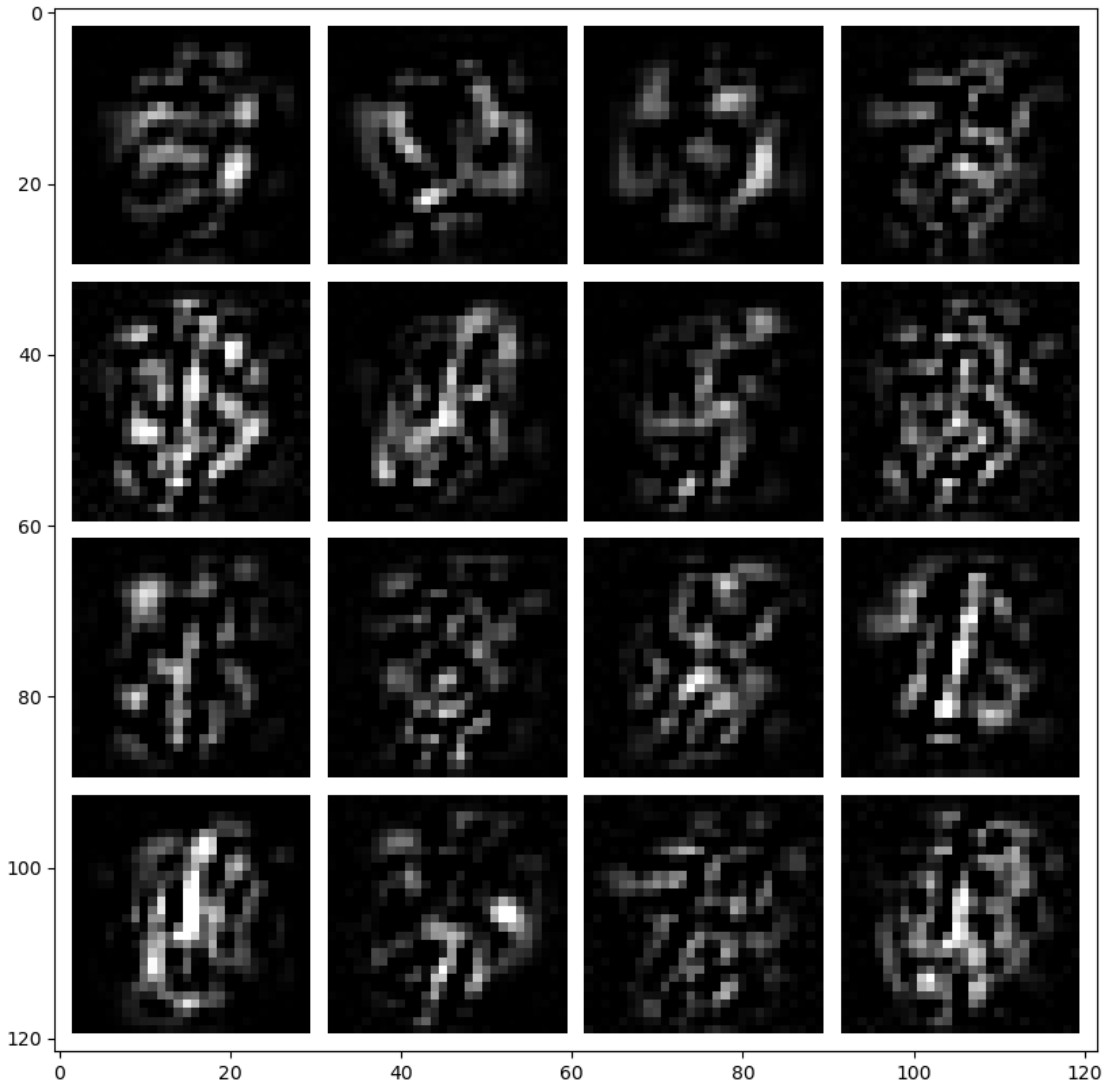
Let's look at some reconstructions and decoded embedding samples!

[12]:
```
# visualize VAE reconstructions and samples from the generative model
print("beta = ", beta)
vis_reconstruction(vae_model, randomize=True)
vis_samples(vae_model)
```

```
beta =  0
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



4.3 Tweaking the loss function β [2pt]

Prob1-6: Let's repeat the same experiment for $\beta = 10$, a very high value for the coefficient.

[13]:
```
# VAE training loop
learning_rate = 1e-3
nz = 32
beta = 10

######################################## TODO␣
↪########################################
epochs = 15       # recommended 5-20 epochs
```

```
#################################### END TODO␣
↪#####################################

# build VAE model
vae_model = VAE(nz, beta).to(device)    # transfer model to GPU if available
vae_model = vae_model.train()   # set model in train mode (eg batchnorm params␣
↪get updated)

# build optimizer and loss function
##################################### TODO␣
↪######################################
# Build the optimizer for the vae_model. We will again use the Adam optimizer␣
↪with #
# the given learning rate and otherwise default parameters.                  ␣
↪    #
#################################################################################
# same as AE
from torch.optim import Adam
optimizer = Adam(vae_model.parameters(), lr=learning_rate)
#################################### END TODO␣
↪#####################################

train_it = 0
rec_loss, kl_loss = [], []
print(f"Running {epochs} epochs with {beta=}")
for ep in range(epochs):
  print("Run Epoch {}".format(ep))
  ##################################### TODO␣
↪######################################
  # Implement the main training loop for the VAE model.                       ␣
↪     #
  # HINT: Your training loop should sample batches from the data loader, run␣
↪the     #
  #       forward pass of the VAE, compute the loss, perform the backward pass␣
↪and   #
  #       perform one gradient step with the optimizer.                       ␣
↪     #
  # HINT: Don't forget to erase old gradients before performing the backward␣
↪pass.   #
  # HINT: This time we will use the loss() function of our model for computing␣
↪the   #
  #       training loss. It outputs the total training loss and a dict␣
↪containing    #
  #       the breakdown of reconstruction and KL loss.                        ␣
↪     #
```

```
␣
↪##################################################################################
 for indx, (img, _) in enumerate(mnist_data_loader): # index and img,label␣
↪tuple
    optimizer.zero_grad()
    batch_features = torch.reshape(img, (-1, 784) ).to(device)
    vae_output = vae_model(batch_features)
    total_loss, losses = vae_model.loss(batch_features, outputs=vae_output)
    rec_loss.append(losses['rec_loss']); kl_loss.append(losses['kl_loss'])
    total_loss.backward()
    optimizer.step()

    if train_it % 100 == 0:
      print("It {}: Total Loss: {}, \t Rec Loss: {},\t KL Loss: {}"\
            .format(train_it, total_loss, losses['rec_loss'],␣
↪losses['kl_loss']))
    train_it += 1
 ##################################### END TODO␣
↪#####################################

print("Done!")

rec_loss_plotdata = [foo.detach().cpu() for foo in rec_loss]
kl_loss_plotdata = [foo.detach().cpu() for foo in kl_loss]

# log the loss training curves
fig = plt.figure(figsize = (10, 5))
ax1 = plt.subplot(121)
ax1.plot(rec_loss_plotdata)
ax1.title.set_text("Reconstruction Loss")
ax2 = plt.subplot(122)
ax2.plot(kl_loss_plotdata)
ax2.title.set_text("KL Loss")
plt.show()
```

```
Running 15 epochs with beta=10
Run Epoch 0
It 0: Total Loss: 40.692604064941406,    Rec Loss: 0.12546515464782715,  KL
Loss: 4.056714057922363
It 100: Total Loss: 0.11989874392747879,         Rec Loss: 0.06820663064718246,
KL Loss: 0.00516921142116189
It 200: Total Loss: 0.08034989982843399,         Rec Loss: 0.06451409310102463,
KL Loss: 0.0015835806261748075
It 300: Total Loss: 0.07434115558862686,         Rec Loss: 0.0588078498840332,
KL Loss: 0.0015533303376287222
It 400: Total Loss: 0.07327618449926376,         Rec Loss: 0.05913497135043144,
KL Loss: 0.0014141214778646827
```

```
It 500: Total Loss: 0.06681125611066818,        Rec Loss: 0.058871444314718246,
KL Loss: 0.0007939814240671694
It 600: Total Loss: 0.06202341988682747,        Rec Loss: 0.05337893217802048,
KL Loss: 0.0008644488407298923
It 700: Total Loss: 0.06096790358424187,        Rec Loss: 0.0539504736661911,
KL Loss: 0.0007017430616542697
It 800: Total Loss: 0.05927952751517296,        Rec Loss: 0.05196771025657654,
KL Loss: 0.0007311817607842386
It 900: Total Loss: 0.05788804590702057,        Rec Loss: 0.05209960788488388,
KL Loss: 0.0005788437556475401
Run Epoch 1
It 1000: Total Loss: 0.0554080568253994,        Rec Loss: 0.04956437274813652,
KL Loss: 0.0005843684775754809
It 1100: Total Loss: 0.06153205782175064,       Rec Loss: 0.05230631306767464,
KL Loss: 0.0009225746616721153
It 1200: Total Loss: 0.0723884254693985,        Rec Loss: 0.05915689468383789,
KL Loss: 0.0013231527991592884
It 1300: Total Loss: 0.05178862065076828,       Rec Loss: 0.04559628665447235,
KL Loss: 0.0006192334112711251
It 1400: Total Loss: 0.054462019354104996,      Rec Loss: 0.045153528451919556,
KL Loss: 0.0009308489970862865
It 1500: Total Loss: 0.052752330899238586,      Rec Loss: 0.04540740326046944,
KL Loss: 0.0007344929035753012
It 1600: Total Loss: 0.0487823449075222,        Rec Loss: 0.04286495968699455,
KL Loss: 0.0005917383823543787
It 1700: Total Loss: 0.053556784987449646,      Rec Loss: 0.045418914407491684,
KL Loss: 0.0008137869299389422
It 1800: Total Loss: 0.054872240871191025,      Rec Loss: 0.047628939151763916,
KL Loss: 0.0007243301370181143
Run Epoch 2
It 1900: Total Loss: 0.06344566494226456,       Rec Loss: 0.05487685278058052,
KL Loss: 0.0008568809134885669
It 2000: Total Loss: 0.047035593539476395,      Rec Loss: 0.04121028259396553,
KL Loss: 0.0005825311527587473
It 2100: Total Loss: 0.04514414444565773,       Rec Loss: 0.039664965122938156,
KL Loss: 0.0005479178507812321
It 2200: Total Loss: 0.04465391859412193,       Rec Loss: 0.03920718654990196,
KL Loss: 0.0005446730647236109
It 2300: Total Loss: 0.041403863579034805,      Rec Loss: 0.0360909029841423,
KL Loss: 0.0005312961293384433
It 2400: Total Loss: 0.0480203703045845,        Rec Loss: 0.03986736759543419,
KL Loss: 0.0008153000962920487
It 2500: Total Loss: 0.04585277661681175,       Rec Loss: 0.04036518931388855,
KL Loss: 0.0005487586022354662
It 2600: Total Loss: 0.04411100968718529,       Rec Loss: 0.03721568360924721,
KL Loss: 0.0006895326660014689
It 2700: Total Loss: 0.04331619292497635,       Rec Loss: 0.03607232868671417,
KL Loss: 0.0007243863656185567
```

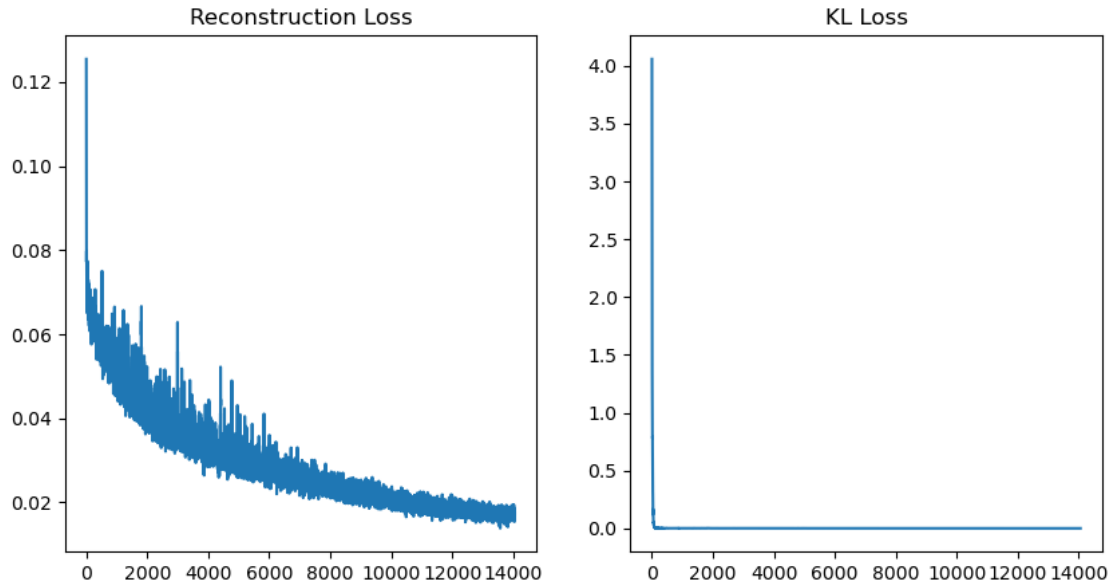
```
It 2800: Total Loss: 0.04827246814966202,        Rec Loss: 0.042035169899463654,
KL Loss: 0.0006237298366613686
Run Epoch 3
It 2900: Total Loss: 0.04553946107625961,        Rec Loss: 0.03880332410335541,
KL Loss: 0.0006736135692335665
It 3000: Total Loss: 0.05212422460317612,        Rec Loss: 0.04345635324716568,
KL Loss: 0.0008667873335070908
It 3100: Total Loss: 0.038541458547115326,       Rec Loss: 0.03376023843884468,
KL Loss: 0.0004781220341101289
It 3200: Total Loss: 0.03979228064417839,        Rec Loss: 0.035083748400211334,
KL Loss: 0.00047085335245355964
It 3300: Total Loss: 0.039367400109767914,       Rec Loss: 0.03337535262107849,
KL Loss: 0.0005992045626044273
It 3400: Total Loss: 0.047209709882736206,       Rec Loss: 0.04021235927939415,
KL Loss: 0.0006997348973527551
It 3500: Total Loss: 0.037588346749544144,       Rec Loss: 0.03229314088821411,
KL Loss: 0.0005295206792652607
It 3600: Total Loss: 0.037897203117609024,       Rec Loss: 0.033532194793224335,
KL Loss: 0.00043650076258927584
It 3700: Total Loss: 0.04104303941130638,        Rec Loss: 0.035323020070791245,
KL Loss: 0.0005720017943531275
Run Epoch 4
It 3800: Total Loss: 0.042215876281261444,       Rec Loss: 0.03679080680012703,
KL Loss: 0.000542506983038038
It 3900: Total Loss: 0.041346143931150436,       Rec Loss: 0.03559679538011551,
KL Loss: 0.0005749347619712353
It 4000: Total Loss: 0.04157992824912071,        Rec Loss: 0.036013226956129074,
KL Loss: 0.0005566701292991638
It 4100: Total Loss: 0.04371052607893944,        Rec Loss: 0.03825009986758232,
KL Loss: 0.0005460427491925657
It 4200: Total Loss: 0.03733864799141884,        Rec Loss: 0.03224525228142738,
KL Loss: 0.0005093394429422915
It 4300: Total Loss: 0.0354495570063591,         Rec Loss: 0.031237095594406128,
KL Loss: 0.0004212461062707007
It 4400: Total Loss: 0.04742869362235069,        Rec Loss: 0.04122641682624817,
KL Loss: 0.000620227656327188
It 4500: Total Loss: 0.037319134920835495,       Rec Loss: 0.03263605758547783,
KL Loss: 0.00046830775681883097
It 4600: Total Loss: 0.03558066487312317,        Rec Loss: 0.030918866395950317,
KL Loss: 0.00046617991756647825
Run Epoch 5
It 4700: Total Loss: 0.031834907829761505,       Rec Loss: 0.027796506881713867,
KL Loss: 0.00040383997838944197
It 4800: Total Loss: 0.03536798804998398,        Rec Loss: 0.031072456389665604,
KL Loss: 0.0004295532708056271
It 4900: Total Loss: 0.036754533648490906,       Rec Loss: 0.0312197282910347,
KL Loss: 0.0005534803494811058
It 5000: Total Loss: 0.03035397082567215,        Rec Loss: 0.02670816332101822,
```

```
KL Loss: 0.0003645807155407965
It 5100: Total Loss: 0.035819608718156815,       Rec Loss: 0.03160678595304489,
KL Loss: 0.00042128219502046704
It 5200: Total Loss: 0.03870804235339165,        Rec Loss: 0.03379615396261215,
KL Loss: 0.0004911889554932714
It 5300: Total Loss: 0.034724161028862,          Rec Loss: 0.030658571049571037,
KL Loss: 0.0004065590910613537
It 5400: Total Loss: 0.0310883317142725,         Rec Loss: 0.02714347280561924,
KL Loss: 0.00039448594907298684
It 5500: Total Loss: 0.032965097576379776,       Rec Loss: 0.029236722737550735,
KL Loss: 0.0003728375304490328
It 5600: Total Loss: 0.029132958501577377,       Rec Loss: 0.02552364394068718,
KL Loss: 0.0003609313862398267
Run Epoch 6
It 5700: Total Loss: 0.030104774981737137,       Rec Loss: 0.02655075490474701,
KL Loss: 0.00035540194949135184
It 5800: Total Loss: 0.03253660351037979,        Rec Loss: 0.029017183929681778,
KL Loss: 0.0003519419697113335
It 5900: Total Loss: 0.030785655602812767,       Rec Loss: 0.02720971405506134,
KL Loss: 0.00035759410820901394
It 6000: Total Loss: 0.0325789712369442,         Rec Loss: 0.029146360233426094,
KL Loss: 0.0003432610537856817
It 6100: Total Loss: 0.027857046574354172,       Rec Loss: 0.024886924773454666,
KL Loss: 0.0002970121568068862
It 6200: Total Loss: 0.03501202538609505,        Rec Loss: 0.03143827244639397,
KL Loss: 0.00035737536381930113
It 6300: Total Loss: 0.030355675145983696,       Rec Loss: 0.02680407278239727,
KL Loss: 0.0003551602712832391
It 6400: Total Loss: 0.0323035791516304,         Rec Loss: 0.028854943811893463,
KL Loss: 0.0003448637144174427
It 6500: Total Loss: 0.030921243131160736,       Rec Loss: 0.027793990448117256,
KL Loss: 0.0003127253148704767
Run Epoch 7
It 6600: Total Loss: 0.028782187029719353,       Rec Loss: 0.025762241333723068,
KL Loss: 0.0003019946161657572
It 6700: Total Loss: 0.0337042361497879,         Rec Loss: 0.03035564534366131,
KL Loss: 0.00033485924359411
It 6800: Total Loss: 0.028985917568206787,       Rec Loss: 0.025912629440426826,
KL Loss: 0.00030732882441952825
It 6900: Total Loss: 0.030573446303606033,       Rec Loss: 0.027597183361649513,
KL Loss: 0.0002976263058371842
It 7000: Total Loss: 0.029569003731012344,       Rec Loss: 0.026415416970849037,
KL Loss: 0.000315358629450202
It 7100: Total Loss: 0.0262245312333107,         Rec Loss: 0.02339661307632923,
KL Loss: 0.0002827918797265738
It 7200: Total Loss: 0.027385903522372246,       Rec Loss: 0.024365244433283806,
KL Loss: 0.0003020659787580371
It 7300: Total Loss: 0.02801918238401413,        Rec Loss: 0.025189997628331184,
```

```
KL Loss: 0.0002829185104928911
It 7400: Total Loss: 0.028023062273859978,       Rec Loss: 0.024914322420954704,
KL Loss: 0.00031087404931895435
Run Epoch 8
It 7500: Total Loss: 0.026964738965034485,       Rec Loss: 0.024071762338280678,
KL Loss: 0.0002892976626753807
It 7600: Total Loss: 0.027063021436333656,       Rec Loss: 0.02422831580042839,
KL Loss: 0.0002834705519489944
It 7700: Total Loss: 0.02905024215579033,        Rec Loss: 0.02631594054400921,
KL Loss: 0.0002734301669988781
It 7800: Total Loss: 0.027198834344744682,       Rec Loss: 0.02465052902698517,
KL Loss: 0.0002548304619267583
It 7900: Total Loss: 0.026985257863998413,       Rec Loss: 0.02424566075205803,
KL Loss: 0.00027395979850552976
It 8000: Total Loss: 0.026344021782279015,       Rec Loss: 0.02365303784608841,
KL Loss: 0.00026909844018518925
It 8100: Total Loss: 0.02737419120967388,        Rec Loss: 0.024616805836558342,
KL Loss: 0.00027573859551921487
It 8200: Total Loss: 0.02510819397866726,        Rec Loss: 0.02255084551870823,
KL Loss: 0.00025573489256203175
It 8300: Total Loss: 0.025300711393356323,       Rec Loss: 0.02281453087925911,
KL Loss: 0.0002486179582774639
It 8400: Total Loss: 0.02436136268079281,        Rec Loss: 0.021957898512482643,
KL Loss: 0.00024034641683101654
Run Epoch 9
It 8500: Total Loss: 0.02218724973499775,        Rec Loss: 0.01993088610470295,
KL Loss: 0.0002256363513879478
It 8600: Total Loss: 0.025793859735131264,       Rec Loss: 0.023327482864260674,
KL Loss: 0.00024663773365318775
It 8700: Total Loss: 0.023651087656617165,       Rec Loss: 0.021407585591077805,
KL Loss: 0.0002243501367047429
It 8800: Total Loss: 0.025422165170311928,       Rec Loss: 0.022998707368969917,
KL Loss: 0.00024234573356807232
It 8900: Total Loss: 0.02378649078309536,        Rec Loss: 0.021633639931678772,
KL Loss: 0.00021528505021706223
It 9000: Total Loss: 0.021978601813316345,       Rec Loss: 0.019707726314663887,
KL Loss: 0.00022708764299750328
It 9100: Total Loss: 0.0243410412222147,         Rec Loss: 0.02219739742577076,
KL Loss: 0.00021436443785205483
It 9200: Total Loss: 0.024333249777555466,       Rec Loss: 0.022075230255723,
KL Loss: 0.00022580201039090753
It 9300: Total Loss: 0.02490476332604885,        Rec Loss: 0.022730320692062378,
KL Loss: 0.0002174442051909864
Run Epoch 10
It 9400: Total Loss: 0.021706262603402138,       Rec Loss: 0.01957760378718376,
KL Loss: 0.0002128658234141767
It 9500: Total Loss: 0.024160834029316902,       Rec Loss: 0.02196037769317627,
KL Loss: 0.00022004562197253108
```

```
It 9600: Total Loss: 0.02527516894042492,        Rec Loss: 0.02306233160197735,
KL Loss: 0.00022128375712782145
It 9700: Total Loss: 0.024629872292280197,       Rec Loss: 0.022388532757759094,
KL Loss: 0.0002241340116597712
It 9800: Total Loss: 0.021428100764751434,       Rec Loss: 0.019497742876410484,
KL Loss: 0.00019303569570183754
It 9900: Total Loss: 0.0213693268597126,         Rec Loss: 0.019489916041493416,
KL Loss: 0.00018794112838804722
It 10000: Total Loss: 0.02351035177707672,       Rec Loss: 0.021534180268645287,
KL Loss: 0.00019761704606935382
It 10100: Total Loss: 0.02275552973151207,       Rec Loss: 0.02084130235016346,
KL Loss: 0.00019142270321026444
It 10200: Total Loss: 0.022323554381728172,      Rec Loss: 0.0203376617282629,
KL Loss: 0.00019858923042193055
It 10300: Total Loss: 0.020934319123625755,      Rec Loss: 0.019096190109848976,
KL Loss: 0.00018381295376457274
Run Epoch 11
It 10400: Total Loss: 0.020296869799494743,      Rec Loss: 0.018327990546822548,
KL Loss: 0.00019688793690875173
It 10500: Total Loss: 0.020897069945931435,      Rec Loss: 0.01906953938305378,
KL Loss: 0.0001827530504669994
It 10600: Total Loss: 0.021783463656902313,      Rec Loss: 0.019984113052487373,
KL Loss: 0.00017993513029068708
It 10700: Total Loss: 0.021017540246248245,      Rec Loss: 0.019213948398828506,
KL Loss: 0.0001803591731004417
It 10800: Total Loss: 0.022781090810894966,      Rec Loss: 0.020849674940109253,
KL Loss: 0.00019314163364470005
It 10900: Total Loss: 0.020681317895650864,      Rec Loss: 0.01887303777039051,
KL Loss: 0.00018082797760143876
It 11000: Total Loss: 0.02083142101764679,       Rec Loss: 0.018989210948348045,
KL Loss: 0.0001842209603637457
It 11100: Total Loss: 0.02093757688999176,       Rec Loss: 0.019210081547498703,
KL Loss: 0.0001727496273815632
It 11200: Total Loss: 0.019622907042503357,      Rec Loss: 0.01796911470592022,
KL Loss: 0.0001653791987337172
Run Epoch 12
It 11300: Total Loss: 0.01963057927787304,       Rec Loss: 0.0179604385048151,
KL Loss: 0.00016701407730579376
It 11400: Total Loss: 0.0206094179302454,        Rec Loss: 0.01890820823609829,
KL Loss: 0.00017012091120705009
It 11500: Total Loss: 0.020315736532211304,      Rec Loss: 0.018629401922225952,
KL Loss: 0.00016863341443240643
It 11600: Total Loss: 0.02025231160223484,       Rec Loss: 0.01850486360490322,
KL Loss: 0.0001747448113746941
It 11700: Total Loss: 0.020048614591360092,      Rec Loss: 0.018478570505976677,
KL Loss: 0.00015700439689680934
It 11800: Total Loss: 0.020240789279341698,      Rec Loss: 0.01865086704492569,
KL Loss: 0.00015899224672466516
```

```
It 11900: Total Loss: 0.022755691781640053,      Rec Loss: 0.02116403542459011,
KL Loss: 0.00015916564734652638
It 12000: Total Loss: 0.02330436371266842,       Rec Loss: 0.021661167964339256,
KL Loss: 0.00016431952826678753
It 12100: Total Loss: 0.021229134872555733,      Rec Loss: 0.019692962989211082,
KL Loss: 0.00015361717669293284
Run Epoch 13
It 12200: Total Loss: 0.019799115136265755,      Rec Loss: 0.018379367887973785,
KL Loss: 0.00014197471318766475
It 12300: Total Loss: 0.01946120522916317,       Rec Loss: 0.017887897789478302,
KL Loss: 0.00015733076725155115
It 12400: Total Loss: 0.018896887078881264,      Rec Loss: 0.01742432825267315,
KL Loss: 0.00014725583605468273
It 12500: Total Loss: 0.019660603255033493,      Rec Loss: 0.018168481066823006,
KL Loss: 0.00014921213733032346
It 12600: Total Loss: 0.019328230991959572,      Rec Loss: 0.017889967188239098,
KL Loss: 0.00014382641529664397
It 12700: Total Loss: 0.019024699926376343,      Rec Loss: 0.01766810193657875,
KL Loss: 0.0001356597349513322
It 12800: Total Loss: 0.017874017357826233,      Rec Loss: 0.016567975282669067,
KL Loss: 0.00013060425408184528
It 12900: Total Loss: 0.017709339037537575,      Rec Loss: 0.016373269259929657,
KL Loss: 0.00013360701268538833
It 13000: Total Loss: 0.021704671904444695,      Rec Loss: 0.020241795107722282,
KL Loss: 0.00014628772623836994
It 13100: Total Loss: 0.01939859427511692,       Rec Loss: 0.018064886331558228,
KL Loss: 0.0001333708642050624
Run Epoch 14
It 13200: Total Loss: 0.01804516278207302,       Rec Loss: 0.016687970608472824,
KL Loss: 0.0001357191358692944
It 13300: Total Loss: 0.019252749159932137,      Rec Loss: 0.017907246947288513,
KL Loss: 0.00013455015141516924
It 13400: Total Loss: 0.018871000036597252,      Rec Loss: 0.017481127753853798,
KL Loss: 0.00013898717588745058
It 13500: Total Loss: 0.01982622593641281,       Rec Loss: 0.018533840775489807,
KL Loss: 0.0001292385859414935
It 13600: Total Loss: 0.019231662154197693,      Rec Loss: 0.017938774079084396,
KL Loss: 0.00012928881915286183
It 13700: Total Loss: 0.019016481935977936,      Rec Loss: 0.017754720523953438,
KL Loss: 0.00012617604807019234
It 13800: Total Loss: 0.018522940576076508,      Rec Loss: 0.01719343662261963,
KL Loss: 0.00013295048847794533
It 13900: Total Loss: 0.01738261617720127,       Rec Loss: 0.016105594113469124,
KL Loss: 0.00012770213652402163
It 14000: Total Loss: 0.0170329287648201,        Rec Loss: 0.015801358968019485,
KL Loss: 0.0001231570786330849
Done!
```



Inline Question: What can you observe when setting $\beta = 0$ and $\beta = 10$? Explain your observations! [2pt] (max 200 words)

Answer: That beta values are extremely important to guage the loss correctly. Beta = 0 complemetly ignores the KL loss but a value as high as 10 relatively ignores how well the model is able to reconstruct the image.

4.4 Obtaining the best β-factor [5pt]

Prob 1-6 continued: Now we can start tuning the beta value to achieve a good result. First describe what a "good result" would look like (focus what you would expect for reconstructions and sample quality).

Inline Question: Characterize what properties you would expect for reconstructions and samples of a well-tuned VAE! [3pt] (max 200 words)

Answer: It should be able to denoise images once reconstructed while still keeping the intergrity of the image both semantically and clarity wise.

Now that you know what outcome we would like to obtain, try to tune β to achieve this result. Logarithmic search in steps of 10x will be helpful, good results can be achieved after ~20 epochs of training. Training reconstructions should be high quality, test samples should be diverse, distinguishable numbers, most samples recognizable as numbers.

Answer: Tuned beta value 0.1_____ [2pt]

[20]:
```
# Tuning for best beta
learning_rate = 1e-3
nz = 32
```

```
######################################## TODO␣
↪#######################################
epochs = 20     # recommended 5-20 epochs
beta = 0.1 # Tune this for best results
##################################### END TODO␣
↪#######################################

# build VAE model
vae_model = VAE(nz, beta).to(device)    # transfer model to GPU if available
vae_model = vae_model.train()   # set model in train mode (eg batchnorm params␣
↪get updated)

# build optimizer and loss function
######################################## TODO␣
↪#######################################
# Build the optimizer for the vae_model. We will again use the Adam optimizer␣
↪with #
# the given learning rate and otherwise default parameters.                  ␣
↪    #
##################################################################################
# same as AE
from torch.optim import Adam
optimizer = Adam(vae_model.parameters(), lr=learning_rate)
##################################### END TODO␣
↪#######################################

train_it = 0
rec_loss, kl_loss = [], []
print(f"Running {epochs} epochs with {beta=}")
for ep in range(epochs):
  print("Run Epoch {}".format(ep))
  ######################################## TODO␣
↪#######################################
  # Implement the main training loop for the VAE model.                       ␣
↪     #
  # HINT: Your training loop should sample batches from the data loader, run␣
↪the     #
  #       forward pass of the VAE, compute the loss, perform the backward pass␣
↪and   #
  #       perform one gradient step with the optimizer.                       ␣
↪     #
  # HINT: Don't forget to erase old gradients before performing the backward␣
↪pass.   #
  # HINT: This time we will use the loss() function of our model for computing␣
↪the   #
```

```
 #       training loss. It outputs the total training loss and a dict␣
↪containing    #
 #       the breakdown of reconstruction and KL loss.                  ␣
↪     #
␣
↪##################################################################################
 for indx, (img, _) in enumerate(mnist_data_loader): # index and img,label␣
↪tuple
   optimizer.zero_grad()
   batch_features = torch.reshape(img, (-1, 784) ).to(device)
   vae_output = vae_model(batch_features)
   total_loss, losses = vae_model.loss(batch_features, outputs=vae_output)
   rec_loss.append(losses['rec_loss']); kl_loss.append(losses['kl_loss'])
   total_loss.backward()
   optimizer.step()

   if train_it % 100 == 0:
     print("It {}: Total Loss: {}, \t Rec Loss: {},\t KL Loss: {}"\
           .format(train_it, total_loss, losses['rec_loss'],␣
↪losses['kl_loss']))
   train_it += 1
 ##################################### END TODO␣
↪#####################################

print("Done!")

rec_loss_plotdata = [foo.detach().cpu() for foo in rec_loss]
kl_loss_plotdata = [foo.detach().cpu() for foo in kl_loss]

# log the loss training curves
fig = plt.figure(figsize = (10, 5))
ax1 = plt.subplot(121)
ax1.plot(rec_loss_plotdata)
ax1.title.set_text("Reconstruction Loss")
ax2 = plt.subplot(122)
ax2.plot(kl_loss_plotdata)
ax2.title.set_text("KL Loss")
plt.show()
```

```
Running 20 epochs with beta=0.1
Run Epoch 0
It 0: Total Loss: 0.49524736404418945,   Rec Loss: 0.13641993701457977,  KL
Loss: 3.5882742404937744
It 100: Total Loss: 0.06618674844503403,         Rec Loss: 0.06493497639894485,
KL Loss: 0.012517754919826984
It 200: Total Loss: 0.05338617414236069,         Rec Loss: 0.05222732201218605,
KL Loss: 0.011588537134230137
```

```
It 300: Total Loss: 0.04674878343939781,        Rec Loss: 0.045398712158203125,
KL Loss: 0.013500720262527466
It 400: Total Loss: 0.0386401042342186,         Rec Loss: 0.037288907915353775,
KL Loss: 0.013511947356164455
It 500: Total Loss: 0.03938858211040497,        Rec Loss: 0.03775431215763092,
KL Loss: 0.016342684626579285
It 600: Total Loss: 0.03747275844216347,        Rec Loss: 0.03605631738901138,
KL Loss: 0.014164423570036888
It 700: Total Loss: 0.03321193531155586,        Rec Loss: 0.03179987147450447,
KL Loss: 0.014120631851255894
It 800: Total Loss: 0.03300557658076286,        Rec Loss: 0.031521521508693695,
KL Loss: 0.014840558171272278
It 900: Total Loss: 0.028282035142183304,       Rec Loss: 0.026978032663464546,
KL Loss: 0.013040032237768173
Run Epoch 1
It 1000: Total Loss: 0.027364972978830338,      Rec Loss: 0.026172839105129242,
KL Loss: 0.011921340599656105
It 1100: Total Loss: 0.028370702639222145,      Rec Loss: 0.027121631428599358,
KL Loss: 0.012490707449615002
It 1200: Total Loss: 0.026845959946513176,      Rec Loss: 0.02569751814007759,
KL Loss: 0.011484420858323574
It 1300: Total Loss: 0.026572857052087784,      Rec Loss: 0.02540932595729828,
KL Loss: 0.011635318398475647
It 1400: Total Loss: 0.02287272736430168,       Rec Loss: 0.021617820486426353,
KL Loss: 0.012549066916108131
It 1500: Total Loss: 0.02420825883746147,       Rec Loss: 0.023087739944458008,
KL Loss: 0.011205179616808891
It 1600: Total Loss: 0.02398248016834259,       Rec Loss: 0.02295989915728569,
KL Loss: 0.010225816629827023
It 1700: Total Loss: 0.0242948979139328,        Rec Loss: 0.023123493418097496,
KL Loss: 0.011714035645127296
It 1800: Total Loss: 0.02163090929389,   Rec Loss: 0.020584583282470703,
KL Loss: 0.010463252663612366
Run Epoch 2
It 1900: Total Loss: 0.020809972658753395,      Rec Loss: 0.01982540264725685,
KL Loss: 0.009845704771578312
It 2000: Total Loss: 0.0234064981341362,        Rec Loss: 0.022415349259972572,
KL Loss: 0.009911494329571724
It 2100: Total Loss: 0.020454153418540955,      Rec Loss: 0.01948183961212635,
KL Loss: 0.009723147377371788
It 2200: Total Loss: 0.021938355639576912,      Rec Loss: 0.02098899520933628,
KL Loss: 0.009493599645793438
It 2300: Total Loss: 0.021989023312926292,      Rec Loss: 0.021108267828822136,
KL Loss: 0.008807560428977013
It 2400: Total Loss: 0.02025890164077282,       Rec Loss: 0.019344711676239967,
KL Loss: 0.009141901507973671
It 2500: Total Loss: 0.021737942472100258,      Rec Loss: 0.02090523950755596,
KL Loss: 0.008327038027346134
```

```
It 2600: Total Loss: 0.020618993788957596,       Rec Loss: 0.0197466854006052,
KL Loss: 0.008723076432943344
It 2700: Total Loss: 0.01792244054377079,        Rec Loss: 0.0171295665204525,
KL Loss: 0.007928740233182907
It 2800: Total Loss: 0.018989354372024536,       Rec Loss: 0.018160520121455193,
KL Loss: 0.008288335986435413
Run Epoch 3
It 2900: Total Loss: 0.01975048892199993,        Rec Loss: 0.019003158435225487,
KL Loss: 0.007473296485841274
It 3000: Total Loss: 0.019361786544322968,       Rec Loss: 0.01852557063102722,
KL Loss: 0.00836215540766716
It 3100: Total Loss: 0.01633233204483986,        Rec Loss: 0.01559493038803339,
KL Loss: 0.007374025881290436
It 3200: Total Loss: 0.018877839669585228,       Rec Loss: 0.018083758652210236,
KL Loss: 0.007940814830362797
It 3300: Total Loss: 0.01778615079820156,        Rec Loss: 0.017076769843697548,
KL Loss: 0.0070938123390078545
It 3400: Total Loss: 0.01862029917538166,        Rec Loss: 0.017817828804254532,
KL Loss: 0.008024708367884159
It 3500: Total Loss: 0.01739366352558136,        Rec Loss: 0.01664603501558304,
KL Loss: 0.007476276718080044
It 3600: Total Loss: 0.016961954534053802,       Rec Loss: 0.01625492423772812,
KL Loss: 0.007070294115692377
It 3700: Total Loss: 0.017385436221957207,       Rec Loss: 0.016709398478269577,
KL Loss: 0.0067603797651827335
Run Epoch 4
It 3800: Total Loss: 0.016800912097096443,       Rec Loss: 0.016148000955581665,
KL Loss: 0.006529103498905897
It 3900: Total Loss: 0.018058255314826965,       Rec Loss: 0.01729017123579979,
KL Loss: 0.00768083892762661
It 4000: Total Loss: 0.015661168843507767,       Rec Loss: 0.015040346421301365,
KL Loss: 0.006208229344338179
It 4100: Total Loss: 0.016544895246624947,       Rec Loss: 0.015881886705756187,
KL Loss: 0.006630082614719868
It 4200: Total Loss: 0.0154840387403965,         Rec Loss: 0.014818349853157997,
KL Loss: 0.0066568851470947266
It 4300: Total Loss: 0.016851622611284256,       Rec Loss: 0.016245143488049507,
KL Loss: 0.006064796354621649
It 4400: Total Loss: 0.016024013981223106,       Rec Loss: 0.015370495617389679,
KL Loss: 0.006535189226269722
It 4500: Total Loss: 0.01767202652990818,        Rec Loss: 0.01702215149998665,
KL Loss: 0.006498758681118488
It 4600: Total Loss: 0.016455477103590965,       Rec Loss: 0.015848668292164803,
KL Loss: 0.006068084854632616
Run Epoch 5
It 4700: Total Loss: 0.014046993106603622,       Rec Loss: 0.013480709865689278,
KL Loss: 0.005662828683853149
It 4800: Total Loss: 0.016232404857873917,       Rec Loss: 0.01565631851553917,
```

```
KL Loss: 0.005760860629379749
It 4900: Total Loss: 0.016237996518611908,       Rec Loss: 0.015638213604688644,
KL Loss: 0.005997832864522934
It 5000: Total Loss: 0.01597602851688862,        Rec Loss: 0.015437951311469078,
KL Loss: 0.005380779504776001
It 5100: Total Loss: 0.014454067684710026,       Rec Loss: 0.01392856054008007,
KL Loss: 0.00525507191196084
It 5200: Total Loss: 0.01532320212572813,        Rec Loss: 0.014776825904846191,
KL Loss: 0.005463758483529091
It 5300: Total Loss: 0.013665499165654182,       Rec Loss: 0.01314383465796709,
KL Loss: 0.005216644611209631
It 5400: Total Loss: 0.015242756344377995,       Rec Loss: 0.014700818806886673,
KL Loss: 0.005419377703219652
It 5500: Total Loss: 0.015918370336294174,       Rec Loss: 0.015375419519841671,
KL Loss: 0.005429500713944435
It 5600: Total Loss: 0.014607840217649937,       Rec Loss: 0.014127781614661217,
KL Loss: 0.004800589755177498
Run Epoch 6
It 5700: Total Loss: 0.015762636438012123,       Rec Loss: 0.015255879610776901,
KL Loss: 0.00506757153198123
It 5800: Total Loss: 0.01430542953312397,        Rec Loss: 0.013816027902066708,
KL Loss: 0.004894017241895199
It 5900: Total Loss: 0.01463817898184061,        Rec Loss: 0.014157942496240139,
KL Loss: 0.004802363459020853
It 6000: Total Loss: 0.01329557690769434,        Rec Loss: 0.012843032367527485,
KL Loss: 0.00452544167637825
It 6100: Total Loss: 0.014610390178859234,       Rec Loss: 0.014134910888969898,
KL Loss: 0.004754796624183655
It 6200: Total Loss: 0.014015528373420238,       Rec Loss: 0.013533329591155052,
KL Loss: 0.004821992013603449
It 6300: Total Loss: 0.014473251067101955,       Rec Loss: 0.014006984420120716,
KL Loss: 0.00466266879811883
It 6400: Total Loss: 0.014005685225129128,       Rec Loss: 0.013569243252277374,
KL Loss: 0.004364421591162682
It 6500: Total Loss: 0.014451000839471817,       Rec Loss: 0.013965723104774952,
KL Loss: 0.004852775949984789
Run Epoch 7
It 6600: Total Loss: 0.01464562863111496,        Rec Loss: 0.014185508713126183,
KL Loss: 0.004601201508194208
It 6700: Total Loss: 0.012385326437652111,       Rec Loss: 0.011958125047385693,
KL Loss: 0.004272013902664185
It 6800: Total Loss: 0.014320113696157932,       Rec Loss: 0.013872917741537094,
KL Loss: 0.004471957217901945
It 6900: Total Loss: 0.013865350745618343,       Rec Loss: 0.01342741772532463,
KL Loss: 0.00437932601198554
It 7000: Total Loss: 0.01450097095221281,        Rec Loss: 0.014069486409425735,
KL Loss: 0.004314846359193325
It 7100: Total Loss: 0.014511597342789173,       Rec Loss: 0.014066338539123535,
```

```
KL Loss: 0.004452588502317667
It 7200: Total Loss: 0.013345606625080109,       Rec Loss: 0.012927805073559284,
KL Loss: 0.0041780127212405205
It 7300: Total Loss: 0.013423586264252663,       Rec Loss: 0.013011832721531391,
KL Loss: 0.004117533564567566
It 7400: Total Loss: 0.013585799373686314,       Rec Loss: 0.013178559951484203,
KL Loss: 0.004072397015988827
Run Epoch 8
It 7500: Total Loss: 0.014197160489857197,       Rec Loss: 0.013791125267744064,
KL Loss: 0.004060348961502314
It 7600: Total Loss: 0.014399624429643154,       Rec Loss: 0.014013996347784996,
KL Loss: 0.003856282914057374
It 7700: Total Loss: 0.01609669253230095,        Rec Loss: 0.015674389898777008,
KL Loss: 0.0042230174876749516
It 7800: Total Loss: 0.014180546626448631,       Rec Loss: 0.013767818920314312,
KL Loss: 0.00412727240473032
It 7900: Total Loss: 0.01379569061100483,        Rec Loss: 0.013404476456344128,
KL Loss: 0.003912144806236029
It 8000: Total Loss: 0.013266888447105885,       Rec Loss: 0.012890498153865337,
KL Loss: 0.0037639017682522535
It 8100: Total Loss: 0.013153398409485817,       Rec Loss: 0.012782149948179722,
KL Loss: 0.003712485311552882
It 8200: Total Loss: 0.013356908224523067,       Rec Loss: 0.013015015050768852,
KL Loss: 0.0034189284779131413
It 8300: Total Loss: 0.014627931639552116,       Rec Loss: 0.014237520284950733,
KL Loss: 0.0039041144773364067
It 8400: Total Loss: 0.01273256167769432,        Rec Loss: 0.012377734296023846,
KL Loss: 0.00354827125556767
Run Epoch 9
It 8500: Total Loss: 0.01435922272503376,        Rec Loss: 0.013977090828120708,
KL Loss: 0.0038213226944208145
It 8600: Total Loss: 0.013424484990537167,       Rec Loss: 0.013042627833783627,
KL Loss: 0.0038185743615031242
It 8700: Total Loss: 0.01462989579886198,        Rec Loss: 0.0142715685069561,
KL Loss: 0.0035832696594297886
It 8800: Total Loss: 0.013155517168343067,       Rec Loss: 0.012800808064639568,
KL Loss: 0.0035470868460834026
It 8900: Total Loss: 0.012628809548914433,       Rec Loss: 0.012286745011806488,
KL Loss: 0.0034206470008939505
It 9000: Total Loss: 0.013413965702056885,       Rec Loss: 0.013058984652161598,
KL Loss: 0.003549807472154498
It 9100: Total Loss: 0.01322172675281763,        Rec Loss: 0.012867002747952938,
KL Loss: 0.0035472363233566284
It 9200: Total Loss: 0.013139035552740097,       Rec Loss: 0.012805615551769733,
KL Loss: 0.0033341986127197742
It 9300: Total Loss: 0.013532864861190319,       Rec Loss: 0.013176078908145428,
KL Loss: 0.003567855805158615
Run Epoch 10
```

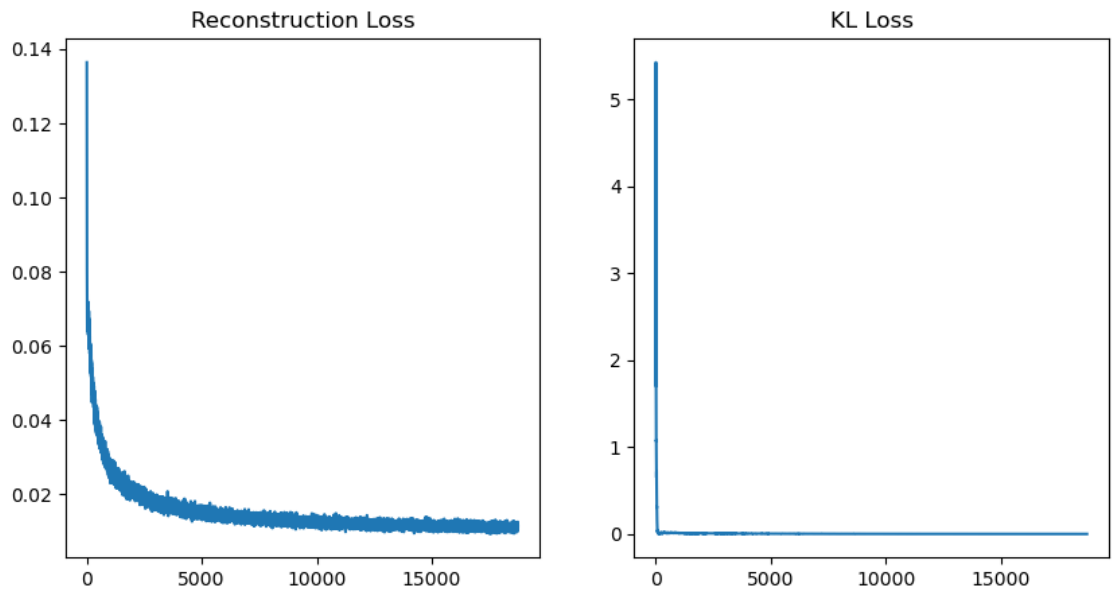
```
It 9400: Total Loss: 0.011851323768496513,       Rec Loss: 0.01152653805911541,
KL Loss: 0.003247853834182024
It 9500: Total Loss: 0.013270624913275242,       Rec Loss: 0.012938653118908405,
KL Loss: 0.0033197137527167797
It 9600: Total Loss: 0.012282073497772217,       Rec Loss: 0.011947396211326122,
KL Loss: 0.0033467719331383705
It 9700: Total Loss: 0.013174828141927719,       Rec Loss: 0.012842810712754726,
KL Loss: 0.003320177784189582
It 9800: Total Loss: 0.011970028281211853,       Rec Loss: 0.0116460295394063,
KL Loss: 0.0032399827614426613
It 9900: Total Loss: 0.012120638974010944,       Rec Loss: 0.011821642518043518,
KL Loss: 0.0029899664223194122
It 10000: Total Loss: 0.014740396291017532,      Rec Loss: 0.014392955228686333,
KL Loss: 0.0034744087606668472
It 10100: Total Loss: 0.012152183800935745,      Rec Loss: 0.011841881088912487,
KL Loss: 0.003103023860603571
It 10200: Total Loss: 0.012704617343842983,      Rec Loss: 0.012402459047734737,
KL Loss: 0.0030215848237276077
It 10300: Total Loss: 0.01249126996845007,       Rec Loss: 0.012179804034531116,
KL Loss: 0.0031146628316491842
Run Epoch 11
It 10400: Total Loss: 0.01260558795183897,       Rec Loss: 0.012297169305384159,
KL Loss: 0.00308418576605618
It 10500: Total Loss: 0.012067405506968498,      Rec Loss: 0.0117472680285573,
KL Loss: 0.0032013722229748964
It 10600: Total Loss: 0.012111160904169083,      Rec Loss: 0.011789915151894093,
KL Loss: 0.003212460083886981
It 10700: Total Loss: 0.012799462303519249,      Rec Loss: 0.012490133754909039,
KL Loss: 0.0030932859517633915
It 10800: Total Loss: 0.012297934852540493,      Rec Loss: 0.011987019330263138,
KL Loss: 0.0031091521959751844
It 10900: Total Loss: 0.012873368337750435,      Rec Loss: 0.012570145539939404,
KL Loss: 0.003032232169061899
It 11000: Total Loss: 0.012015806511044502,      Rec Loss: 0.01172984391450882,
KL Loss: 0.0028596287593245506
It 11100: Total Loss: 0.01280362531542778,       Rec Loss: 0.012511180713772774,
KL Loss: 0.002924449509009719
It 11200: Total Loss: 0.01233730185776949,       Rec Loss: 0.012043643742799759,
KL Loss: 0.0029365778900682926
Run Epoch 12
It 11300: Total Loss: 0.013989914208650589,      Rec Loss: 0.013675122521817684,
KL Loss: 0.003147913608700037
It 11400: Total Loss: 0.01190387737005949,       Rec Loss: 0.011631780304014683,
KL Loss: 0.002720970194786787
It 11500: Total Loss: 0.011269115842878819,      Rec Loss: 0.010986371897161007,
KL Loss: 0.0028274422511458397
It 11600: Total Loss: 0.01176472194492817,       Rec Loss: 0.01147956307977438,
KL Loss: 0.002851585391908884
```

```
It 11700: Total Loss: 0.014068718999624252,      Rec Loss: 0.013780061155557632,
KL Loss: 0.002886576810851693
It 11800: Total Loss: 0.011106230318546295,      Rec Loss: 0.010839959606528282,
KL Loss: 0.0026627075858414173
It 11900: Total Loss: 0.011986861936748028,      Rec Loss: 0.011713488027453423,
KL Loss: 0.0027337369974702597
It 12000: Total Loss: 0.010944233275949955,      Rec Loss: 0.010673627257347107,
KL Loss: 0.002706058556213975
It 12100: Total Loss: 0.012364882975816727,      Rec Loss: 0.012084118090569973,
KL Loss: 0.002807644195854664
Run Epoch 13
It 12200: Total Loss: 0.013034729287028313,      Rec Loss: 0.012734544463455677,
KL Loss: 0.0030018524266779423
It 12300: Total Loss: 0.012120491825044155,      Rec Loss: 0.011836996302008629,
KL Loss: 0.002834954299032688
It 12400: Total Loss: 0.012051887810230255,      Rec Loss: 0.01176355965435505,
KL Loss: 0.0028832806274294853
It 12500: Total Loss: 0.011735895648598671,      Rec Loss: 0.011476295068860054,
KL Loss: 0.0025960062630474567
It 12600: Total Loss: 0.010949919000267982,      Rec Loss: 0.0106950169429183,
KL Loss: 0.0025490219704806805
It 12700: Total Loss: 0.011888845823705196,      Rec Loss: 0.011627928353846073,
KL Loss: 0.0026091751642525196
It 12800: Total Loss: 0.011762158945202827,      Rec Loss: 0.011502447538077831,
KL Loss: 0.0025971136055886745
It 12900: Total Loss: 0.013636549934744835,      Rec Loss: 0.01334590557962656,
KL Loss: 0.0029064458794891834
It 13000: Total Loss: 0.011240217834711075,      Rec Loss: 0.01098562777042389,
KL Loss: 0.002545896451920271
It 13100: Total Loss: 0.01074913889169693,       Rec Loss: 0.010489935986697674,
KL Loss: 0.0025920290499925613
Run Epoch 14
It 13200: Total Loss: 0.01167720090597868,       Rec Loss: 0.011406617239117622,
KL Loss: 0.0027058369014412165
It 13300: Total Loss: 0.012320187874138355,      Rec Loss: 0.012055027298629284,
KL Loss: 0.0026516076177358627
It 13400: Total Loss: 0.012438078410923481,      Rec Loss: 0.01217565592378378,
KL Loss: 0.0026242269668728113
It 13500: Total Loss: 0.01186853926628828,       Rec Loss: 0.01161801815032959,
KL Loss: 0.002505215583369136
It 13600: Total Loss: 0.012135780416429043,      Rec Loss: 0.011878018267452717,
KL Loss: 0.0025776238180696964
It 13700: Total Loss: 0.01180735882371664,       Rec Loss: 0.011552386917173862,
KL Loss: 0.0025497213937342167
It 13800: Total Loss: 0.011649305932223797,      Rec Loss: 0.011396761983633041,
KL Loss: 0.0025254362262785435
It 13900: Total Loss: 0.01237491238862276,       Rec Loss: 0.012102445587515831,
KL Loss: 0.002724671270698309
```

```
It 14000: Total Loss: 0.011339472606778145,      Rec Loss: 0.011098641902208328,
KL Loss: 0.0024083054158836603
Run Epoch 15
It 14100: Total Loss: 0.012398788705468178,      Rec Loss: 0.012146325781941414,
KL Loss: 0.002524628769606352
It 14200: Total Loss: 0.01243450678884983,       Rec Loss: 0.012182867154479027,
KL Loss: 0.002516393782570958
It 14300: Total Loss: 0.012178373523056507,      Rec Loss: 0.011915851384401321,
KL Loss: 0.002625223947688937
It 14400: Total Loss: 0.01071881502866745,       Rec Loss: 0.010467338375747204,
KL Loss: 0.002514767460525036
It 14500: Total Loss: 0.011784997768700123,      Rec Loss: 0.011537191458046436,
KL Loss: 0.002478065202012658
It 14600: Total Loss: 0.012181549333035946,      Rec Loss: 0.011920752935111523,
KL Loss: 0.0026079611852765083
It 14700: Total Loss: 0.011524539440870285,      Rec Loss: 0.011289281770586967,
KL Loss: 0.0023525748401880264
It 14800: Total Loss: 0.010394169948995113,      Rec Loss: 0.010163495317101479,
KL Loss: 0.0023067486472427845
It 14900: Total Loss: 0.012234454974532127,      Rec Loss: 0.011995145119726658,
KL Loss: 0.0023931015748530626
Run Epoch 16
It 15000: Total Loss: 0.010655034333467484,      Rec Loss: 0.010415563359856606,
KL Loss: 0.0023947071749716997
It 15100: Total Loss: 0.01126840990036726,       Rec Loss: 0.011042592115700245,
KL Loss: 0.0022581759840250015
It 15200: Total Loss: 0.01196220051497221,       Rec Loss: 0.011722846888005733,
KL Loss: 0.0023935376666486263
It 15300: Total Loss: 0.011739430949091911,      Rec Loss: 0.011503322049975395,
KL Loss: 0.002361088525503874
It 15400: Total Loss: 0.010840686969459057,      Rec Loss: 0.010614283382892609,
KL Loss: 0.002264037262648344
It 15500: Total Loss: 0.010637227445840836,      Rec Loss: 0.010410754010081291,
KL Loss: 0.0022647310979664326
It 15600: Total Loss: 0.011592690832912922,      Rec Loss: 0.0113602289929986,
KL Loss: 0.0023246153723448515
It 15700: Total Loss: 0.01220970880240202,       Rec Loss: 0.011975755915045738,
KL Loss: 0.0023395325988531113
It 15800: Total Loss: 0.011106397025287151,      Rec Loss: 0.010884858667850494,
KL Loss: 0.0022153877653181553
It 15900: Total Loss: 0.011098278686404228,      Rec Loss: 0.010872863233089447,
KL Loss: 0.002254154998809099
Run Epoch 17
It 16000: Total Loss: 0.011259224265813828,      Rec Loss: 0.011035076342523098,
KL Loss: 0.002241482026875019
It 16100: Total Loss: 0.012495594099164009,      Rec Loss: 0.012270350940525532,
KL Loss: 0.0022524362429976463
It 16200: Total Loss: 0.011244098655879498,      Rec Loss: 0.011023652739822865,
```

KL Loss: 0.0022044561337679625
It 16300: Total Loss: 0.011909028515219688, Rec Loss: 0.011676556430757046,
KL Loss: 0.0023247250355780125
It 16400: Total Loss: 0.011192245408892632, Rec Loss: 0.010966039262712002,
KL Loss: 0.002262064255774021
It 16500: Total Loss: 0.011794046498835087, Rec Loss: 0.01156585942953825,
KL Loss: 0.0022818748839199543
It 16600: Total Loss: 0.01078678760677576, Rec Loss: 0.010563230141997337,
KL Loss: 0.0022355779074132442
It 16700: Total Loss: 0.011812394484877586, Rec Loss: 0.011597594246268272,
KL Loss: 0.002147998660802841
It 16800: Total Loss: 0.012661381624639034, Rec Loss: 0.012439623475074768,
KL Loss: 0.0022175770718604326
Run Epoch 18
It 16900: Total Loss: 0.012763128615915775, Rec Loss: 0.012523832730948925,
KL Loss: 0.0023929625749588013
It 17000: Total Loss: 0.010406303219497204, Rec Loss: 0.010193577036261559,
KL Loss: 0.002127258572727442
It 17100: Total Loss: 0.01194713544100523, Rec Loss: 0.011736171320080757,
KL Loss: 0.0021096395794302225
It 17200: Total Loss: 0.011554962955415249, Rec Loss: 0.011330465786159039,
KL Loss: 0.0022449735552072525
It 17300: Total Loss: 0.011295911855995655, Rec Loss: 0.011080420576035976,
KL Loss: 0.002154915127903223
It 17400: Total Loss: 0.011568128131330013, Rec Loss: 0.011358019895851612,
KL Loss: 0.0021010865457355976
It 17500: Total Loss: 0.010671177878975868, Rec Loss: 0.010465271770954132,
KL Loss: 0.002059063408523798
It 17600: Total Loss: 0.010946108028292656, Rec Loss: 0.010745218954980373,
KL Loss: 0.002008887240663171
It 17700: Total Loss: 0.010589144192636013, Rec Loss: 0.010382510721683502,
KL Loss: 0.0020663319155573845
It 17800: Total Loss: 0.010817582719027996, Rec Loss: 0.01060649286955595,
KL Loss: 0.002110897097736597
Run Epoch 19
It 17900: Total Loss: 0.01064419373869896, Rec Loss: 0.010439356788992882,
KL Loss: 0.0020483708940446377
It 18000: Total Loss: 0.011037820018827915, Rec Loss: 0.0108172707259655,
KL Loss: 0.0022054961882531643
It 18100: Total Loss: 0.011686167679727077, Rec Loss: 0.01147804968059063,
KL Loss: 0.0020811844151467085
It 18200: Total Loss: 0.01126767136156559, Rec Loss: 0.01106226909905672,
KL Loss: 0.002054020995274186
It 18300: Total Loss: 0.01075480692088604, Rec Loss: 0.010553145781159401,
KL Loss: 0.002016613259911537
It 18400: Total Loss: 0.010523932054638863, Rec Loss: 0.0103126997128129,
KL Loss: 0.002112321788445115
It 18500: Total Loss: 0.011303627863526344, Rec Loss: 0.011106887832283974,

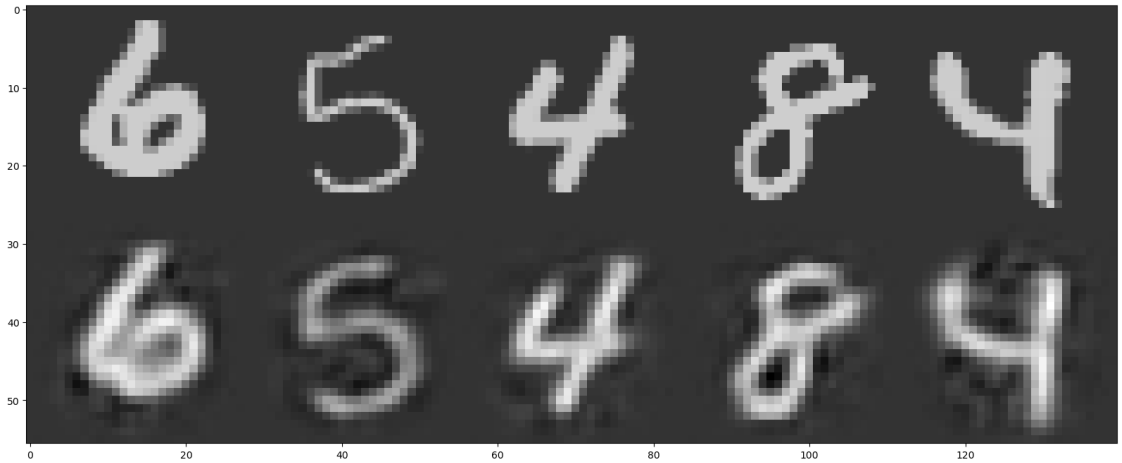
```
KL Loss: 0.0019674012437462807
It 18600: Total Loss: 0.010327789932489395,      Rec Loss: 0.010134424082934856,
KL Loss: 0.001933656632900238
It 18700: Total Loss: 0.01149399857968092,       Rec Loss: 0.011282269842922688,
KL Loss: 0.0021172917913645506
Done!
```



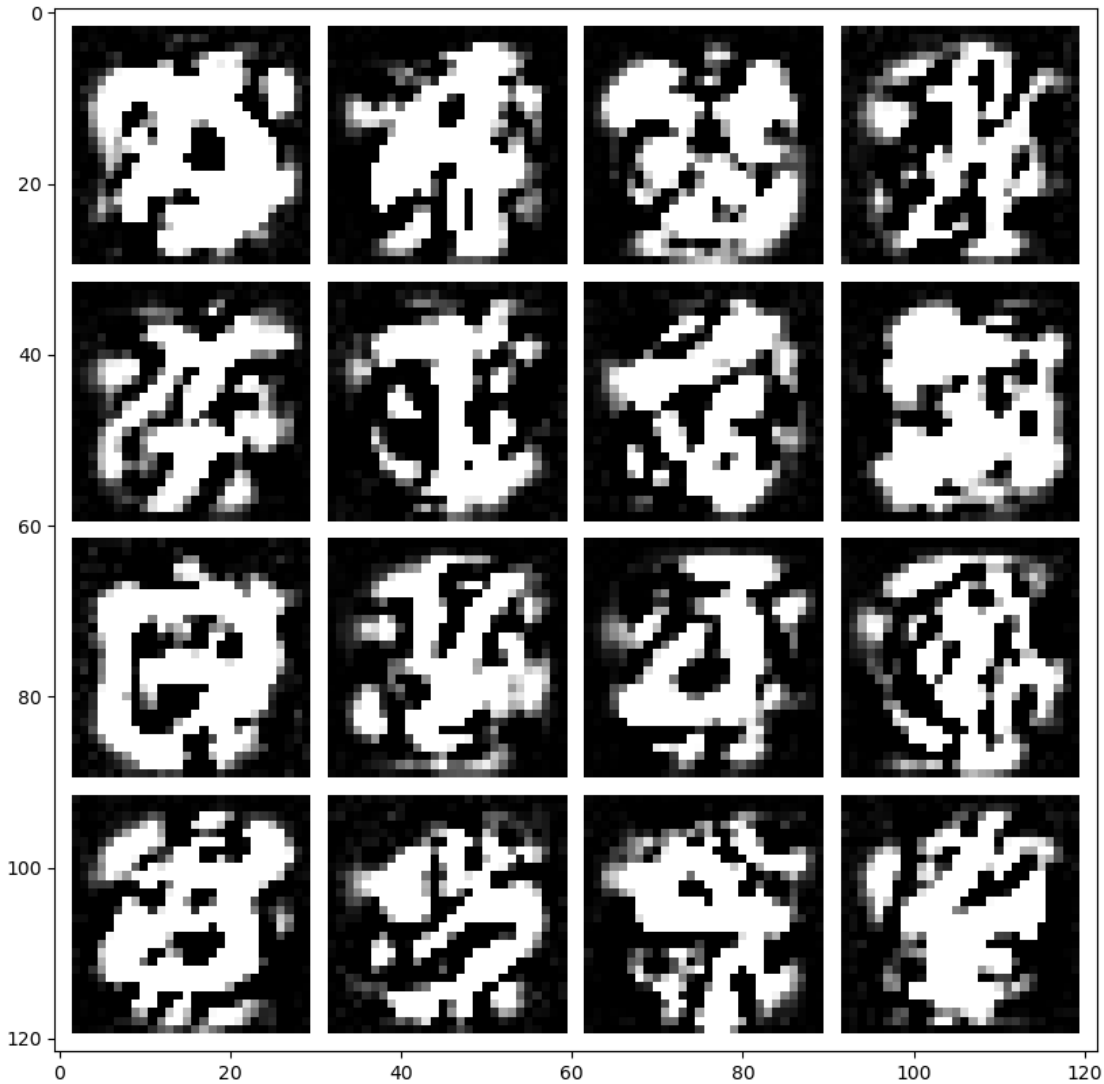
Let's look at some reconstructions and decoded embedding samples for this beta!

[21]:
```
# [OPTIONAL] visualize VAE reconstructions and samples from the generative model
print("BEST beta = ", beta)
vis_reconstruction(vae_model, randomize=True)
vis_samples(vae_model)
```

```
BEST beta =  0.1
```



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



5 4. Embedding Space Interpolation [3pt]

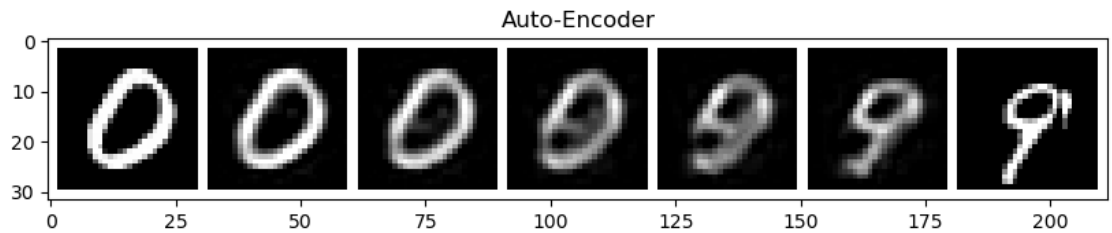
As mentioned in the introduction, AEs and VAEs cannot only be used to generate images, but also to learn low-dimensional representations of their inputs. In this final section we will investigate the representations we learned with both models by **interpolating in embedding space** between different images. We will encode two images into their low-dimensional embedding representations, then interpolate these embeddings and reconstruct the result.

[22]:
```
# Prob1-7
nz=32

def get_image_with_label(target_label):
  """Returns a random image from the training set with the requested digit."""
```

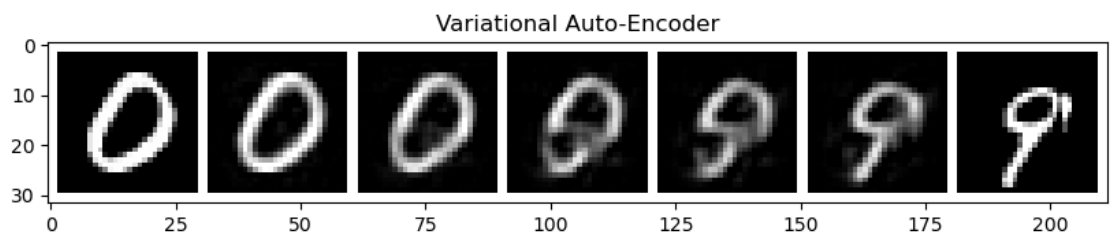
```
  for img_batch, label_batch in mnist_data_loader:
    for img, label in zip(img_batch, label_batch):
      if label == target_label:
        return img.to(device)

def interpolate_and_visualize(model, tag, start_img, end_img):
  """Encodes images and performs interpolation. Displays decodings."""
  model.eval()    # put model in eval mode to avoid updating batchnorm

  # encode both images into embeddings (use posterior mean for interpolation)
  z_start = model.encoder(start_img[None].reshape(1,784))[..., :nz]
  z_end = model.encoder(end_img[None].reshape(1,784))[..., :nz]

  # compute interpolated latents
  N_INTER_STEPS = 5
  z_inter = [z_start + i/N_INTER_STEPS * (z_end - z_start) for i in␣
 ↪range(N_INTER_STEPS)]

  # decode interpolated embeddings (as a single batch)
  img_inter = model.decoder(torch.cat(z_inter))
  img_inter = img_inter.reshape(-1, 28, 28)

  # reshape result and display interpolation
  vis_imgs = torch.cat([start_img, img_inter, end_img]).reshape(-1,1,28,28)
  fig = plt.figure(figsize = (10, 10))
  ax1 = plt.subplot(111)
  ax1.imshow(torchvision.utils.make_grid(vis_imgs, nrow=N_INTER_STEPS+2,␣
 ↪pad_value=1.)\
                 .data.cpu().numpy().transpose(1, 2, 0), cmap='gray')
  plt.title(tag)
  plt.show()


### Interpolation 1
START_LABEL = 0 # ... TODO CHOOSE
END_LABEL = 9 # ... TODO CHOOSE
# sample two training images with given labels
start_img = get_image_with_label(START_LABEL)
end_img = get_image_with_label(END_LABEL)
# visualize interpolations for AE and VAE models
interpolate_and_visualize(ae_model, "Auto-Encoder", start_img, end_img)
interpolate_and_visualize(vae_model, "Variational Auto-Encoder", start_img,␣
 ↪end_img)

### Interpolation 2
START_LABEL = 4  # ... TODO CHOOSE
END_LABEL = 7 # ... TODO CHOOSE
```

```
# sample two training images with given labels
start_img = get_image_with_label(START_LABEL)
end_img = get_image_with_label(END_LABEL)
# visualize interpolations for AE and VAE models
interpolate_and_visualize(ae_model, "Auto-Encoder", start_img, end_img)
interpolate_and_visualize(vae_model, "Variational Auto-Encoder", start_img,␣
↪end_img)
```

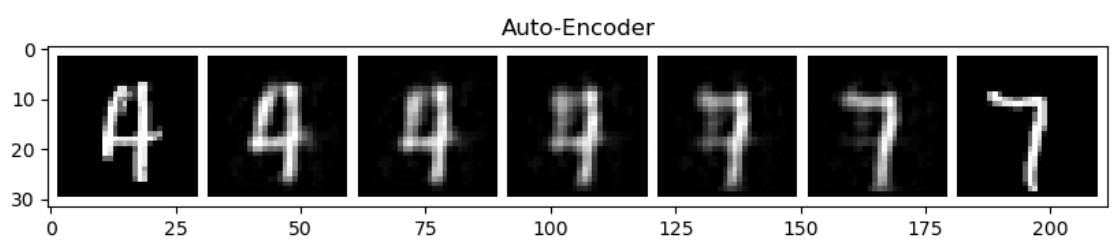
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

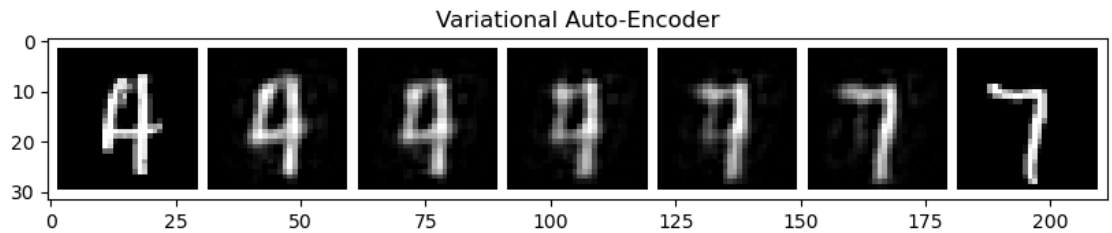



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
```

floats or [0..255] for integers).



Repeat the experiment for different start / end labels and different samples. Describe your observations.

> **Prob1-7 continued: Inline Question: Repeat the interpolation experiment with different start / end labels and multiple samples. Describe your observations! [2 pt]** 1. How do AE and VAE embedding space interpolations differ? 2. How do you expect these differences to affect the usefulness of the learned representation for downstream learning? (max 300 words)
>
> **Answer**: 1. They are relatively the same in the final output but the transistions between digits converge faster to the actual digit in a VAE.
>
> **Answer** 2. AE cannot encapsulate the latent distributions as well as as VAE can and when doing dimension reduction, VAE's will be much better suited.

6 5. Conditional VAE

Let us now try a Conditional VAE Now we will try to create a Conditional VAE, where we can condition the encoder and decoder of the VAE on the label `c`.

6.1 Defining the conditional Encoder, Decoder, and VAE models [5 pt]

Prob1-8. We create a separate encoder and decoder class that take in an additional argument `c` in their forward pass, and then build our CVAE model on top of it. Note that the encoder and decoder just need to append `c` to the standard inputs to these modules.

[23]:
```
def idx2onehot(idx, n):
    """Converts a batch of indices to a one-hot representation."""
    assert torch.max(idx).item() < n
    if idx.dim() == 1:
        idx = idx.unsqueeze(1)
    onehot = torch.zeros(idx.size(0), n).to(idx.device)
    onehot.scatter_(1, idx, 1)

    return onehot

# Let's define encoder and decoder networks
```

```
class CVAEEncoder(nn.Module):
  def __init__(self, nz, input_size, conditional, num_labels):
    super().__init__()
    self.input_size = input_size + num_labels if conditional else input_size
    self.num_labels = num_labels
    self.conditional = conditional

    ################################## TODO␣
↪#########################################
    # Create the network architecture using a nn.Sequential module wrapper.    ␣
↪   #
    # Encoder Architecture:                                                    ␣
↪   #
    # - input_size -> 256                                                      ␣
↪   #
    # - ReLU                                                                   ␣
↪   #
    # - 256 -> 64                                                              ␣
↪   #
    # - ReLU                                                                   ␣
↪   #
    # - 64 -> nz                                                               ␣
↪   #
    # HINT: Verify the shapes of intermediate layers by running partial␣
↪networks   #
    #        (with the next notebook cell) and visualizing the output shapes.  ␣
↪   #
  ␣
↪#################################################################################
    self.net = nn.Sequential(
       nn.Linear(self.input_size,256),
       nn.ReLU(),
       nn.Linear(256, 64),
       nn.ReLU(),
       nn.Linear(64, nz)
    )

    ################################# END TODO␣
↪######################################

  def forward(self, x, c=None):
    ################################## TODO␣
↪#########################################
    # If using conditional VAE, concatenate x and a onehot version of c to␣
↪create  #
```

```
    # the full input. Use function idx2onehot above.                          ␣
↪   #
  ␣
↪##############################################################################
    if self.conditional:
        c = idx2onehot(c, self.num_labels)
        x = torch.cat([x, c], dim=1)
  ␣
↪##############################################################################
    return self.net(x)


class CVAEDecoder(nn.Module):
  def __init__(self, nz, output_size, conditional, num_labels):
    super().__init__()
    self.output_size = output_size
    self.conditional = conditional
    self.num_labels = num_labels
    if self.conditional:
        nz = nz + num_labels
    ################################## TODO␣
↪########################################
    # Create the network architecture using a nn.Sequential module wrapper.    ␣
↪   #
    # Decoder Architecture (mirrors encoder architecture):                     ␣
↪   #
    # - nz -> 64                                                               ␣
↪   #
    # - ReLU                                                                   ␣
↪   #
    # - 64 -> 256                                                              ␣
↪   #
    # - ReLU                                                                   ␣
↪   #
    # - 256 -> output_size                                                     ␣
↪   #
  ␣
↪##############################################################################
    self.net = nn.Sequential(
        nn.Linear(nz, 64),
        nn.ReLU(),
        nn.Linear(64, 256),
        nn.ReLU(),
        nn.Linear(256, output_size)
    )
```

```
    ################################# END TODO␣
↪#######################################

  def forward(self, z, c=None):
    ################################# TODO␣
↪#########################################
    # If using conditional VAE, concatenate z and a onehot version of c to␣
↪create  #
    # the full embedding. Use function idx2onehot above.                    ␣
↪   #
  ␣
↪##############################################################################
    if self.conditional:
        c = idx2onehot(c, self.num_labels)
        z = torch.cat([z, c], dim=1)
    ################################# END TODO␣
↪#######################################

    return self.net(z).reshape(-1, 1, self.output_size)


class CVAE(nn.Module):
    def __init__(self, nz, beta=1.0, conditional=False, num_labels=0):
        super().__init__()
        if conditional:
            assert num_labels > 0
        self.beta = beta
        self.encoder = CVAEEncoder(2*nz, input_size=in_size,␣
↪conditional=conditional, num_labels=num_labels)
        self.decoder = CVAEDecoder(nz, output_size=out_size,␣
↪conditional=conditional, num_labels=num_labels)

    def forward(self, x, c=None):
        if x.dim() > 2:
            x = x.view(-1, 28*28)

        q = self.encoder(x,c)
        mu, log_sigma = torch.chunk(q, 2, dim=-1)

        # sample latent variable z with reparametrization
        eps = torch.normal(mean=torch.zeros_like(mu), std=torch.
↪ones_like(log_sigma))
        # eps = torch.randn_like(mu) # Alternatively use this
        z = mu + eps * torch.exp(log_sigma)

        # compute reconstruction
```

```
        reconstruction = self.decoder(z, c)

        return {'q': q, 'rec': reconstruction, 'c': c}

    def loss(self, x, outputs):
        ######################################## TODO␣
↪#######################################
        # Implement the loss computation of the VAE.                                  ␣
↪            #
        # HINT: Your code should implement the following steps:                       ␣
↪            #
        #           1. compute the image reconstruction loss, similar to AE loss␣
↪above      #
        #           2. compute the KL divergence loss between the inferred␣
↪posterior        #
        #              distribution and a unit Gaussian prior; you can use the␣
↪provided     #
        #              function above for computing the KL divergence between␣
↪two Gaussians #
        #              parametrized by mean and log_sigma                          ␣
↪            #
        # HINT: Make sure to compute the KL divergence in the correct order␣
↪since it is    #
        #       not symmetric!!  ie. KL(p, q) != KL(q, p)                           ␣
↪            #
       ␣
↪##########################################################################################
        rec_loss = nn.functional.mse_loss(outputs['rec'].squeeze(), x)
        mu, log_sigma = torch.chunk(outputs['q'], chunks=2, dim=-1)
        kl_loss = kl_divergence(mu, log_sigma, torch.zeros_like(mu), torch.
↪zeros_like(log_sigma)).mean()

        ##################################### END TODO␣
↪#######################################

        # return weighted objective
        return rec_loss + self.beta * kl_loss, \
            {'rec_loss': rec_loss, 'kl_loss': kl_loss}

    def reconstruct(self, x, c=None):
        """Use mean of posterior estimate for visualization reconstruction."""
        ######################################## TODO␣
↪#######################################
        # This function is used for visualizing reconstructions of our VAE␣
↪model. To       #
```

```
        # obtain the maximum likelihood estimate we bypass the sampling␣
↪procedure of the   #
        # inferred latent and instead directly use the mean of the inferred␣
↪posterior.     #
        # HINT: encode the input image and then decode the mean of the␣
↪posterior to obtain #
        #       the reconstruction.                                        ␣
↪            #
      ␣
↪#######################################################################################
        q = self.encoder(x, c)
        mu, _ = torch.chunk(q, 2, dim=-1)
        image = self.decoder(mu).reshape(-1, 28, 28)
        ##################################### END TODO␣
↪#######################################
        return image
```

6.2 Setting up the CVAE Training loop

[24]:
```
learning_rate = 1e-3
nz = 32

######################################### TODO␣
↪#######################################
# Tune the beta parameter to obtain good training results. However, for the    #
# initial experiments leave beta = 0 in order to verify our implementation.    ␣
↪    #
#######################################################################################
epochs = 5 # works with fewer epochs than AE, VAE. we only test conditional␣
↪samples.
beta =  0.5
##################################### END TODO␣
↪#######################################

# build CVAE model
conditional = True
cvae_model = CVAE(nz, beta, conditional=conditional, num_labels=10).to(device) ␣
↪  # transfer model to GPU if available
cvae_model = cvae_model.train()   # set model in train mode (eg batchnorm␣
↪params get updated)

# build optimizer and loss function
######################################### TODO␣
↪#######################################
# Build the optimizer for the cvae_model. We will again use the Adam optimizer␣
↪with #
```

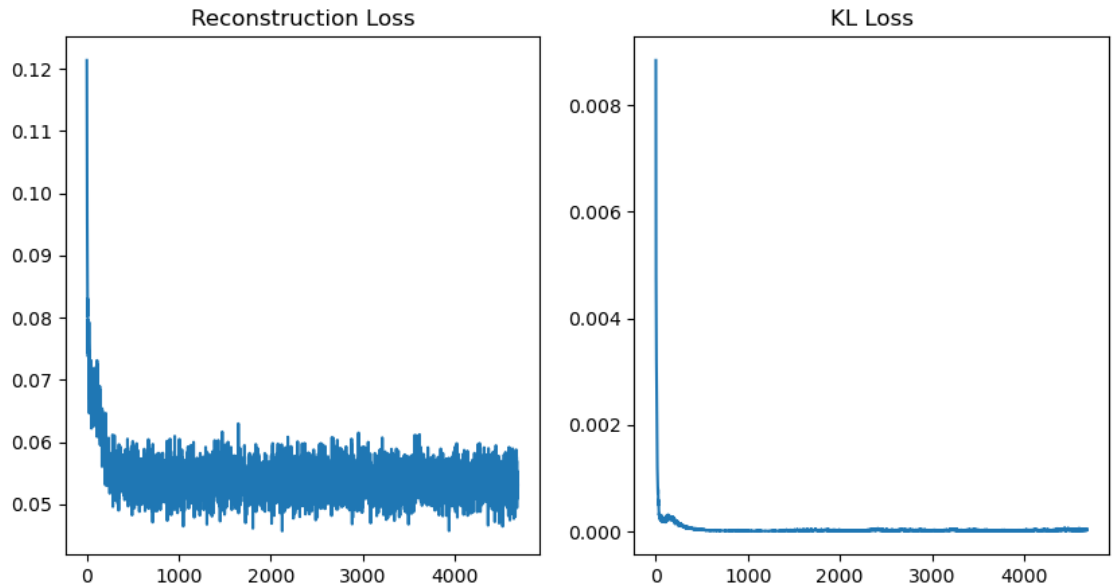
```
# the given learning rate and otherwise default parameters.                    ␣
↪    #
##################################################################################
# same as AE
from torch.optim import Adam
optimizer = Adam(cvae_model.parameters(), lr=learning_rate)
#################################### END TODO␣
↪#######################################

train_it = 0
rec_loss, kl_loss = [], []
print(f"Running {epochs} epochs with {beta=}")
for ep in range(epochs):
  print(f"Run Epoch {ep}")
  ######################################## TODO␣
↪########################################
  # Implement the main training loop for the model.                            ␣
↪      #
  # If using conditional VAE, remember to pass the conditional variable c to␣
↪the      #
  # forward pass                                                               ␣
↪      #
  # HINT: Your training loop should sample batches from the data loader, run␣
↪the      #
  #       forward pass of the model, compute the loss, perform the backward␣
↪pass and #
  #       perform one gradient step with the optimizer.                        ␣
↪      #
  # HINT: Don't forget to erase old gradients before performing the backward␣
↪pass.    #
  # HINT: As before, we will use the loss() function of our model for computing␣
↪the  #
  #       training loss. It outputs the total training loss and a dict␣
↪containing    #
  #       the breakdown of reconstruction and KL loss.                         ␣
↪      #

 ␣
↪##################################################################################
  for indx, (img, label) in enumerate(mnist_data_loader): # index and img,label␣
↪tuple
    optimizer.zero_grad()
    batch_features = torch.reshape(img, (-1, 784) ).to(device)
    cvae_output = cvae_model(batch_features, label)
    total_loss, losses = cvae_model.loss(batch_features, outputs=cvae_output)
    rec_loss.append(losses['rec_loss']); kl_loss.append(losses['kl_loss'])
    total_loss.backward()
```

```
    optimizer.step()

    if train_it % 100 == 0:
      print("It {}: Total Loss: {}, \t Rec Loss: {},\t KL Loss: {}"\
            .format(train_it, total_loss, losses['rec_loss'],␣
 ↪losses['kl_loss']))
    train_it += 1
  ##################################### END TODO␣
 ↪#####################################

print("Done!")

rec_loss_plotdata = [foo.detach().cpu() for foo in rec_loss]
kl_loss_plotdata = [foo.detach().cpu() for foo in kl_loss]

# log the loss training curves
fig = plt.figure(figsize = (10, 5))
ax1 = plt.subplot(121)
ax1.plot(rec_loss_plotdata)
ax1.title.set_text("Reconstruction Loss")
ax2 = plt.subplot(122)
ax2.plot(kl_loss_plotdata)
ax2.title.set_text("KL Loss")
plt.show()
```

```
Running 5 epochs with beta=0.5
Run Epoch 0
It 0: Total Loss: 0.1257585734128952,    Rec Loss: 0.12133775651454926,  KL
Loss: 0.008841624483466148
It 100: Total Loss: 0.06668810546398163,         Rec Loss: 0.06659094989299774,
KL Loss: 0.00019431416876614094
It 200: Total Loss: 0.057594798505306244,        Rec Loss: 0.057484276592731476,
KL Loss: 0.00022104088566266
It 300: Total Loss: 0.06110769137740135,         Rec Loss: 0.06105879694223404,
KL Loss: 9.778619278222322e-05
It 400: Total Loss: 0.05564690753817558,         Rec Loss: 0.05562187731266022,
KL Loss: 5.006376886740327e-05
It 500: Total Loss: 0.05231514573097229,         Rec Loss: 0.05229977145791054,
KL Loss: 3.0748575227335095e-05
It 600: Total Loss: 0.05366450175642967,         Rec Loss: 0.05365365743637085,
KL Loss: 2.1690299035981297e-05
It 700: Total Loss: 0.054544854909181595,        Rec Loss: 0.05453727766871452,
KL Loss: 1.5152094420045614e-05
It 800: Total Loss: 0.05498046055436134,         Rec Loss: 0.054973501712083817,
KL Loss: 1.3918543118052185e-05
It 900: Total Loss: 0.056996166706085205,        Rec Loss: 0.05698634311556816,
KL Loss: 1.964364491868764e-05
```

```
Run Epoch 1
It 1000: Total Loss: 0.0583464615046978,        Rec Loss: 0.05833946913480759,
KL Loss: 1.3984201359562576e-05
It 1100: Total Loss: 0.05548424646258354,       Rec Loss: 0.05547903850674629,
KL Loss: 1.0413263225927949e-05
It 1200: Total Loss: 0.053430814296007156,      Rec Loss: 0.053426653146743774,
KL Loss: 8.324786904267967e-06
It 1300: Total Loss: 0.05262302607297897,       Rec Loss: 0.05261838808655739,
KL Loss: 9.275099728256464e-06
It 1400: Total Loss: 0.052419405430555344,      Rec Loss: 0.052409570664167404,
KL Loss: 1.9666607840918005e-05
It 1500: Total Loss: 0.056570038199424744,      Rec Loss: 0.056556351482868195,
KL Loss: 2.737631439231336e-05
It 1600: Total Loss: 0.05785296484827995,       Rec Loss: 0.05784217640757561,
KL Loss: 2.1580388420261443e-05
It 1700: Total Loss: 0.057336483150720596,      Rec Loss: 0.05732434242963791,
KL Loss: 2.4282810045406222e-05
It 1800: Total Loss: 0.05222703143954277,       Rec Loss: 0.05221225693821907,
KL Loss: 2.9547314625233412e-05
Run Epoch 2
It 1900: Total Loss: 0.05235359072685242,       Rec Loss: 0.05234529450535774,
KL Loss: 1.659471308812499e-05
It 2000: Total Loss: 0.05598422884941101,       Rec Loss: 0.055975291877985,
KL Loss: 1.7875514458864927e-05
It 2100: Total Loss: 0.048600248992443085,      Rec Loss: 0.04859550669789314,
KL Loss: 9.487543138675392e-06
It 2200: Total Loss: 0.05162704363465309,       Rec Loss: 0.05162099003791809,
KL Loss: 1.2104574125260115e-05
It 2300: Total Loss: 0.05703907087445259,       Rec Loss: 0.05702727660536766,
KL Loss: 2.3590357159264386e-05
It 2400: Total Loss: 0.05215980485081673,       Rec Loss: 0.05215099826455116,
KL Loss: 1.7610072973184288e-05
It 2500: Total Loss: 0.05368269234895706,       Rec Loss: 0.05367407947778702,
KL Loss: 1.7228638171218336e-05
It 2600: Total Loss: 0.05114513635635376,       Rec Loss: 0.05113951116800308,
KL Loss: 1.1248019291087985e-05
It 2700: Total Loss: 0.057950567454099655,      Rec Loss: 0.05793479457497597,
KL Loss: 3.154331352561712e-05
It 2800: Total Loss: 0.05271615833044052,       Rec Loss: 0.05270550027489662,
KL Loss: 2.1317362552508712e-05
Run Epoch 3
It 2900: Total Loss: 0.04912349209189415,       Rec Loss: 0.04911353811621666,
KL Loss: 1.9909159163944423e-05
It 3000: Total Loss: 0.055225271731615067,      Rec Loss: 0.055212799459695816,
KL Loss: 2.4947061319835484e-05
It 3100: Total Loss: 0.05240027233958244,       Rec Loss: 0.05239350348711014,
KL Loss: 1.354033884126693e-05
It 3200: Total Loss: 0.05113127455115318,       Rec Loss: 0.051117926836013794,
```

```
KL Loss: 2.6696332497522235e-05
It 3300: Total Loss: 0.05302541330456734,        Rec Loss: 0.053017329424619675,
KL Loss: 1.6166333807632327e-05
It 3400: Total Loss: 0.05164191499352455,        Rec Loss: 0.0516275092959404,
KL Loss: 2.881194814108312e-05
It 3500: Total Loss: 0.04940949007868767,        Rec Loss: 0.04940174147486687,
KL Loss: 1.5495068510062993e-05
It 3600: Total Loss: 0.054981779307127,          Rec Loss: 0.05497114360332489,
KL Loss: 2.126942854374647e-05
It 3700: Total Loss: 0.055811136960983276,       Rec Loss: 0.055802229791879654,
KL Loss: 1.7817816114984453e-05
Run Epoch 4
It 3800: Total Loss: 0.0559321753680706,         Rec Loss: 0.055921126157045364,
KL Loss: 2.209783997386694e-05
It 3900: Total Loss: 0.05269641801714897,        Rec Loss: 0.05268723517656326,
KL Loss: 1.836652518250048e-05
It 4000: Total Loss: 0.05369060859084129,        Rec Loss: 0.05368517339229584,
KL Loss: 1.0868418030440807e-05
It 4100: Total Loss: 0.05872448533773422,        Rec Loss: 0.05870654806494713,
KL Loss: 3.587733954191208e-05
It 4200: Total Loss: 0.0533481203019619,         Rec Loss: 0.05333886295557022,
KL Loss: 1.8517166608944535e-05
It 4300: Total Loss: 0.053729426115751266,       Rec Loss: 0.05371948331594467,
KL Loss: 1.988727308344096e-05
It 4400: Total Loss: 0.05117770656943321,        Rec Loss: 0.05116591602563858,
KL Loss: 2.358309575356543e-05
It 4500: Total Loss: 0.05187997221946716,        Rec Loss: 0.05186071619391441,
KL Loss: 3.8511803722940385e-05
It 4600: Total Loss: 0.05556962639093399,        Rec Loss: 0.05553864315152168,
KL Loss: 6.196324829943478e-05
Done!
```

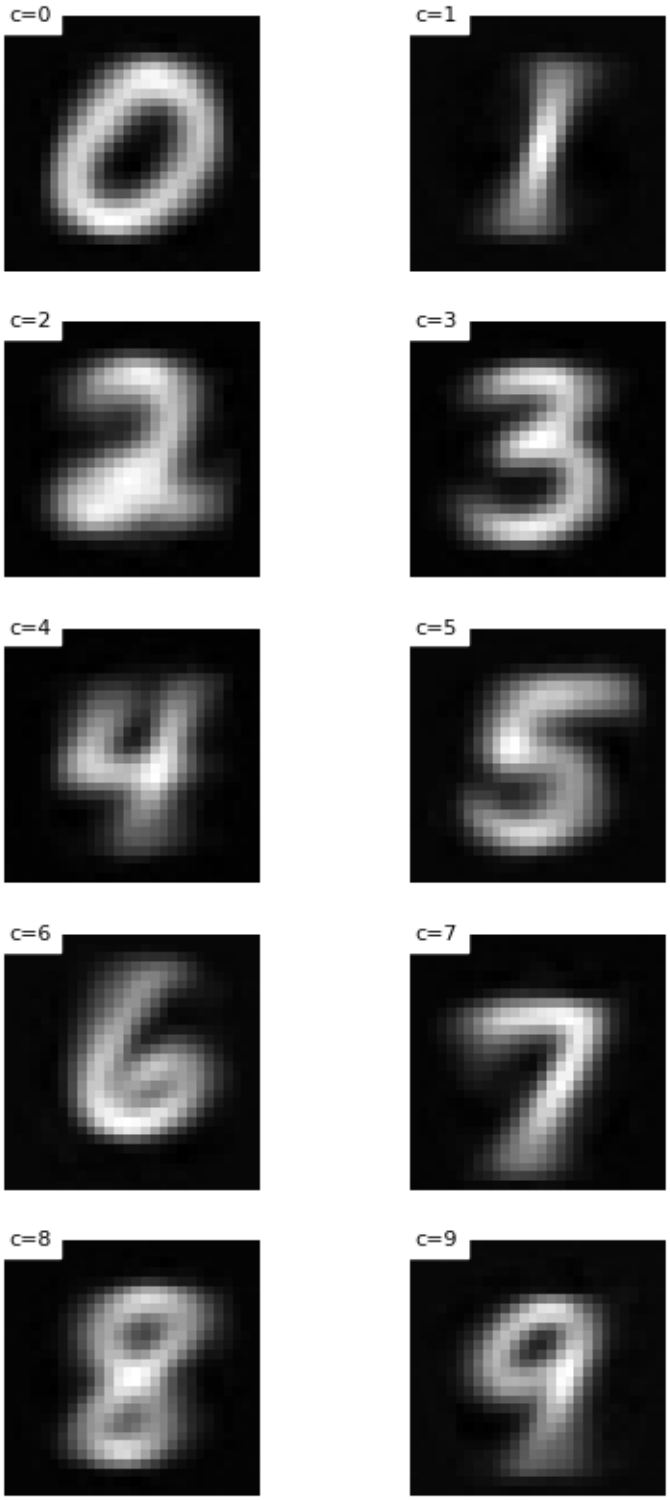


6.2.1 Verifying conditional samples from CVAE [6 pt]

Now let us generate samples from the trained model, conditioned on all the labels.

[25]:
```
# Prob1-9
if conditional:
    c = torch.arange(0, 10).long().unsqueeze(1).to(device)
    z = torch.randn([10, nz]).to(device)
    x = cvae_model.decoder(z, c=c)
else:
    z = torch.randn([10, nz]).to(device)
    x = cvae_model.decoder(z)

plt.figure()
plt.figure(figsize=(5, 10))
for p in range(10):
    plt.subplot(5, 2, p+1)
    if conditional:
        plt.text(
            0, 0, "c={:d}".format(c[p].item()), color='black',
            backgroundcolor='white', fontsize=8)
    plt.imshow(x[p].view(28, 28).cpu().data.numpy(), cmap='gray')
    plt.axis('off')
```

```
<Figure size 640x480 with 0 Axes>
```


c=0
c=1
c=2
c=3
c=4
c=5
c=6
c=7
c=8
c=9

7 Submission Instructions

You need to submit this jupyter notebook and a PDF. See Piazza for detailed submission instructions.

problem2-gan-v0

April 5, 2023

1 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.

[20]:
```
# 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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext 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.

3 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`.

[1]:
```
# Download the CIFAR-10 dataset
!curl -o data/cifar-10-python.tar.gz http://www.cs.toronto.edu/~kriz/
↪cifar-10-python.tar.gz

# Extract the dataset
!tar -xzvf data/cifar-10-python.tar.gz -C data

# Remove the downloaded archive
!del data/cifar-10-python.tar.gz
```

```
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

 0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
 0  162M    0  840k    0     0   888k      0  0:03:07 --:--:--  0:03:07  887k
 1  162M    1 2661k    0     0  1341k      0  0:02:04  0:00:01  0:02:03 1341k
 3  162M    3 6278k    0     0  2133k      0  0:01:18  0:00:02  0:01:16 2133k
 5  162M    5 8575k    0     0  2104k      0  0:01:19  0:00:04  0:01:15 2105k
 6  162M    6 10.5M    0     0  2172k      0  0:01:16  0:00:04  0:01:12 2172k
 7  162M    7 12.7M    0     0  2195k      0  0:01:15  0:00:05  0:01:10 2442k
 9  162M    9 15.1M    0     0  2232k      0  0:01:14  0:00:06  0:01:08 2587k
10  162M   10 17.3M    0     0  2233k      0  0:01:14  0:00:07  0:01:07 2292k
12  162M   12 19.7M    0     0  2260k      0  0:01:13  0:00:08  0:01:05 2390k
13  162M   13 21.8M    0     0  2243k      0  0:01:14  0:00:09  0:01:05 2314k
14  162M   14 23.7M    0     0  2223k      0  0:01:14  0:00:10  0:01:04 2256k
15  162M   15 25.1M    0     0  2153k      0  0:01:17  0:00:11  0:01:06 2045k
16  162M   16 26.1M    0     0  2065k      0  0:01:20  0:00:12  0:01:08 1799k
16  162M   16 27.2M    0     0  1998k      0  0:01:23  0:00:13  0:01:10 1533k
17  162M   17 28.5M    0     0  1959k      0  0:01:24  0:00:14  0:01:10 1390k
18  162M   18 29.6M    0     0  1907k      0  0:01:27  0:00:15  0:01:12 1216k
19  162M   19 30.9M    0     0  1867k      0  0:01:29  0:00:16  0:01:13 1183k
19  162M   19 32.1M    0     0  1831k      0  0:01:30  0:00:17  0:01:13 1227k
20  162M   20 33.3M    0     0  1802k      0  0:01:32  0:00:18  0:01:14 1252k
21  162M   21 34.5M    0     0  1772k      0  0:01:33  0:00:19  0:01:14 1215k
21  162M   21 35.6M    0     0  1741k      0  0:01:35  0:00:20  0:01:15 1211k
22  162M   22 36.9M    0     0  1723k      0  0:01:36  0:00:21  0:01:15 1234k
23  162M   23 37.8M    0     0  1687k      0  0:01:38  0:00:22  0:01:16 1167k
23  162M   23 38.8M    0     0  1659k      0  0:01:40  0:00:23  0:01:17 1114k
24  162M   24 39.9M    0     0  1639k      0  0:01:41  0:00:24  0:01:17 1107k
25  162M   25 41.1M    0     0  1624k      0  0:01:42  0:00:25  0:01:17 1133k
26  162M   26 42.3M    0     0  1610k      0  0:01:43  0:00:26  0:01:17 1115k
26  162M   26 43.7M    0     0  1604k      0  0:01:43  0:00:27  0:01:16 1224k
27  162M   27 45.2M    0     0  1602k      0  0:01:43  0:00:28  0:01:15 1331k
28  162M   28 46.6M    0     0  1590k      0  0:01:44  0:00:30  0:01:14 1347k
29  162M   29 47.6M    0     0  1577k      0  0:01:45  0:00:30  0:01:15 1333k
30  162M   30 48.7M    0     0  1564k      0  0:01:46  0:00:31  0:01:15 1314k
30  162M   30 50.0M    0     0  1555k      0  0:01:47  0:00:32  0:01:15 1279k
31  162M   31 51.4M    0     0  1552k      0  0:01:47  0:00:33  0:01:14 1262k
32  162M   32 53.0M    0     0  1554k      0  0:01:47  0:00:34  0:01:13 1337k
33  162M   33 54.6M    0     0  1553k      0  0:01:47  0:00:36  0:01:11 1411k
34  162M   34 55.8M    0     0  1546k      0  0:01:47  0:00:36  0:01:11 1435k
35  162M   35 57.0M    0     0  1538k      0  0:01:48  0:00:37  0:01:11 1427k
35  162M   35 58.4M    0     0  1537k      0  0:01:48  0:00:38  0:01:10 1432k
36  162M   36 60.0M    0     0  1538k      0  0:01:48  0:00:39  0:01:09 1423k
37  162M   37 61.5M    0     0  1538k      0  0:01:48  0:00:40  0:01:08 1427k
38  162M   38 62.9M    0     0  1536k      0  0:01:48  0:00:41  0:01:07 1463k
39  162M   39 64.4M    0     0  1535k      0  0:01:48  0:00:42  0:01:06 1512k
40  162M   40 65.9M    0     0  1537k      0  0:01:48  0:00:43  0:01:05 1539k
```

41	162M	41	67.5M	0	0	1538k	0	0:01:48	0:00:44	0:01:04	1543k
42	162M	42	68.9M	0	0	1536k	0	0:01:48	0:00:45	0:01:03	1521k
43	162M	43	70.7M	0	0	1543k	0	0:01:47	0:00:46	0:01:01	1594k
45	162M	45	73.3M	0	0	1567k	0	0:01:46	0:00:47	0:00:59	1842k
46	162M	46	75.7M	0	0	1585k	0	0:01:45	0:00:48	0:00:57	2011k
48	162M	48	78.0M	0	0	1600k	0	0:01:44	0:00:49	0:00:55	2157k
49	162M	49	80.1M	0	0	1610k	0	0:01:43	0:00:50	0:00:53	2294k
50	162M	50	82.1M	0	0	1608k	0	0:01:43	0:00:52	0:00:51	2191k
51	162M	51	83.8M	0	0	1622k	0	0:01:42	0:00:52	0:00:50	2150k
52	162M	52	85.4M	0	0	1620k	0	0:01:42	0:00:53	0:00:49	1964k
53	162M	53	87.5M	0	0	1631k	0	0:01:42	0:00:54	0:00:48	1939k
55	162M	55	89.4M	0	0	1637k	0	0:01:41	0:00:55	0:00:46	1910k
56	162M	56	91.3M	0	0	1642k	0	0:01:41	0:00:56	0:00:45	2022k
57	162M	57	93.2M	0	0	1647k	0	0:01:41	0:00:57	0:00:44	1913k
58	162M	58	95.2M	0	0	1654k	0	0:01:40	0:00:58	0:00:42	2022k
59	162M	59	97.2M	0	0	1660k	0	0:01:40	0:00:59	0:00:41	1981k
60	162M	60	99.1M	0	0	1665k	0	0:01:39	0:01:00	0:00:39	1979k
62	162M	62	101M	0	0	1674k	0	0:01:39	0:01:01	0:00:38	2030k
63	162M	63	103M	0	0	1684k	0	0:01:38	0:01:03	0:00:35	2104k
64	162M	64	105M	0	0	1685k	0	0:01:38	0:01:03	0:00:35	2048k
65	162M	65	106M	0	0	1676k	0	0:01:39	0:01:04	0:00:35	1860k
66	162M	66	107M	0	0	1666k	0	0:01:39	0:01:05	0:00:34	1680k
67	162M	67	109M	0	0	1667k	0	0:01:39	0:01:06	0:00:33	1588k
67	162M	67	110M	0	0	1666k	0	0:01:39	0:01:07	0:00:32	1430k
68	162M	68	111M	0	0	1661k	0	0:01:40	0:01:08	0:00:32	1349k
69	162M	69	113M	0	0	1661k	0	0:01:40	0:01:09	0:00:31	1469k
70	162M	70	114M	0	0	1653k	0	0:01:40	0:01:10	0:00:30	1484k
71	162M	71	115M	0	0	1651k	0	0:01:40	0:01:11	0:00:29	1425k
72	162M	72	117M	0	0	1651k	0	0:01:40	0:01:12	0:00:28	1445k
73	162M	73	119M	0	0	1646k	0	0:01:41	0:01:14	0:00:27	1450k
74	162M	74	120M	0	0	1644k	0	0:01:41	0:01:14	0:00:27	1408k
75	162M	75	121M	0	0	1644k	0	0:01:41	0:01:15	0:00:26	1515k
76	162M	76	123M	0	0	1645k	0	0:01:41	0:01:16	0:00:25	1571k
77	162M	77	125M	0	0	1645k	0	0:01:41	0:01:17	0:00:24	1561k
77	162M	77	126M	0	0	1643k	0	0:01:41	0:01:18	0:00:23	1595k
79	162M	79	128M	0	0	1645k	0	0:01:41	0:01:19	0:00:22	1652k
80	162M	80	130M	0	0	1649k	0	0:01:40	0:01:20	0:00:20	1720k
81	162M	81	132M	0	0	1649k	0	0:01:40	0:01:21	0:00:19	1708k
82	162M	82	133M	0	0	1649k	0	0:01:40	0:01:22	0:00:18	1718k
82	162M	82	134M	0	0	1645k	0	0:01:41	0:01:23	0:00:18	1671k
83	162M	83	136M	0	0	1642k	0	0:01:41	0:01:24	0:00:17	1603k
84	162M	84	137M	0	0	1638k	0	0:01:41	0:01:25	0:00:16	1466k
85	162M	85	138M	0	0	1633k	0	0:01:41	0:01:26	0:00:15	1363k
85	162M	85	139M	0	0	1627k	0	0:01:42	0:01:27	0:00:15	1266k
86	162M	86	140M	0	0	1620k	0	0:01:42	0:01:28	0:00:14	1202k
87	162M	87	141M	0	0	1613k	0	0:01:43	0:01:29	0:00:14	1123k
87	162M	87	143M	0	0	1610k	0	0:01:43	0:01:30	0:00:13	1118k
88	162M	88	143M	0	0	1602k	0	0:01:43	0:01:31	0:00:12	1073k

```
 89  162M   89  144M    0     0  1597k      0  0:01:44  0:01:32  0:00:12 1061k
 89  162M   89  145M    0     0  1587k      0  0:01:44  0:01:33  0:00:11 1010k
 90  162M   90  146M    0     0  1580k      0  0:01:45  0:01:34  0:00:11  982k
 90  162M   90  147M    0     0  1574k      0  0:01:45  0:01:35  0:00:10  926k
 91  162M   91  148M    0     0  1568k      0  0:01:46  0:01:36  0:00:10  946k
 91  162M   91  149M    0     0  1561k      0  0:01:46  0:01:37  0:00:09  896k
 92  162M   92  150M    0     0  1554k      0  0:01:47  0:01:38  0:00:09  926k
 92  162M   92  151M    0     0  1547k      0  0:01:47  0:01:39  0:00:08  924k
 93  162M   93  152M    0     0  1543k      0  0:01:47  0:01:40  0:00:07  938k
 94  162M   94  153M    0     0  1537k      0  0:01:48  0:01:41  0:00:07  935k
 94  162M   94  154M    0     0  1534k      0  0:01:48  0:01:42  0:00:06  993k
 95  162M   95  155M    0     0  1532k      0  0:01:48  0:01:43  0:00:05 1091k
 96  162M   96  156M    0     0  1525k      0  0:01:49  0:01:44  0:00:05 1072k
 96  162M   96  157M    0     0  1518k      0  0:01:49  0:01:45  0:00:04 1021k
 97  162M   97  158M    0     0  1513k      0  0:01:50  0:01:46  0:00:04 1012k
 97  162M   97  159M    0     0  1509k      0  0:01:50  0:01:47  0:00:03  995k
 98  162M   98  159M    0     0  1503k      0  0:01:50  0:01:48  0:00:02  912k
 99  162M   99  161M    0     0  1500k      0  0:01:50  0:01:49  0:00:01  982k
 99  162M   99  162M    0     0  1497k      0  0:01:51  0:01:50  0:00:01 1057k
100  162M  100  162M    0     0  1496k      0  0:01:51  0:01:51 --:--:-- 1085k
x cifar-10-batches-py/
x cifar-10-batches-py/data_batch_4
x cifar-10-batches-py/readme.html
x cifar-10-batches-py/test_batch
x cifar-10-batches-py/data_batch_3
x cifar-10-batches-py/batches.meta
x cifar-10-batches-py/data_batch_2
x cifar-10-batches-py/data_batch_5
x cifar-10-batches-py/data_batch_1
Invalid switch - "cifar-10-python.tar.gz".
```

[61]:
```python
def unpickle(file):
    import sys
    if sys.version_info.major == 2:
        import cPickle
        with open(file, 'rb') as fo:
            dict = cPickle.load(fo)
        return dict['data'], dict['labels']
    else:
        import pickle
        with open(file, 'rb') as fo:
            dict = pickle.load(fo, encoding='bytes')
        return dict[b'data'], dict[b'labels']
def load_train_data():
    X = []
    for i in range(5):
        X_, _ = unpickle('data/cifar-10-batches-py/data_batch_%d' % (i + 1))
```

```
        X.append(X_)
    X = np.concatenate(X)
    X = X.reshape((X.shape[0], 3, 32, 32))
    return X

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

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

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

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

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

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

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

[62]:
```
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
```

[63]:
```
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
```

[70]:
```
# Use this to put tensors on GPU/CPU automatically when defining tensors
from torch.optim import Adam

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print("device is: ", device)
class DCGAN(nn.Module):
    def __init__(self):
        super(DCGAN, self).__init__()
        self.num_epoch = 25
        self.batch_size = 128
        self.log_step = 100
        self.visualize_step = 2
        self.code_size = 64  # size of latent vector (size of generator input)
        self.learning_rate = 2e-4
```

```
        self.vis_learning_rate = 1e-2


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

        self._actmax_label = torch.ones([64, 1], device=device)

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


        # To-Do: Apply weight initialization (first implement the weight␣
↪initialization
        # 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
        self.generator_optim = torch.optim.Adam( params = self._generator.
↪parameters(), lr = self.learning_rate, betas = betas)
```

```
        self.discriminator_optim = torch.optim.Adam(params = 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, 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␣
↪binary cross entropy loss)
    def _classification_loss(self, y_pred, y_label):
      sig = nn.Sigmoid()
      return nn.BCELoss()(sig(y_pred), y_label)



        ␣
↪##############################################################################
        #                                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)
               ␣
↪##############################################################################
                # 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_optim.zero_grad()

                # To-Do: feed real samples to the discriminator
                dis_output = 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(dis_output,␣
↪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
                noise = torch.randn([self.batch_size, self.code_size])

                # To-Do: generate fake samples from the noise using the␣
↪generator
                gen_noise_output  = self._generator(noise)

                # 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
                dis_fake = self._discriminator(gen_noise_output.detach())

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


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

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

                # To-Do: generate fake samples from the noise using the␣
↪generator
                fake_noise_img = self._generator(noise)

                # 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_gen_fake = self._discriminator(fake_noise_img)

                # 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_gen_fake, torch.
↪ones(self.batch_size, 1))

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

                # To-Do: Update the generator weights
                self.generator_optim.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, input, target):
      return nn.MSELoss()(input, target)




    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.recon_opt = 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.recon_opt.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.recon_opt.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()
```

```
device is:  cpu
```

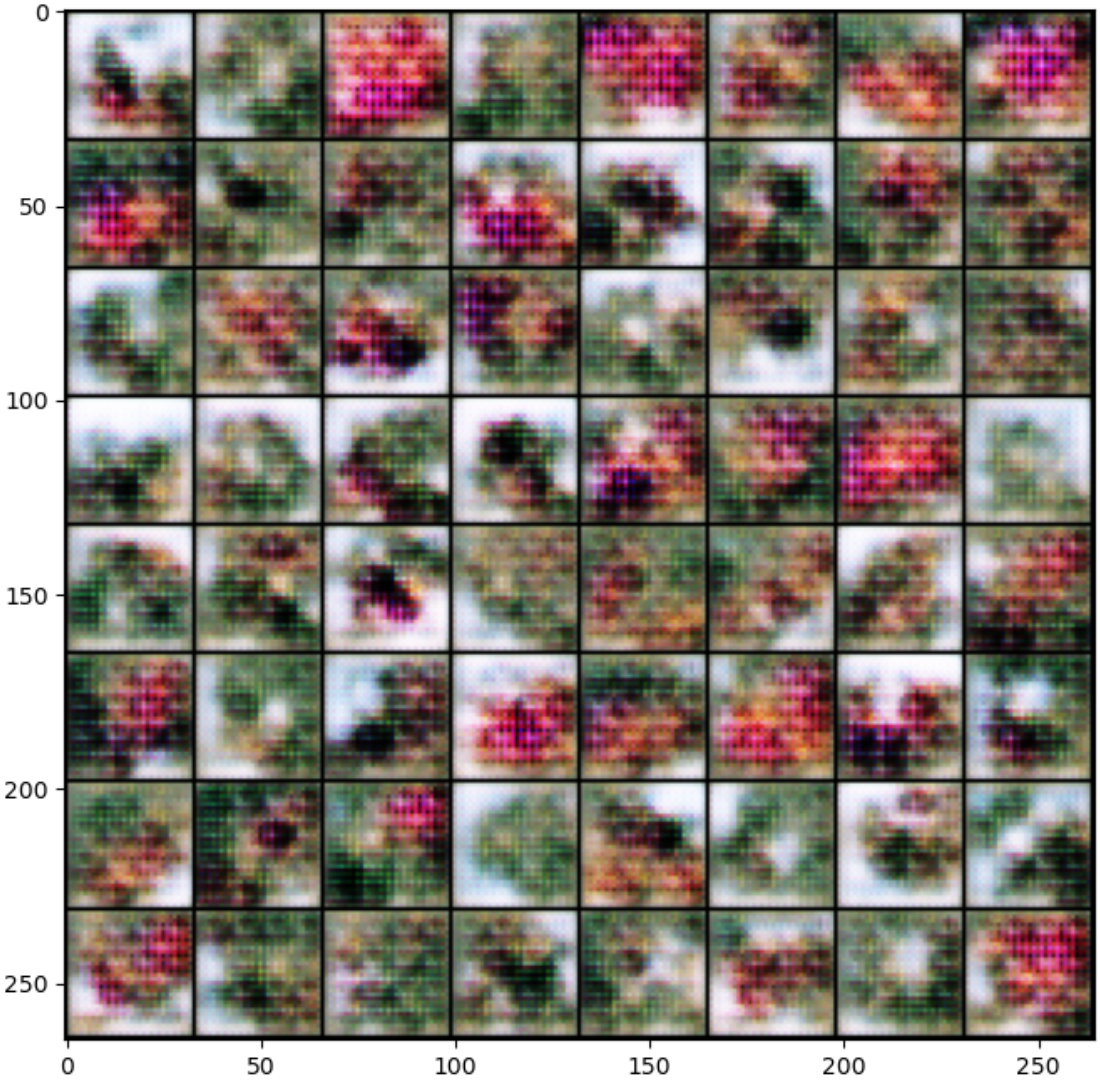
Now let's do the training!

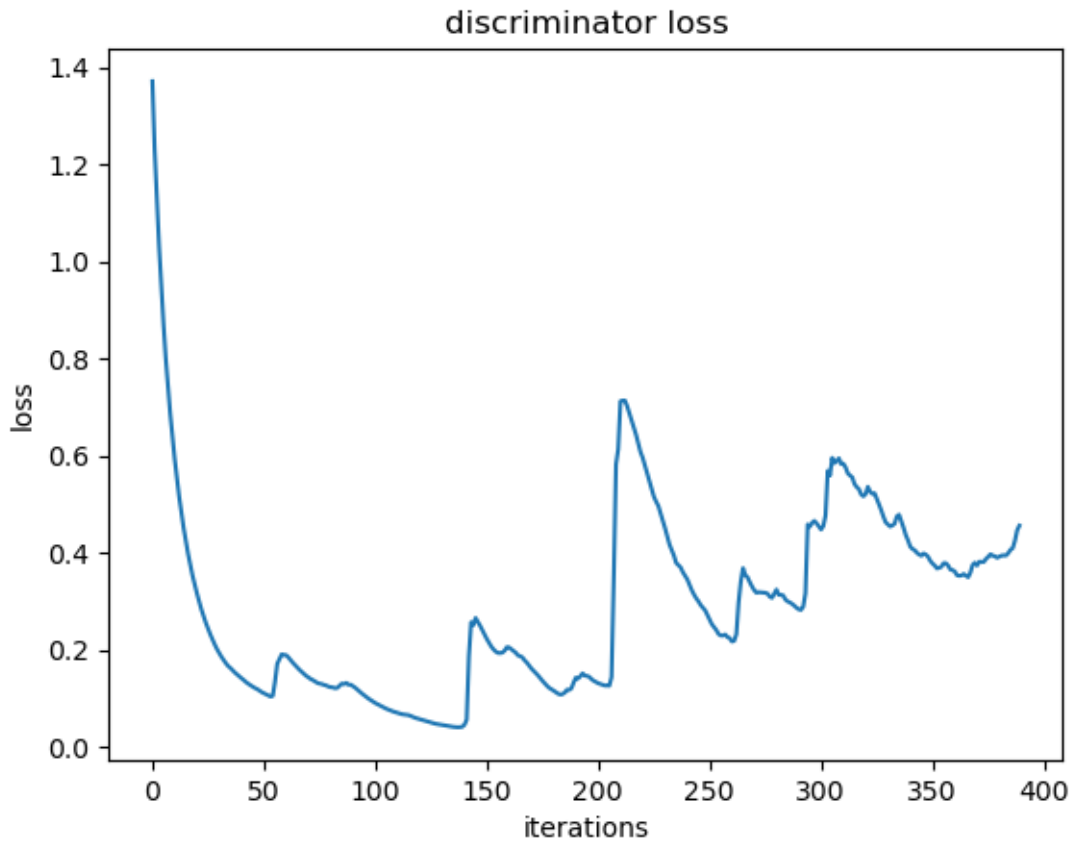
Don't panic if the loss curve goes wild. The two networks are competing for the loss curve to go different directions, so virtually anything can happen. If your code is correct, the generated samples should have a high variety.

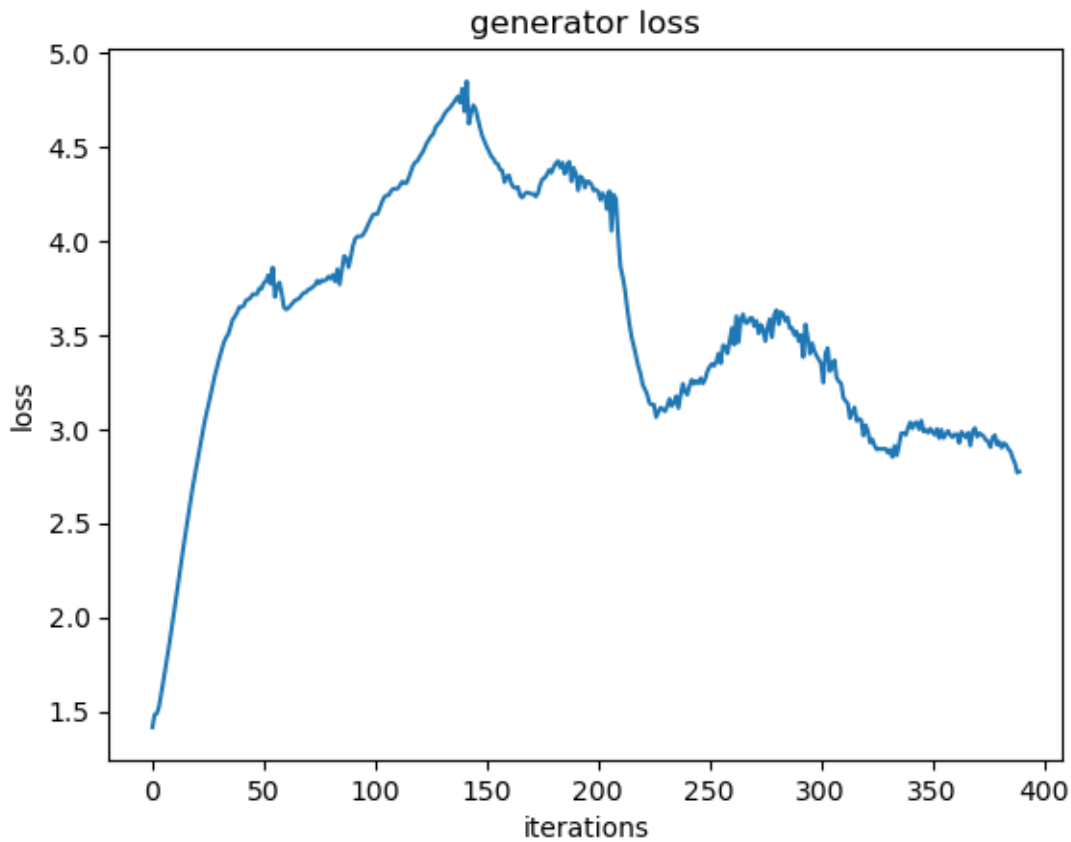
Do NOT change the number of epochs, learning rate, or batch size. If you're using Google Colab, the batch size will not be an issue during training.

[65]:
```
set_seed(42)

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

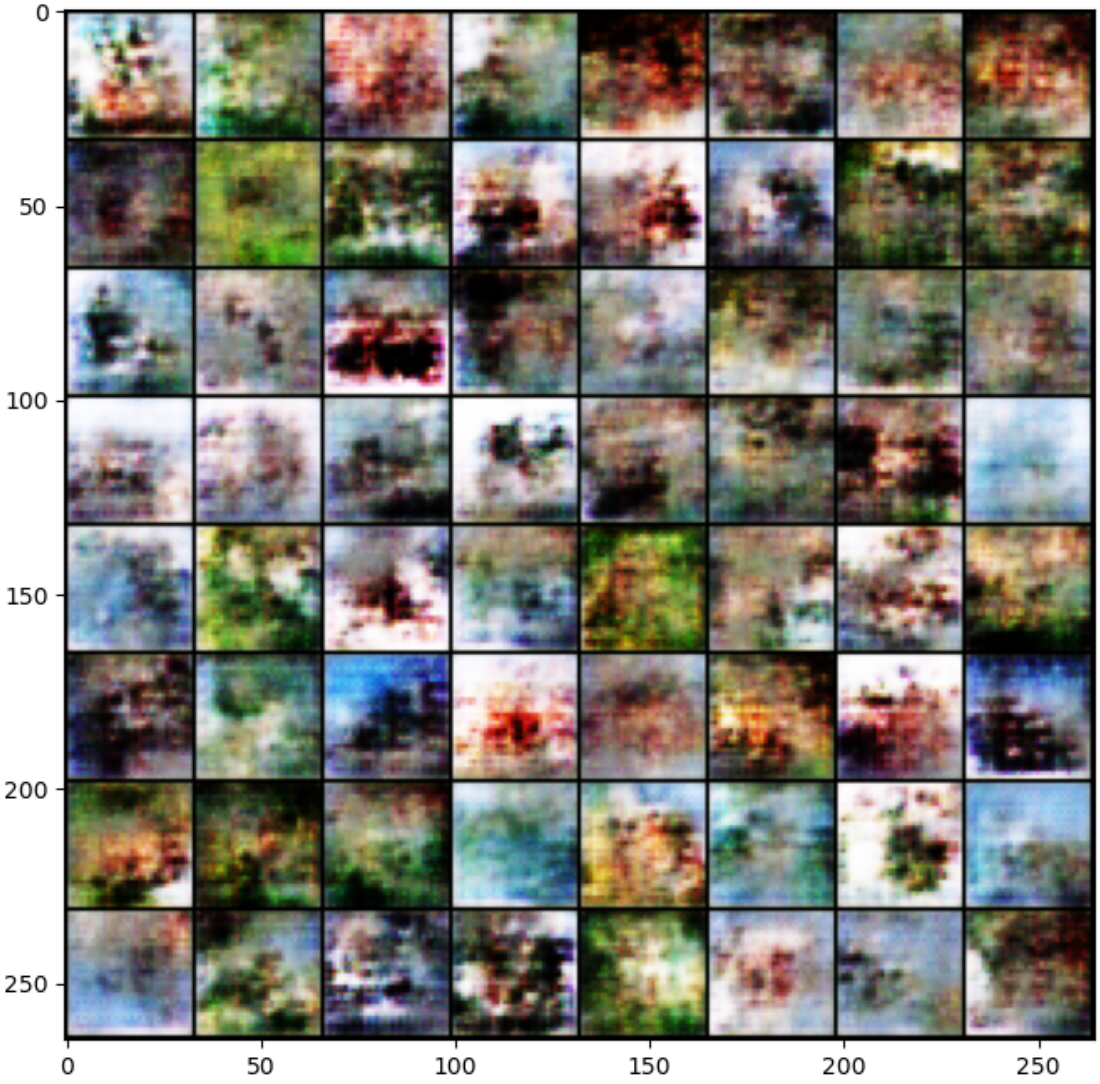
```
Start training …
Iteration 100/9750: dis loss = 0.0318, gen loss = 4.5859
Iteration 200/9750: dis loss = 0.0873, gen loss = 4.3575
Iteration 300/9750: dis loss = 0.3397, gen loss = 2.9726
```

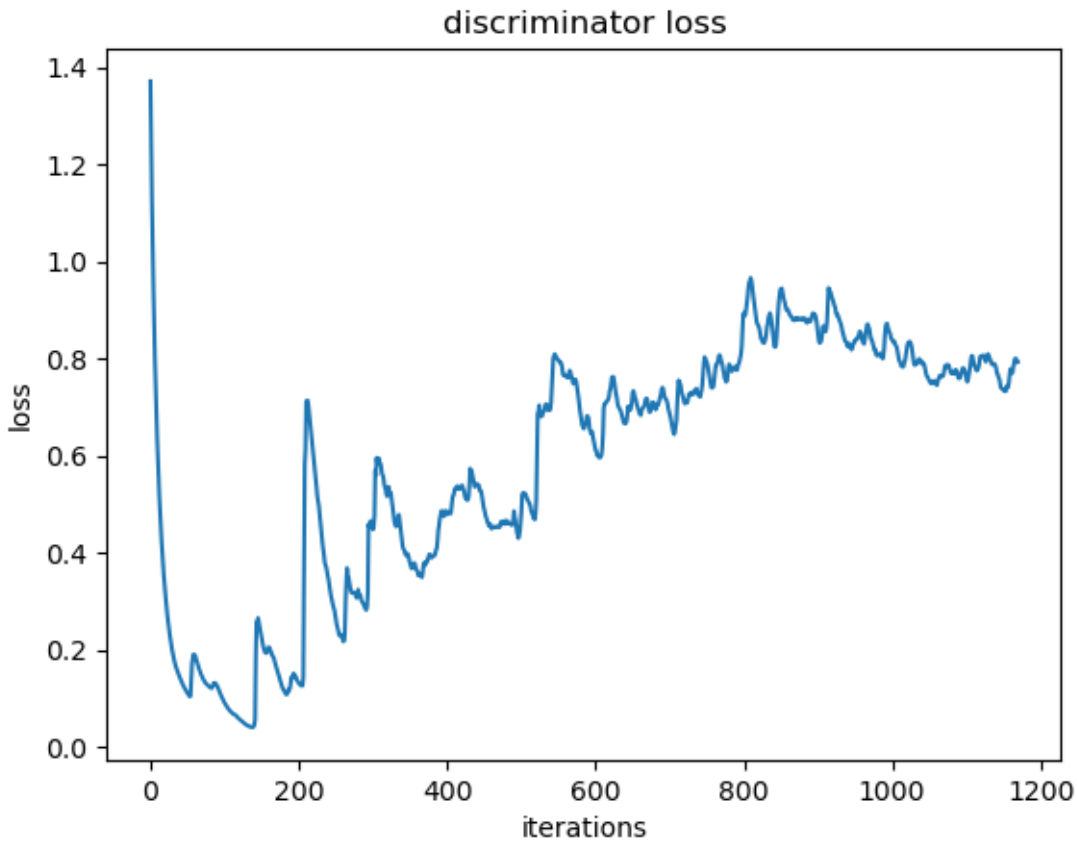

0
50
100
150
200
250
0
50
100
150
200
250

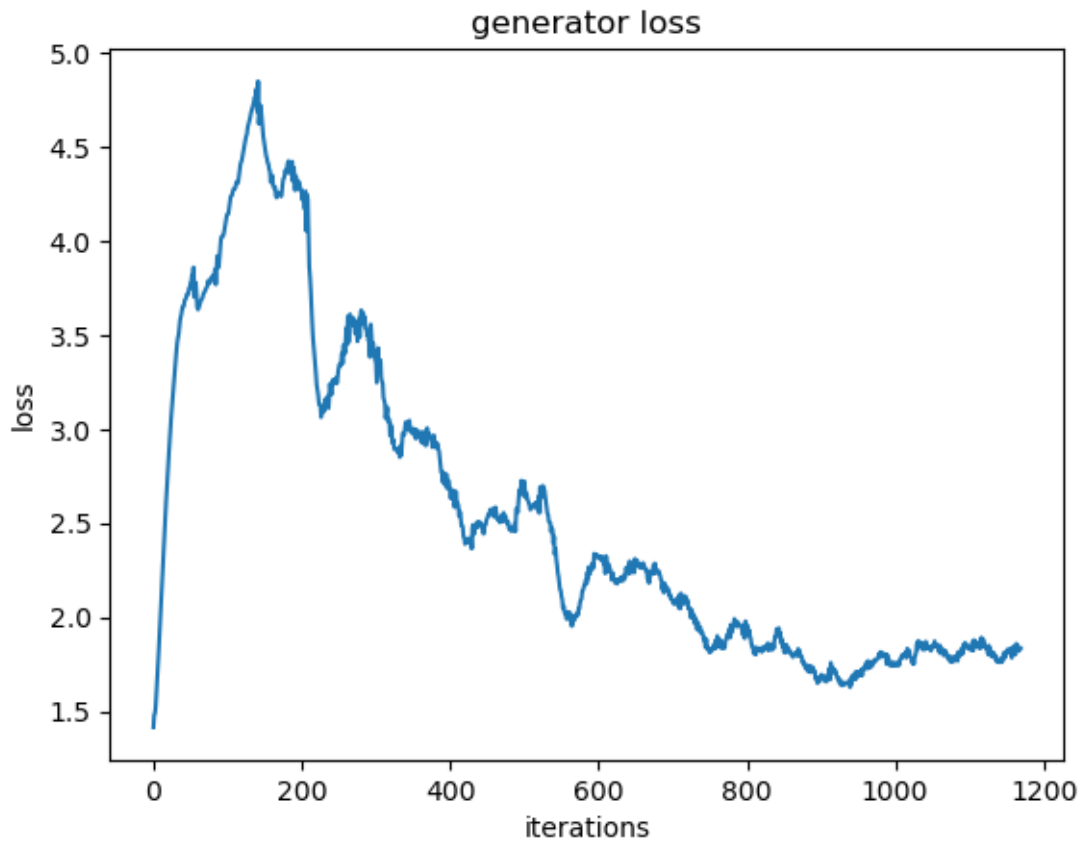

discriminator loss
loss
iterations
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300
350
400




```
Iteration 400/9750: dis loss = 0.3410, gen loss = 2.0854
Iteration 500/9750: dis loss = 0.8352, gen loss = 1.2073
Iteration 600/9750: dis loss = 0.4726, gen loss = 2.2996
Iteration 700/9750: dis loss = 0.5833, gen loss = 1.8358
Iteration 800/9750: dis loss = 1.5961, gen loss = 0.9965
Iteration 900/9750: dis loss = 0.6063, gen loss = 2.1032
Iteration 1000/9750: dis loss = 0.7901, gen loss = 1.4134
Iteration 1100/9750: dis loss = 0.6406, gen loss = 1.5465
```

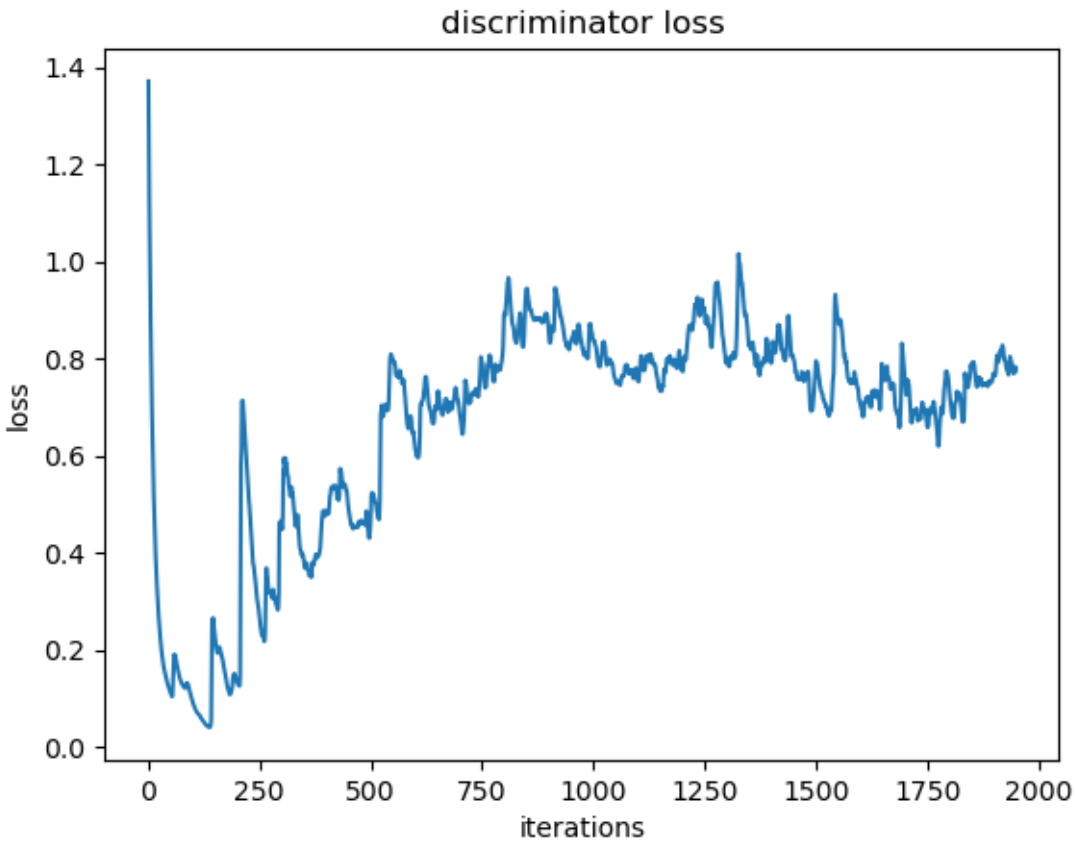

0
50
100
150
200
250
0
50
100
150
200
250

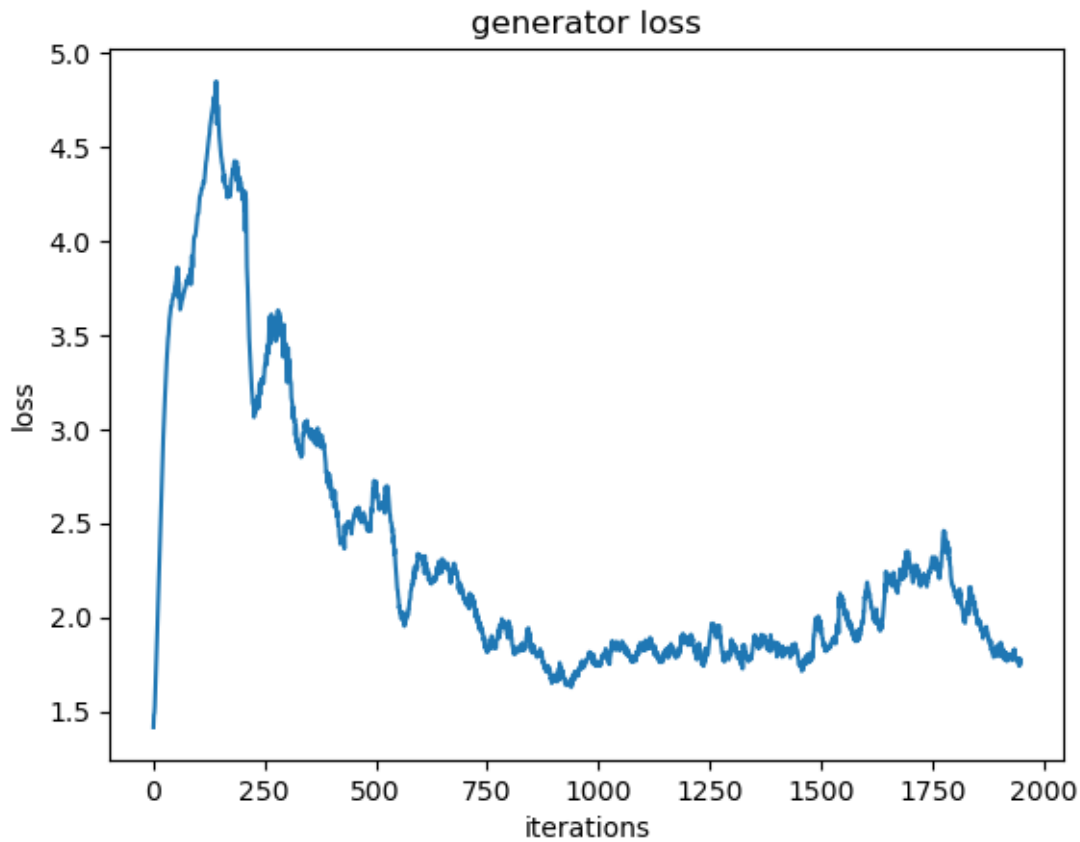

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
200
400
600
800
1000
1200
iterations



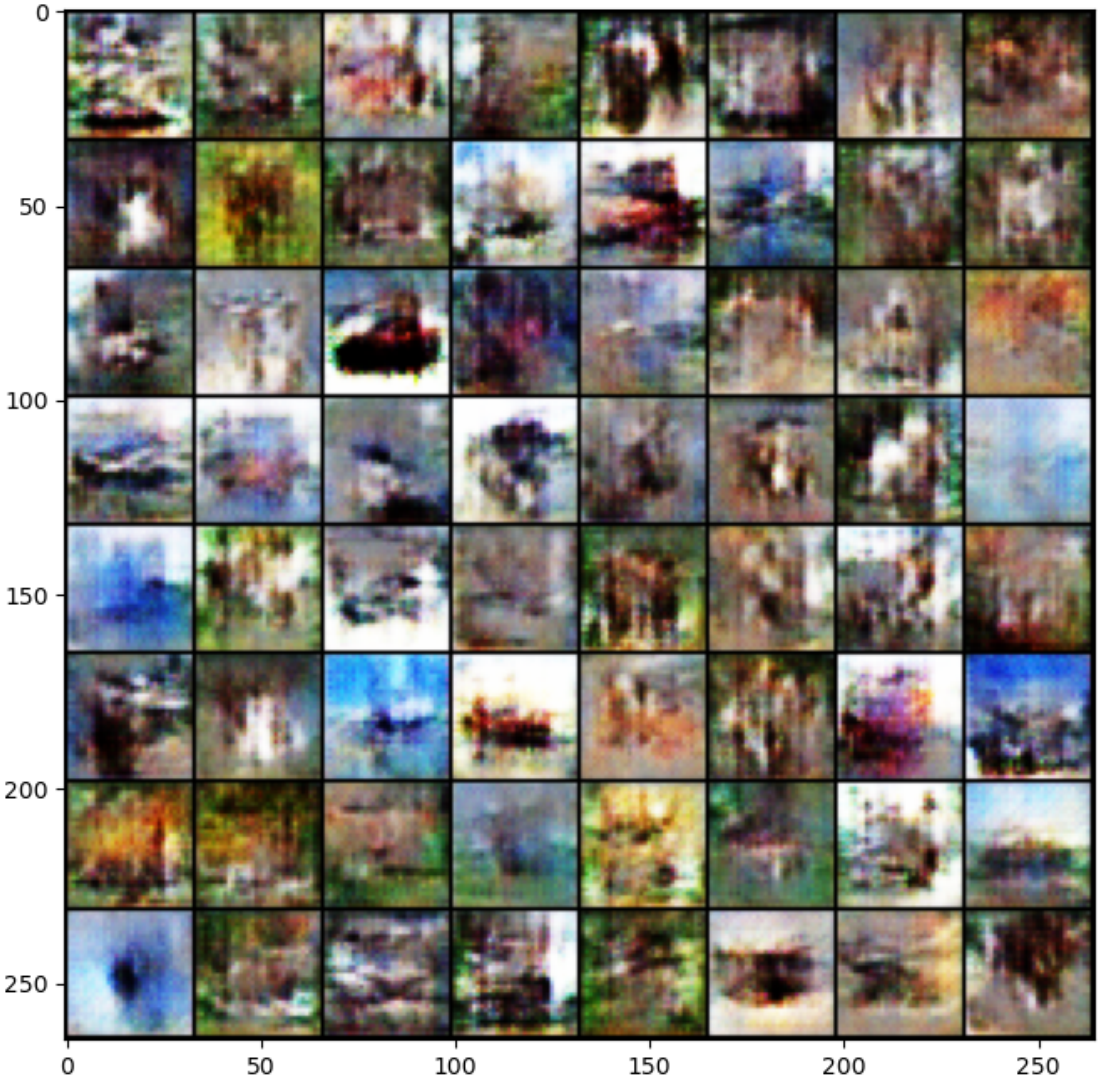

```
Iteration 1200/9750: dis loss = 0.6996, gen loss = 1.6632
Iteration 1300/9750: dis loss = 0.8521, gen loss = 3.4333
Iteration 1400/9750: dis loss = 0.7389, gen loss = 1.5057
Iteration 1500/9750: dis loss = 1.0240, gen loss = 3.0135
Iteration 1600/9750: dis loss = 0.6251, gen loss = 1.9641
Iteration 1700/9750: dis loss = 0.5913, gen loss = 1.8283
Iteration 1800/9750: dis loss = 0.5052, gen loss = 1.8411
Iteration 1900/9750: dis loss = 0.8523, gen loss = 2.0854
```

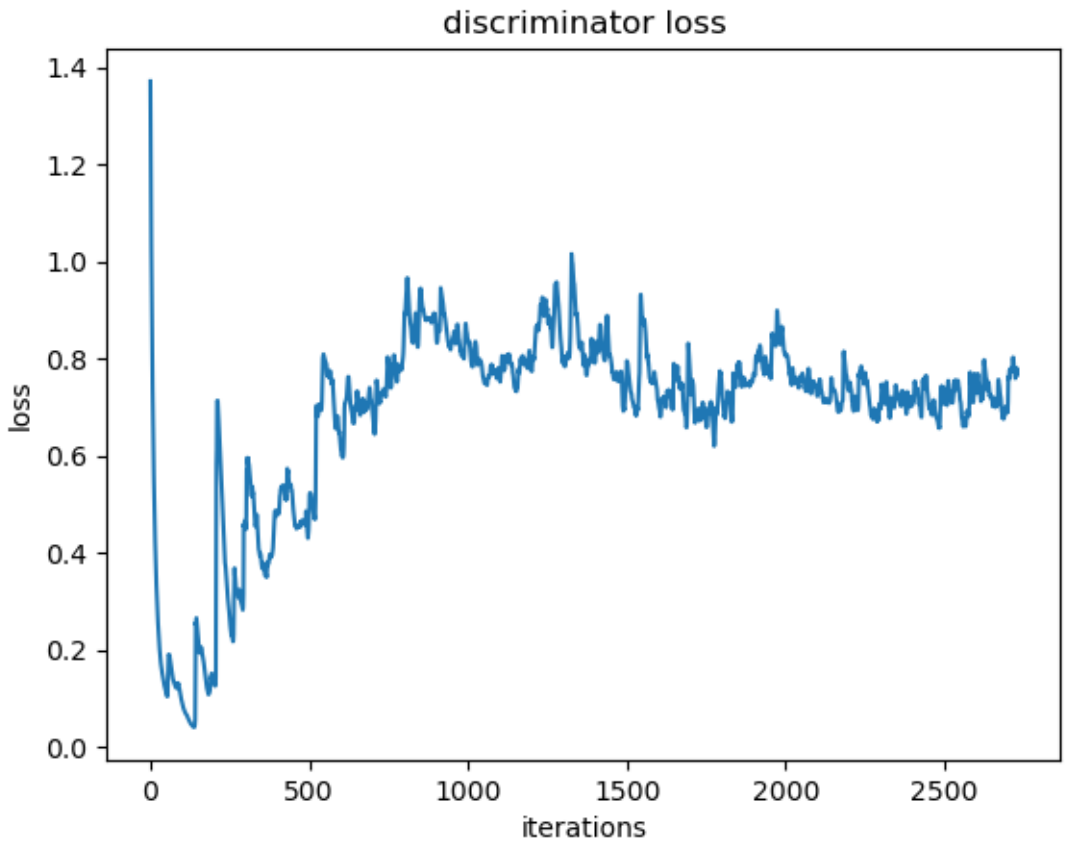
0
50
100
150
200
250
0
50
100
150
200
250

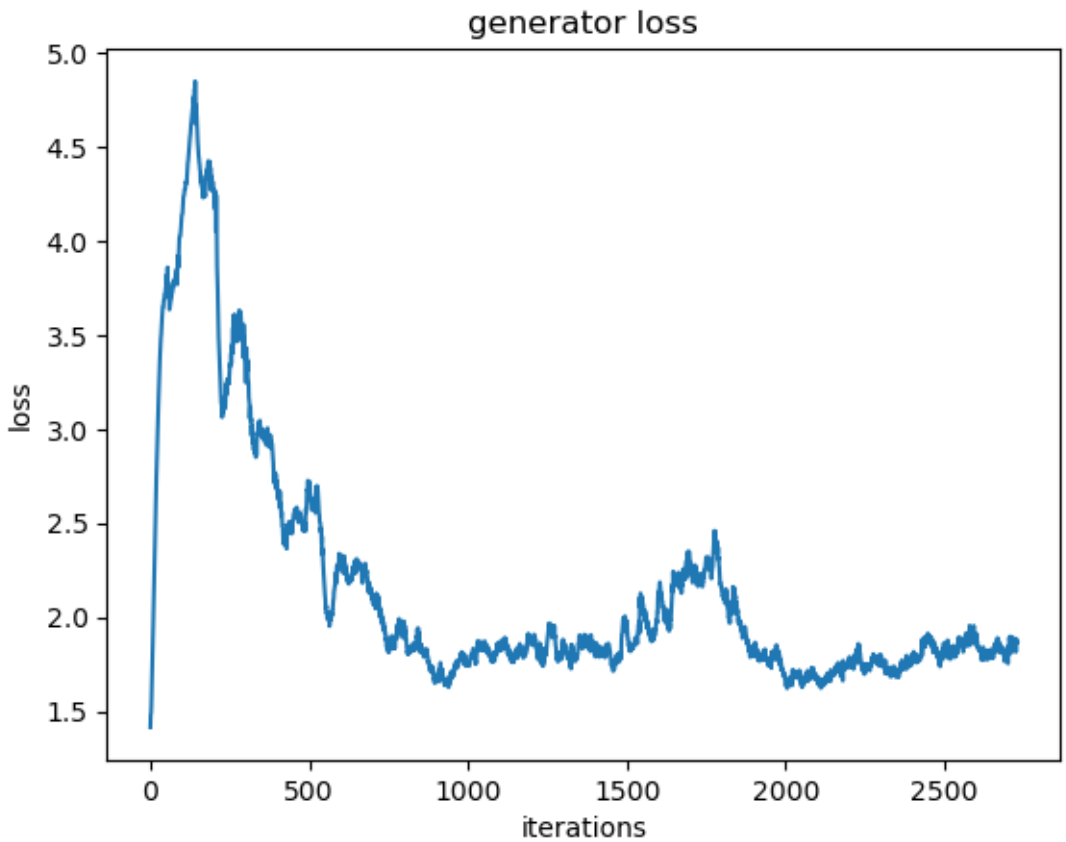

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
250
500
750
1000
1250
1500
1750
2000
iterations



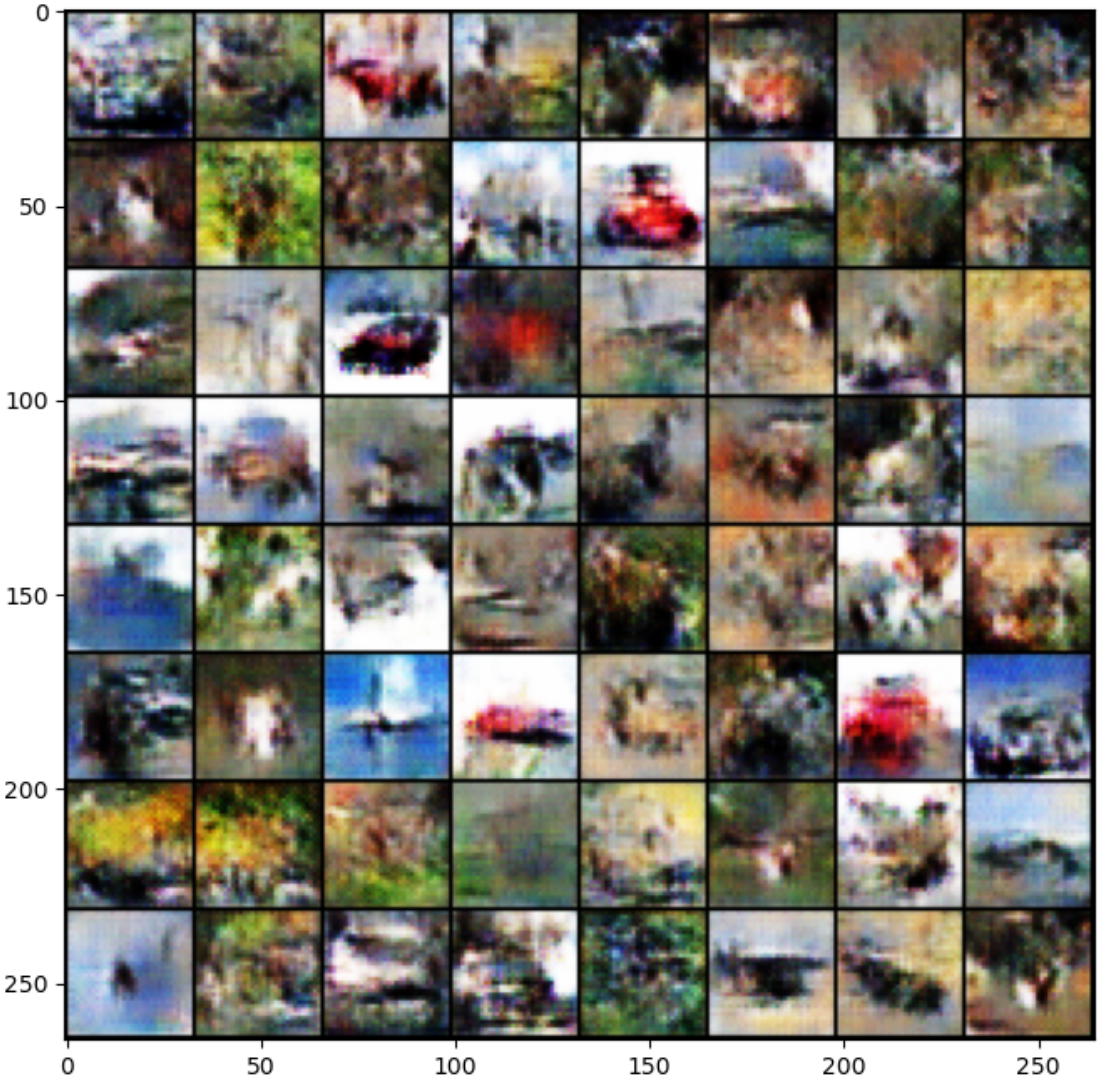

```
Iteration 2000/9750: dis loss = 0.7338, gen loss = 1.3751
Iteration 2100/9750: dis loss = 0.6328, gen loss = 1.6281
Iteration 2200/9750: dis loss = 0.7220, gen loss = 2.2219
Iteration 2300/9750: dis loss = 0.6327, gen loss = 2.5338
Iteration 2400/9750: dis loss = 0.8541, gen loss = 1.2814
Iteration 2500/9750: dis loss = 0.9227, gen loss = 1.3831
Iteration 2600/9750: dis loss = 0.5318, gen loss = 1.7470
Iteration 2700/9750: dis loss = 0.7646, gen loss = 2.9588
```

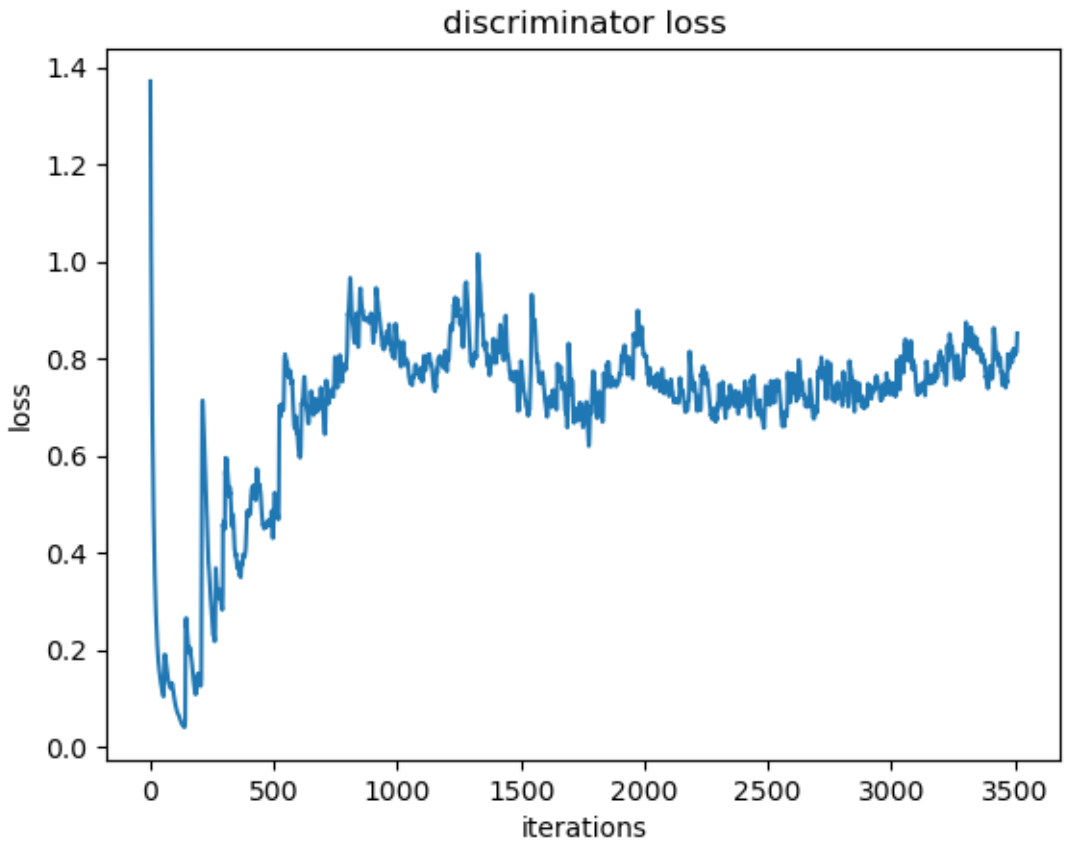

0
50
100
150
200
250
0
50
100
150
200
250

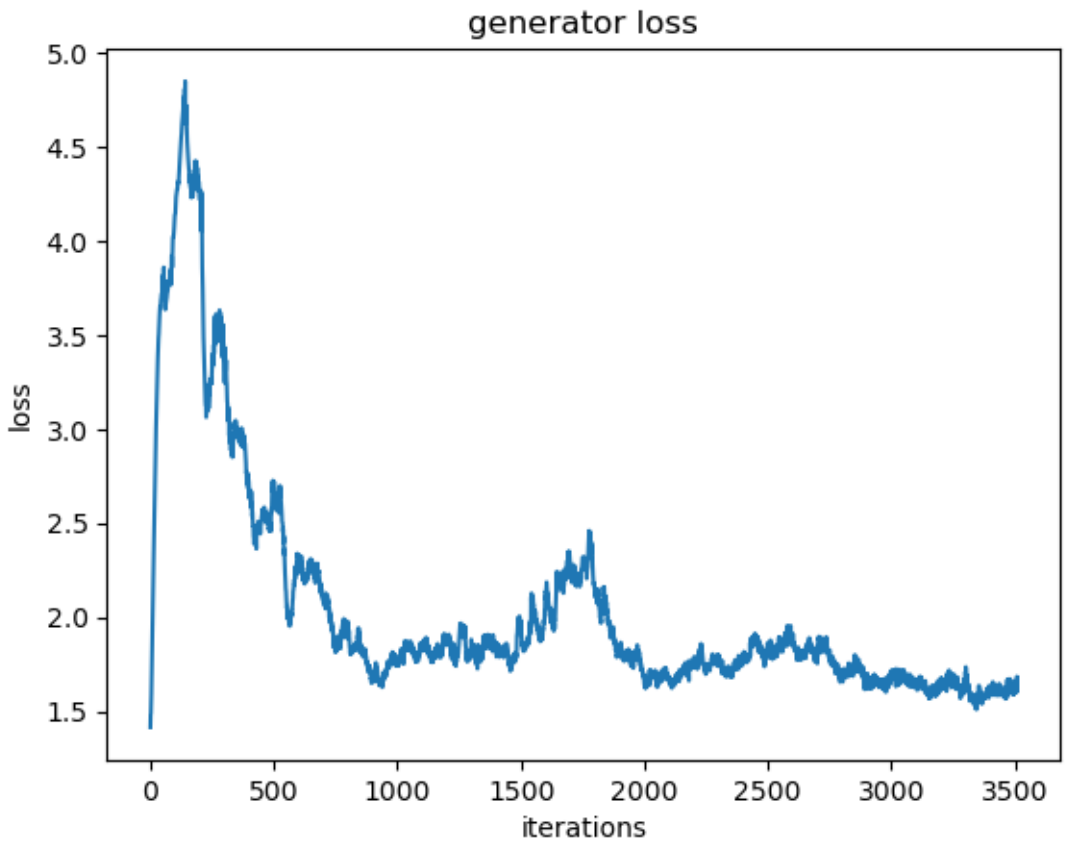

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
500
1000
1500
2000
2500
iterations



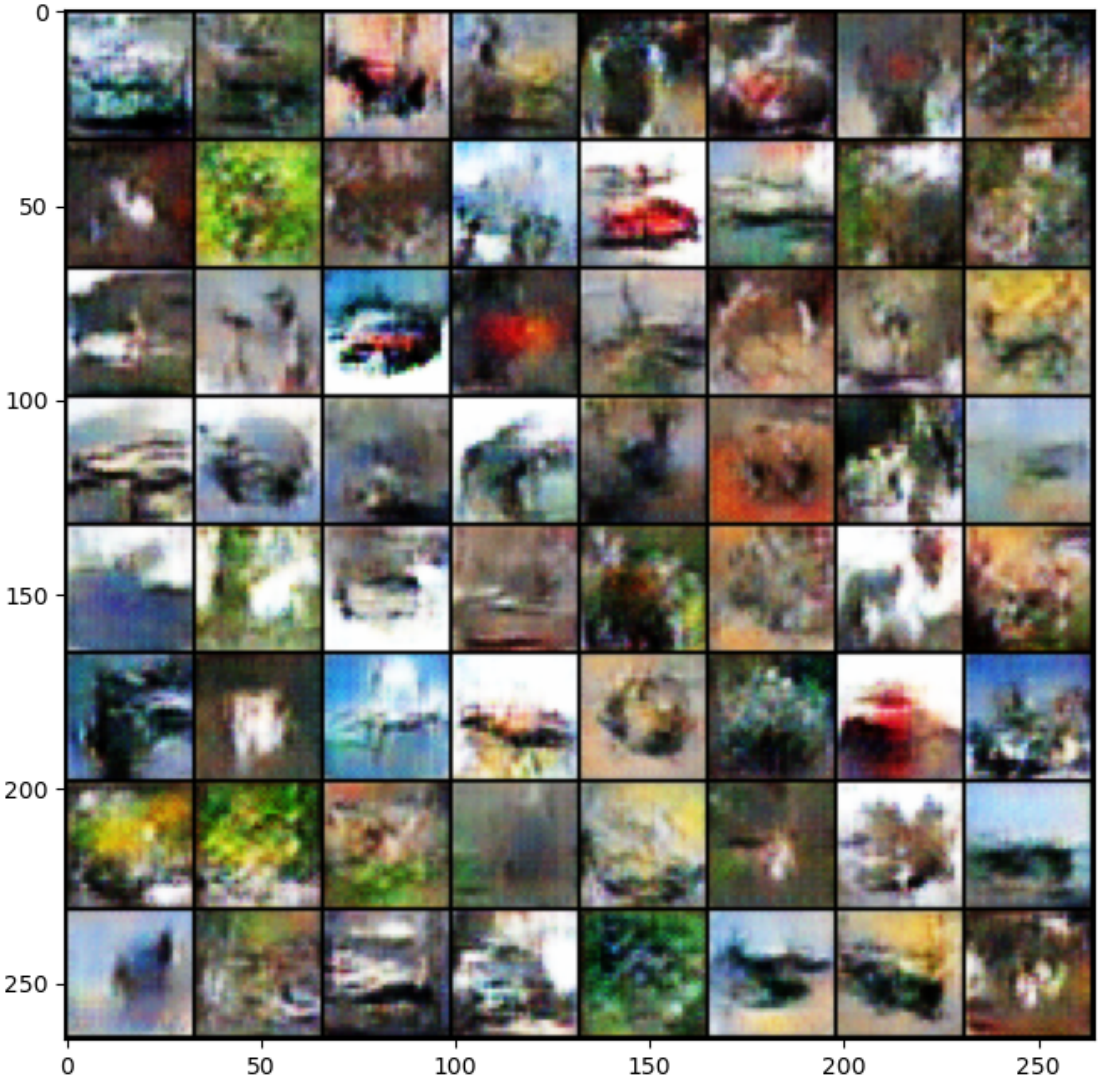

```
Iteration 2800/9750: dis loss = 0.6658, gen loss = 1.8274
Iteration 2900/9750: dis loss = 0.8419, gen loss = 2.2757
Iteration 3000/9750: dis loss = 0.6561, gen loss = 2.1102
Iteration 3100/9750: dis loss = 0.6518, gen loss = 1.6984
Iteration 3200/9750: dis loss = 0.6051, gen loss = 2.0230
Iteration 3300/9750: dis loss = 1.1153, gen loss = 2.9553
Iteration 3400/9750: dis loss = 0.7596, gen loss = 1.5591
Iteration 3500/9750: dis loss = 0.6145, gen loss = 1.1991
```

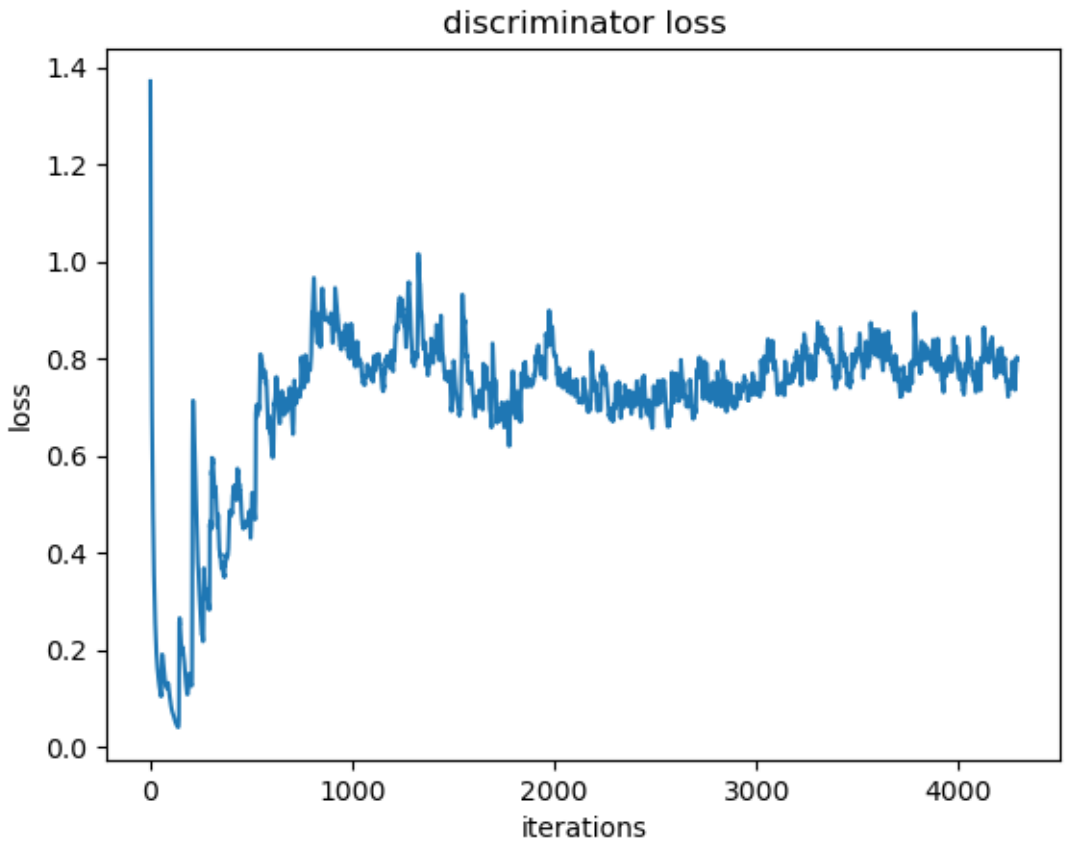

0
50
100
150
200
250
0
50
100
150
200
250

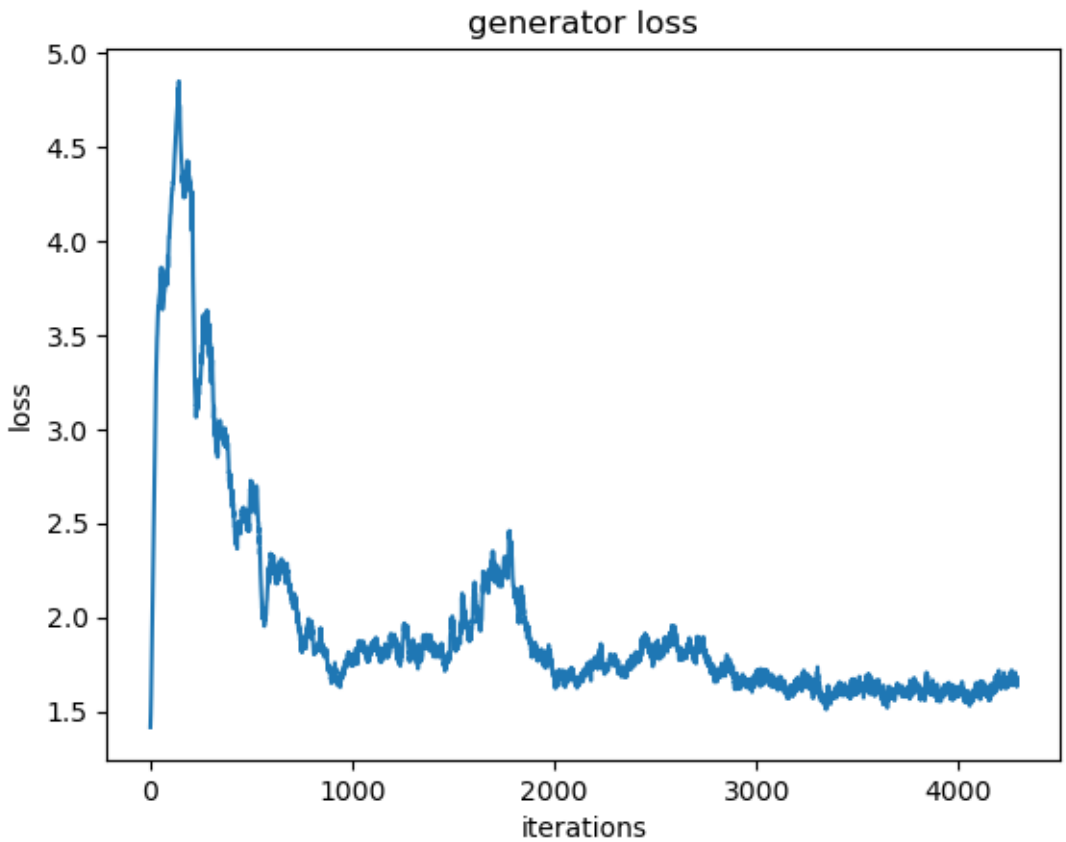

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
500
1000
1500
2000
2500
3000
3500
iterations



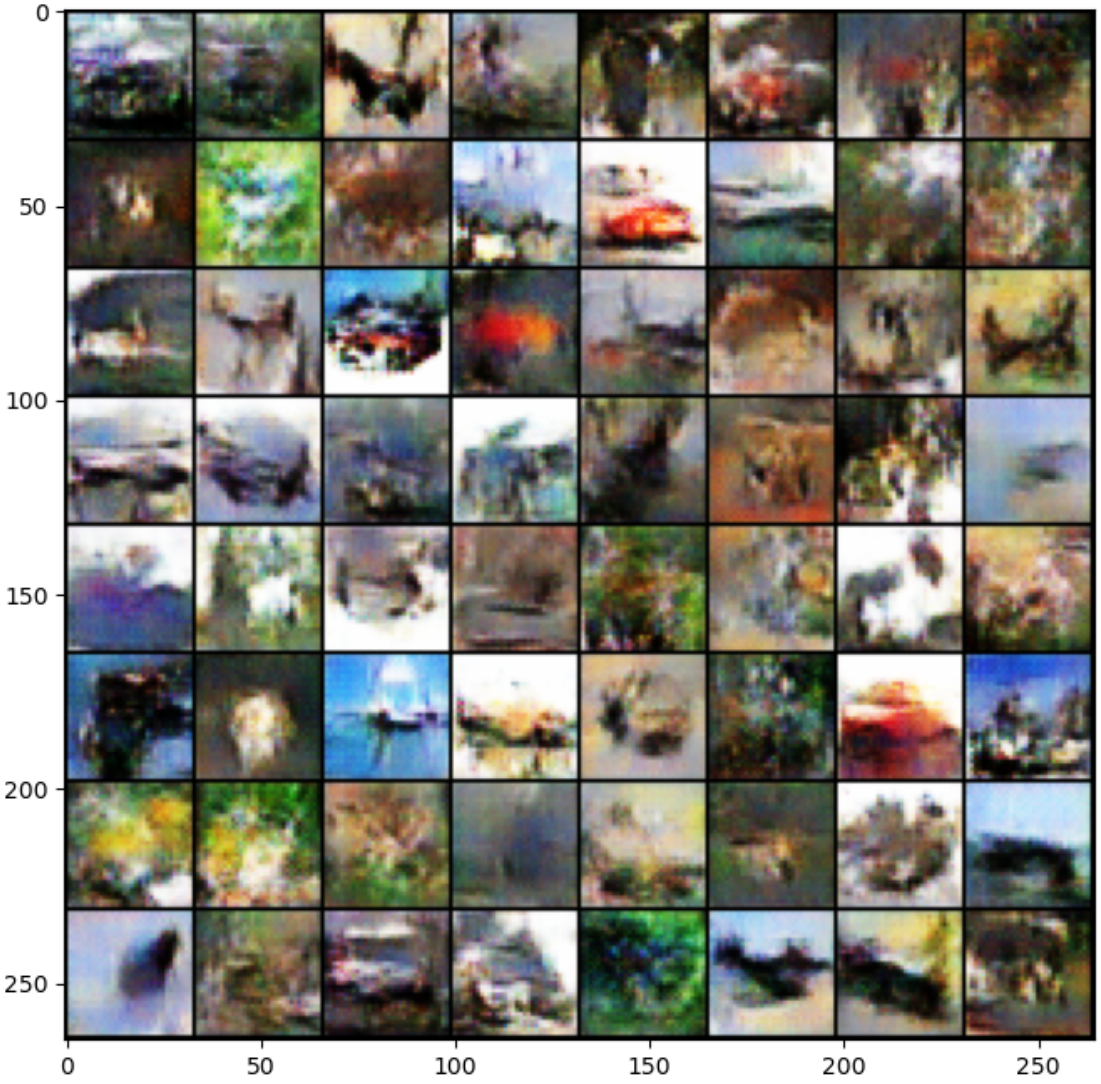

```
Iteration 3600/9750: dis loss = 0.7132, gen loss = 1.9764
Iteration 3700/9750: dis loss = 0.5755, gen loss = 1.4272
Iteration 3800/9750: dis loss = 0.7241, gen loss = 2.5225
Iteration 3900/9750: dis loss = 0.6077, gen loss = 2.0169
Iteration 4000/9750: dis loss = 0.7982, gen loss = 1.4439
Iteration 4100/9750: dis loss = 0.7790, gen loss = 1.1898
Iteration 4200/9750: dis loss = 0.8824, gen loss = 1.1229
```

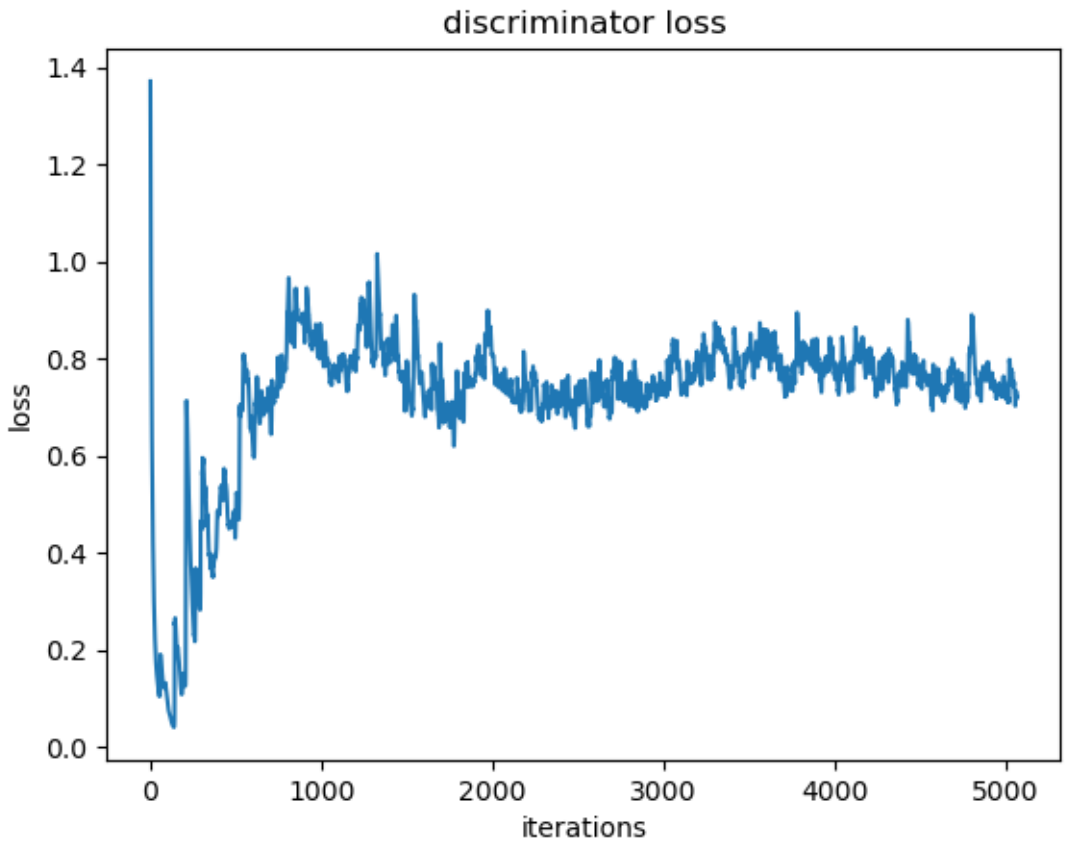

0
50
100
150
200
250
0
50
100
150
200
250

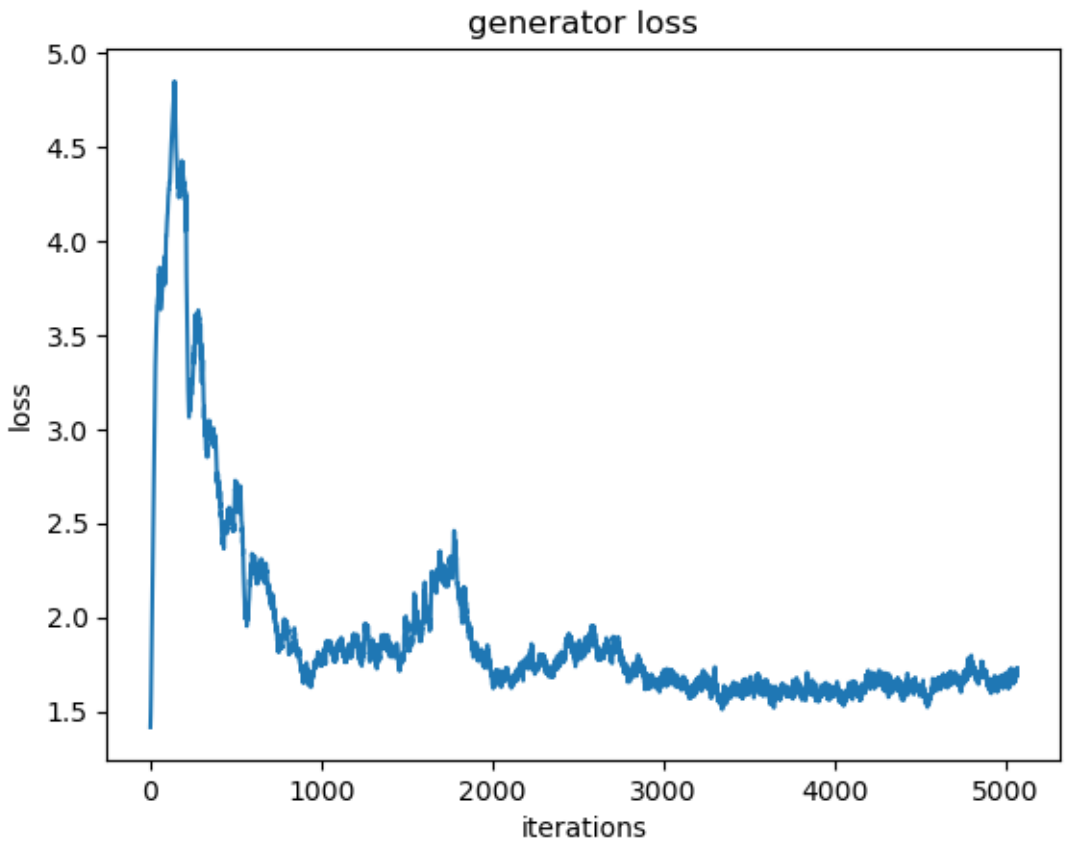

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
1000
2000
3000
4000
iterations



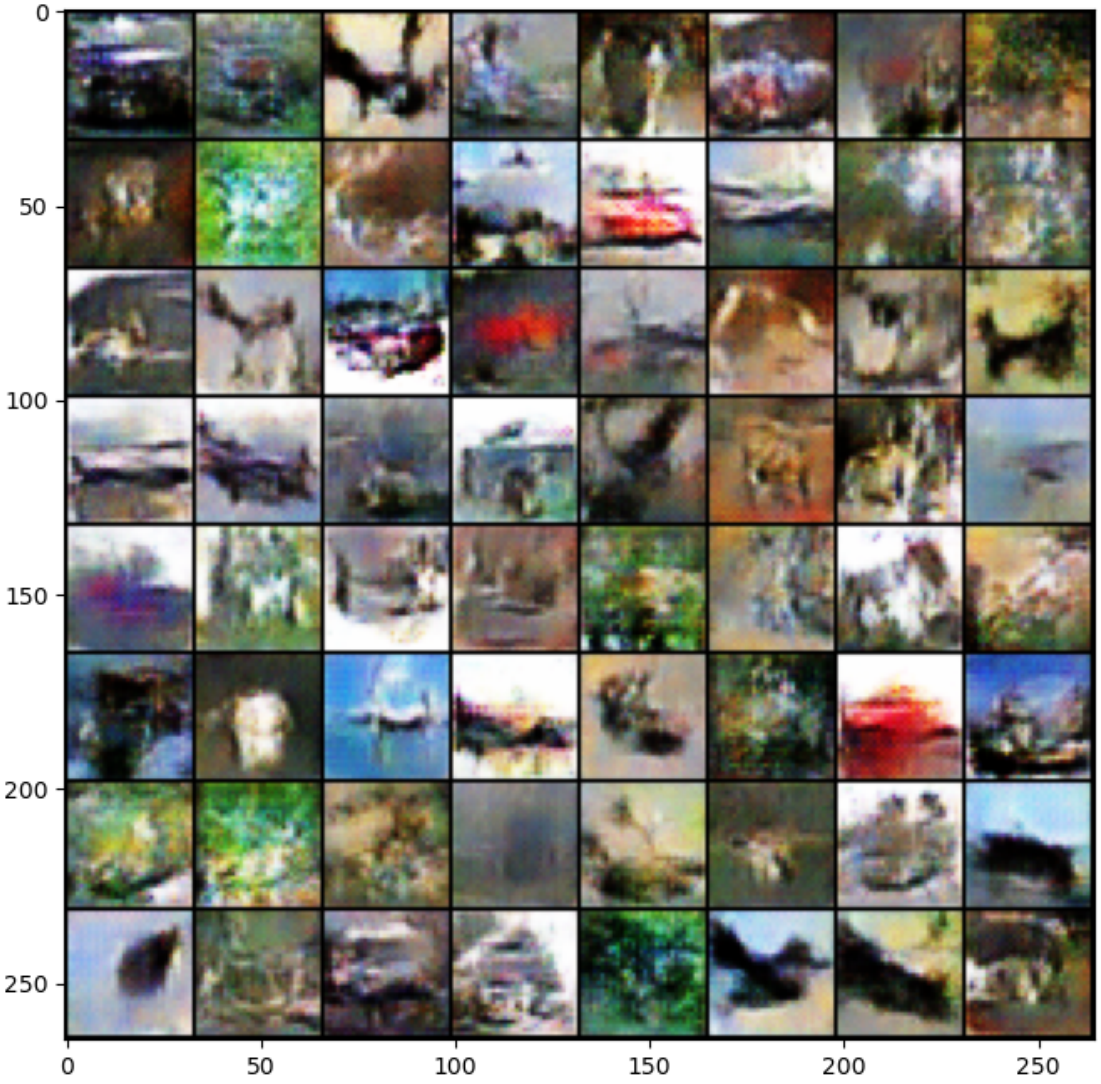

```
Iteration 4300/9750: dis loss = 0.6948, gen loss = 1.2909
Iteration 4400/9750: dis loss = 0.7788, gen loss = 1.9469
Iteration 4500/9750: dis loss = 0.9761, gen loss = 2.7550
Iteration 4600/9750: dis loss = 0.6969, gen loss = 1.4232
Iteration 4700/9750: dis loss = 0.6076, gen loss = 1.1733
Iteration 4800/9750: dis loss = 0.6840, gen loss = 2.0132
Iteration 4900/9750: dis loss = 0.6340, gen loss = 1.7229
Iteration 5000/9750: dis loss = 0.5410, gen loss = 1.9410
```

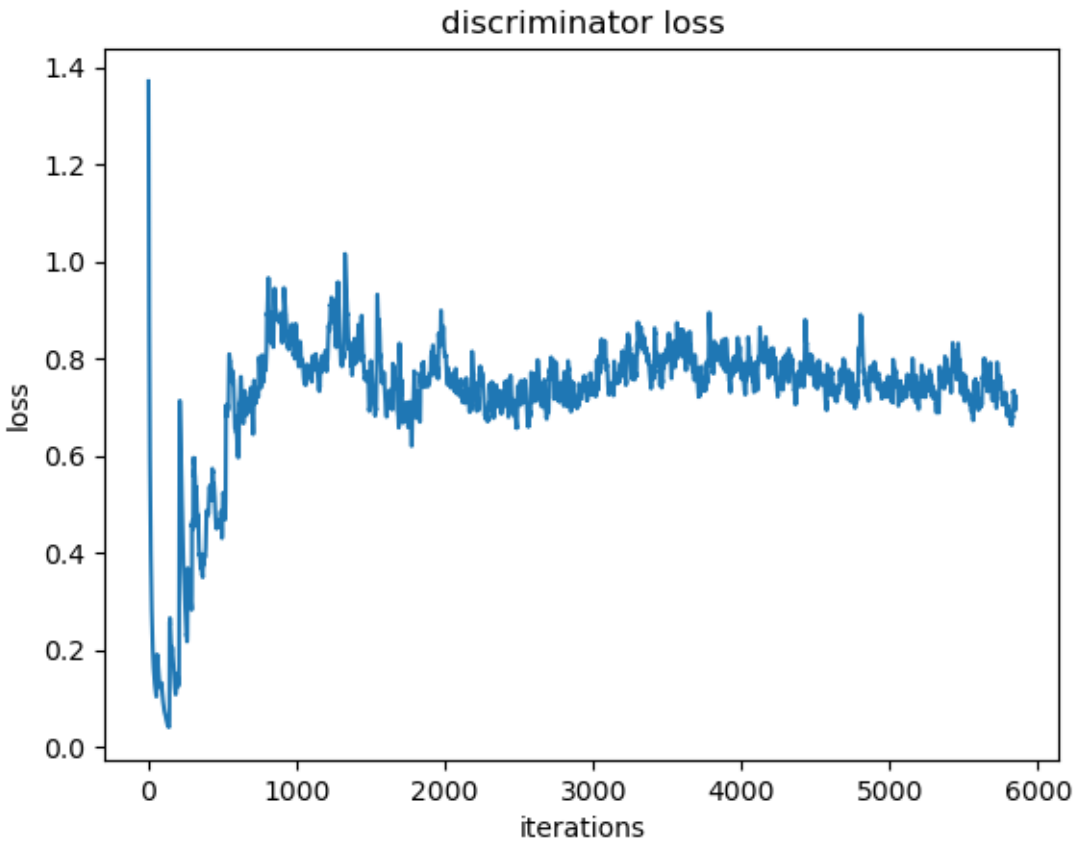

0
50
100
150
200
250
0
50
100
150
200
250

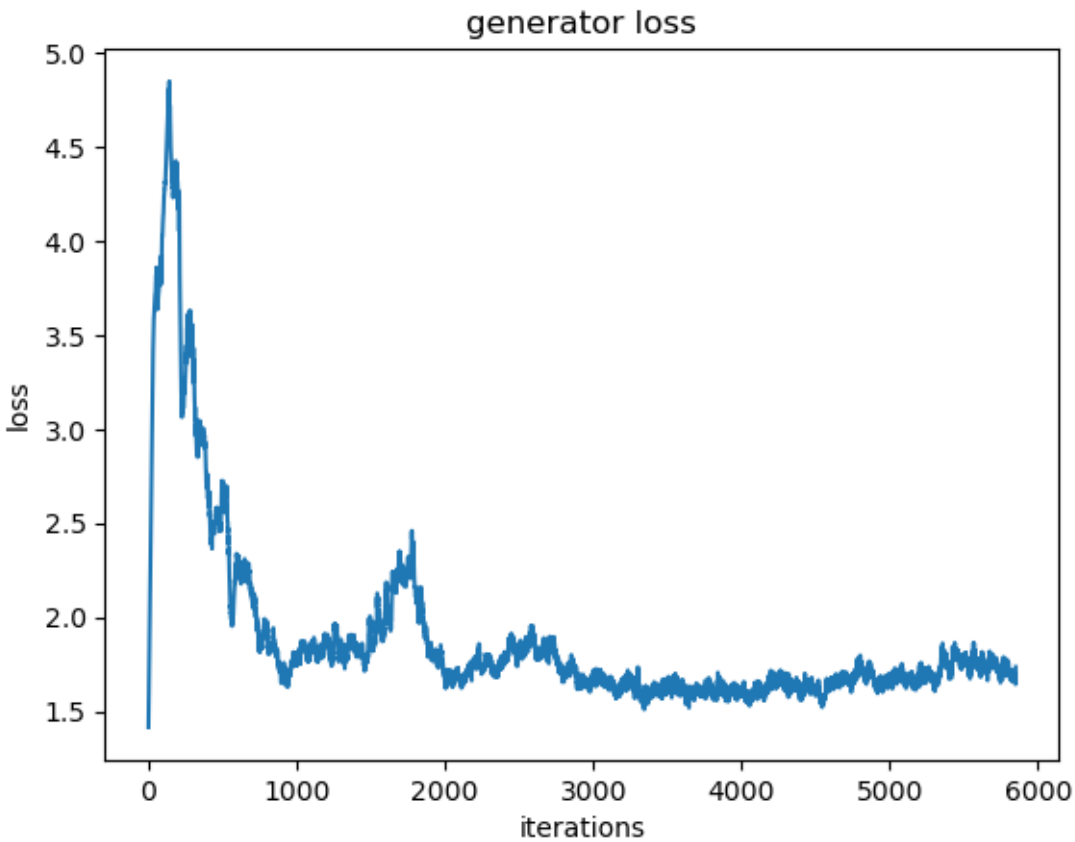

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
1000
2000
3000
4000
5000
iterations



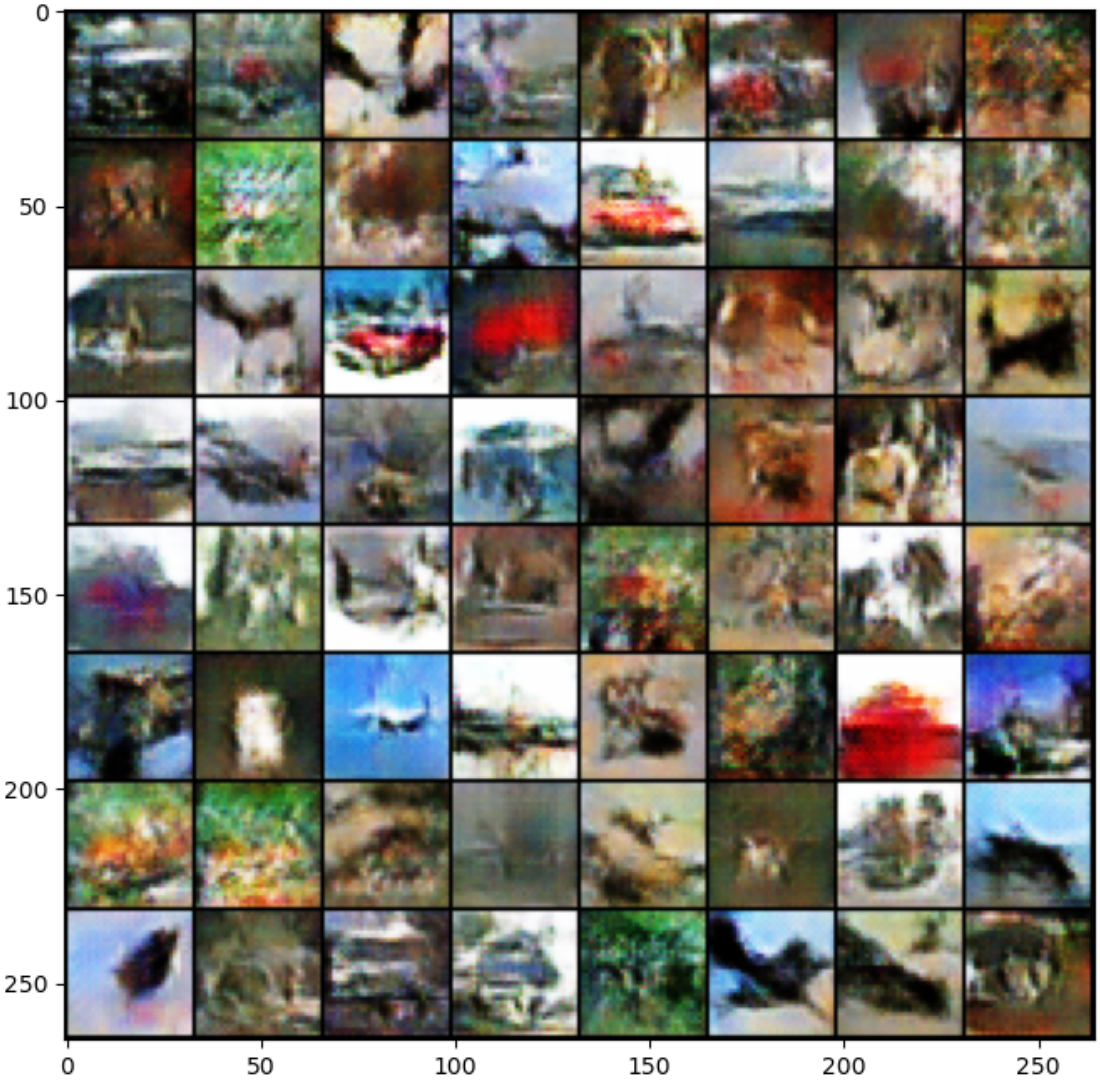

```
Iteration 5100/9750: dis loss = 0.7757, gen loss = 0.9025
Iteration 5200/9750: dis loss = 0.7906, gen loss = 2.5084
Iteration 5300/9750: dis loss = 0.6890, gen loss = 0.8555
Iteration 5400/9750: dis loss = 0.5318, gen loss = 1.9017
Iteration 5500/9750: dis loss = 0.5088, gen loss = 1.7231
Iteration 5600/9750: dis loss = 0.7002, gen loss = 2.5781
Iteration 5700/9750: dis loss = 0.6319, gen loss = 1.1471
Iteration 5800/9750: dis loss = 0.6663, gen loss = 1.7128
```

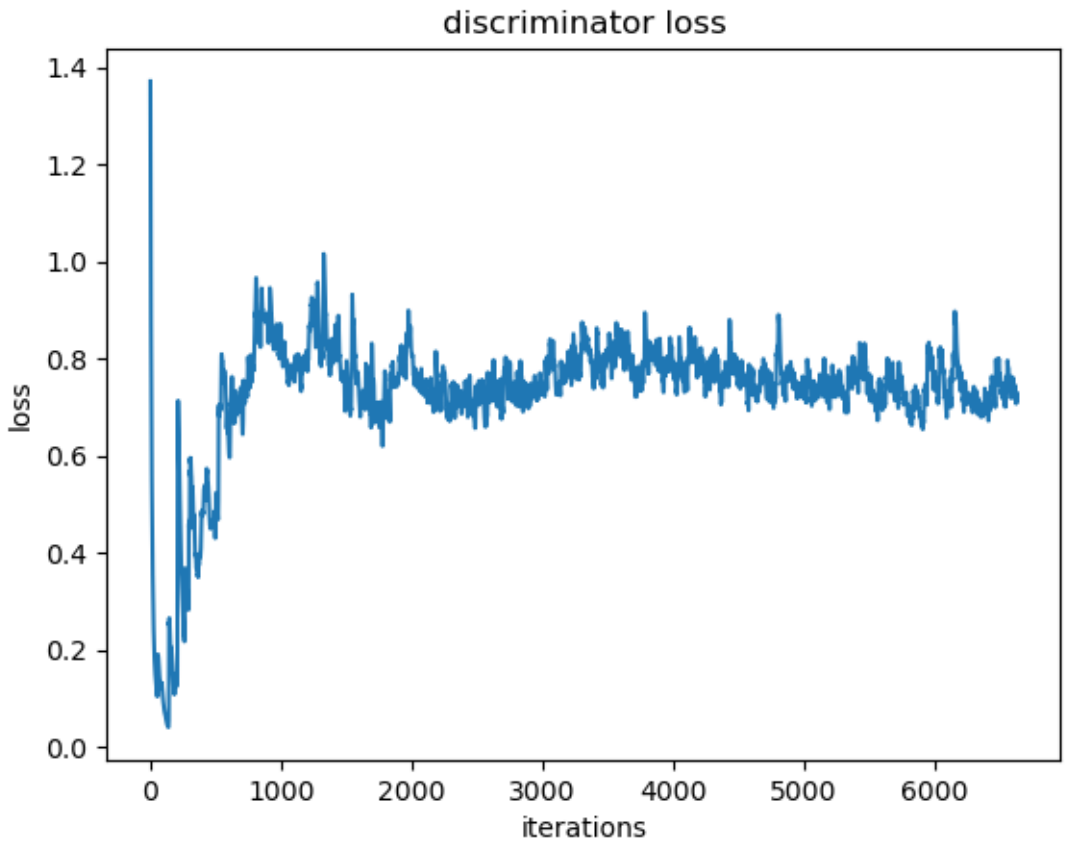

0
50
100
150
200
250
0
50
100
150
200
250

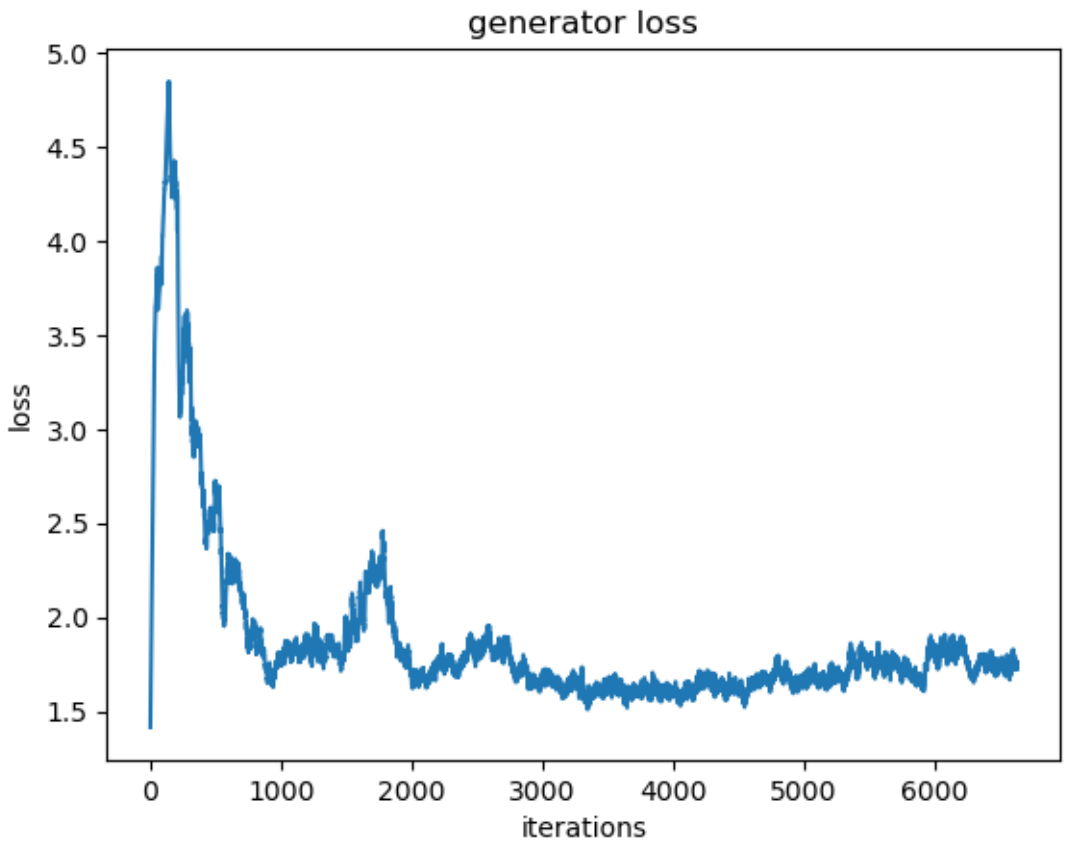

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
1000
2000
3000
4000
5000
6000
iterations



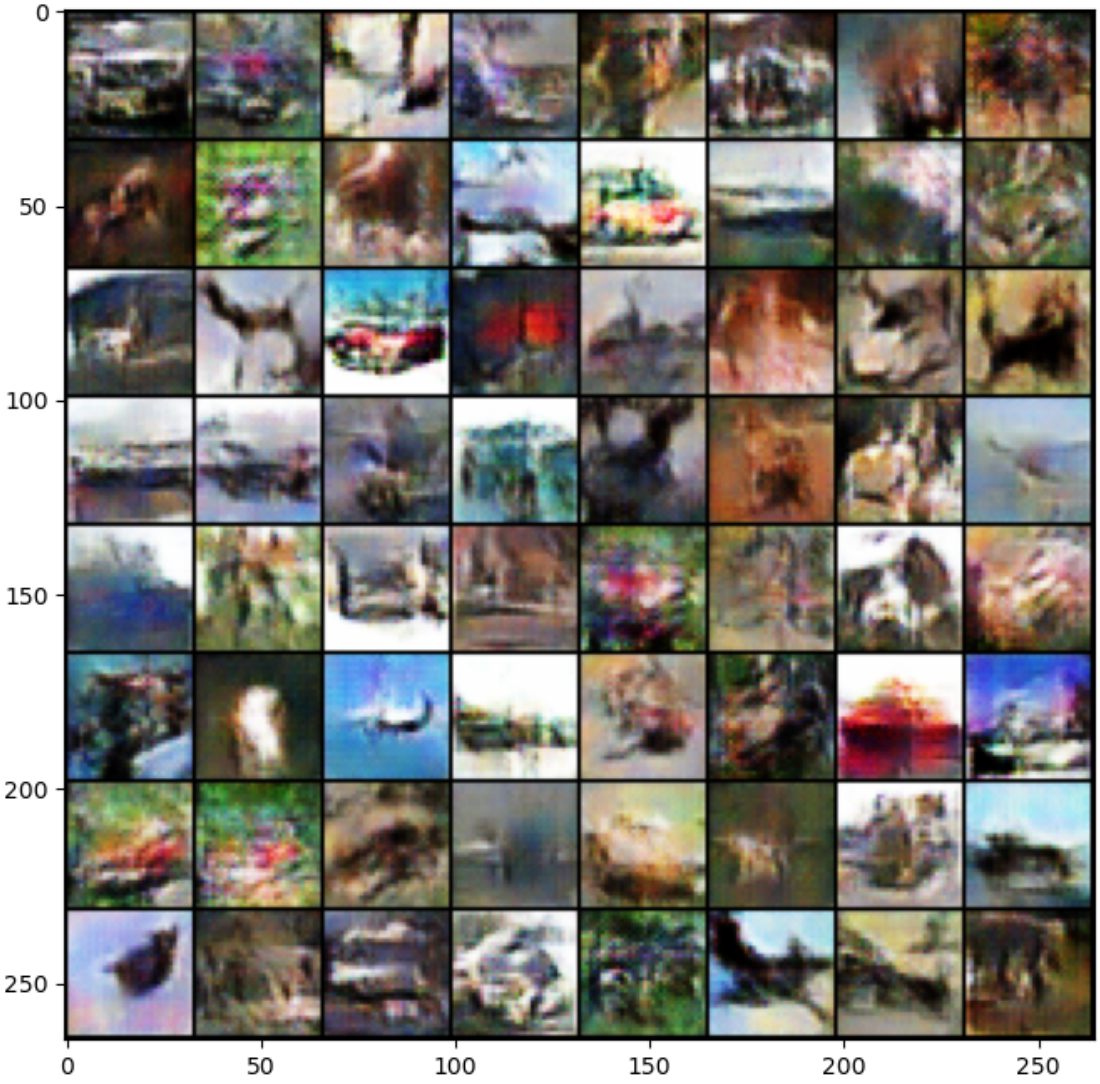

```
Iteration 5900/9750: dis loss = 0.6828, gen loss = 1.8328
Iteration 6000/9750: dis loss = 1.0572, gen loss = 2.8559
Iteration 6100/9750: dis loss = 0.5911, gen loss = 1.7714
Iteration 6200/9750: dis loss = 1.1928, gen loss = 3.6852
Iteration 6300/9750: dis loss = 0.6053, gen loss = 1.1310
Iteration 6400/9750: dis loss = 0.6964, gen loss = 1.2593
Iteration 6500/9750: dis loss = 0.6599, gen loss = 1.4890
Iteration 6600/9750: dis loss = 0.5929, gen loss = 1.4351
```

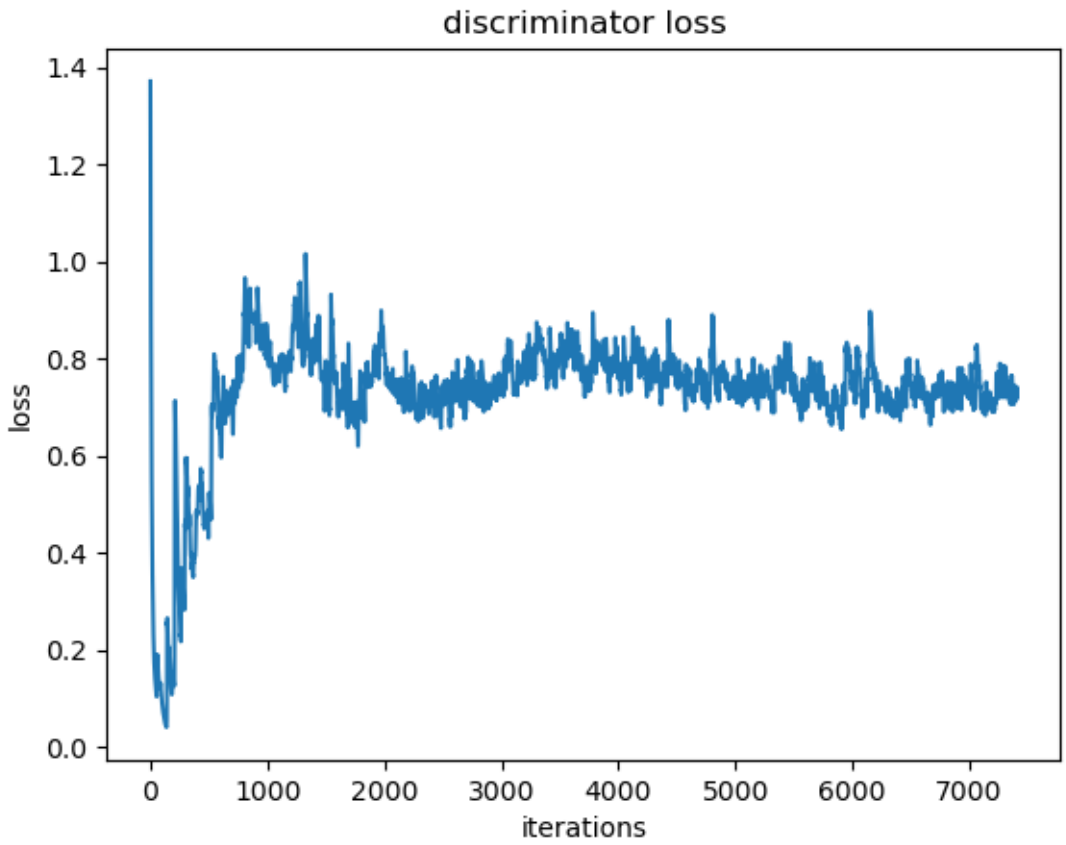

0
50
100
150
200
250
0
50
100
150
200
250

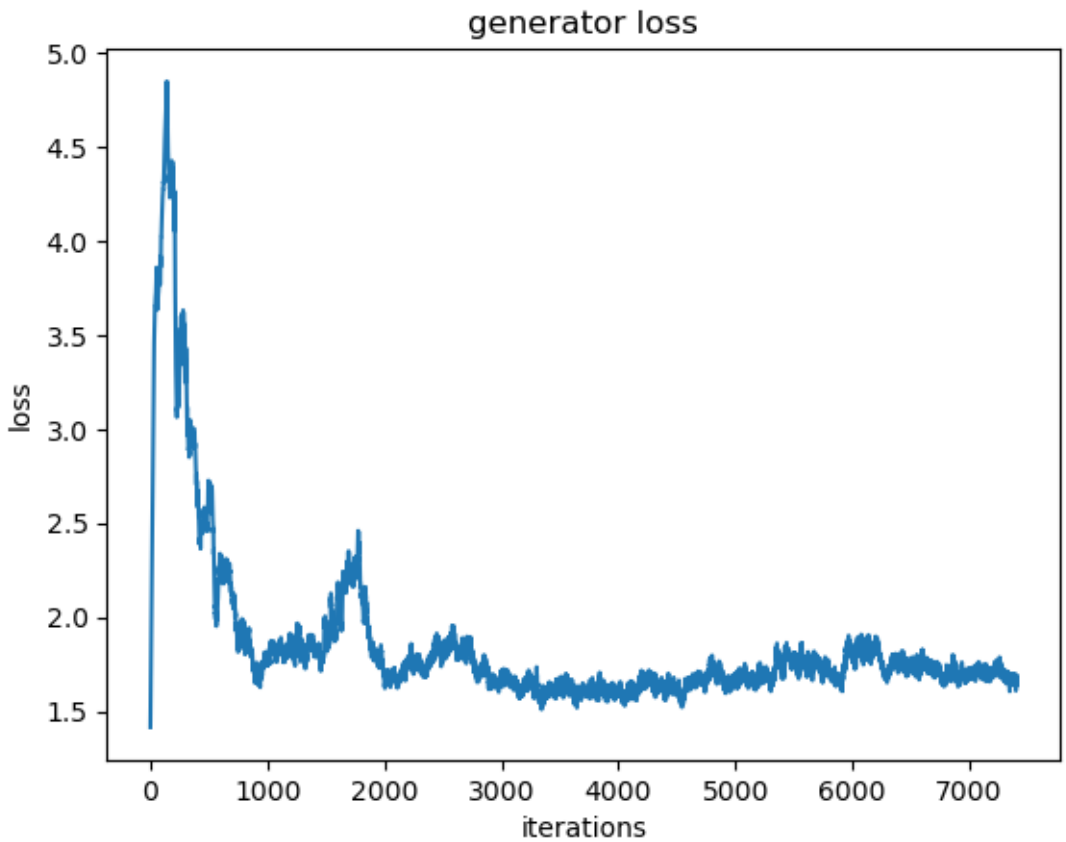

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
1000
2000
3000
4000
5000
6000
iterations



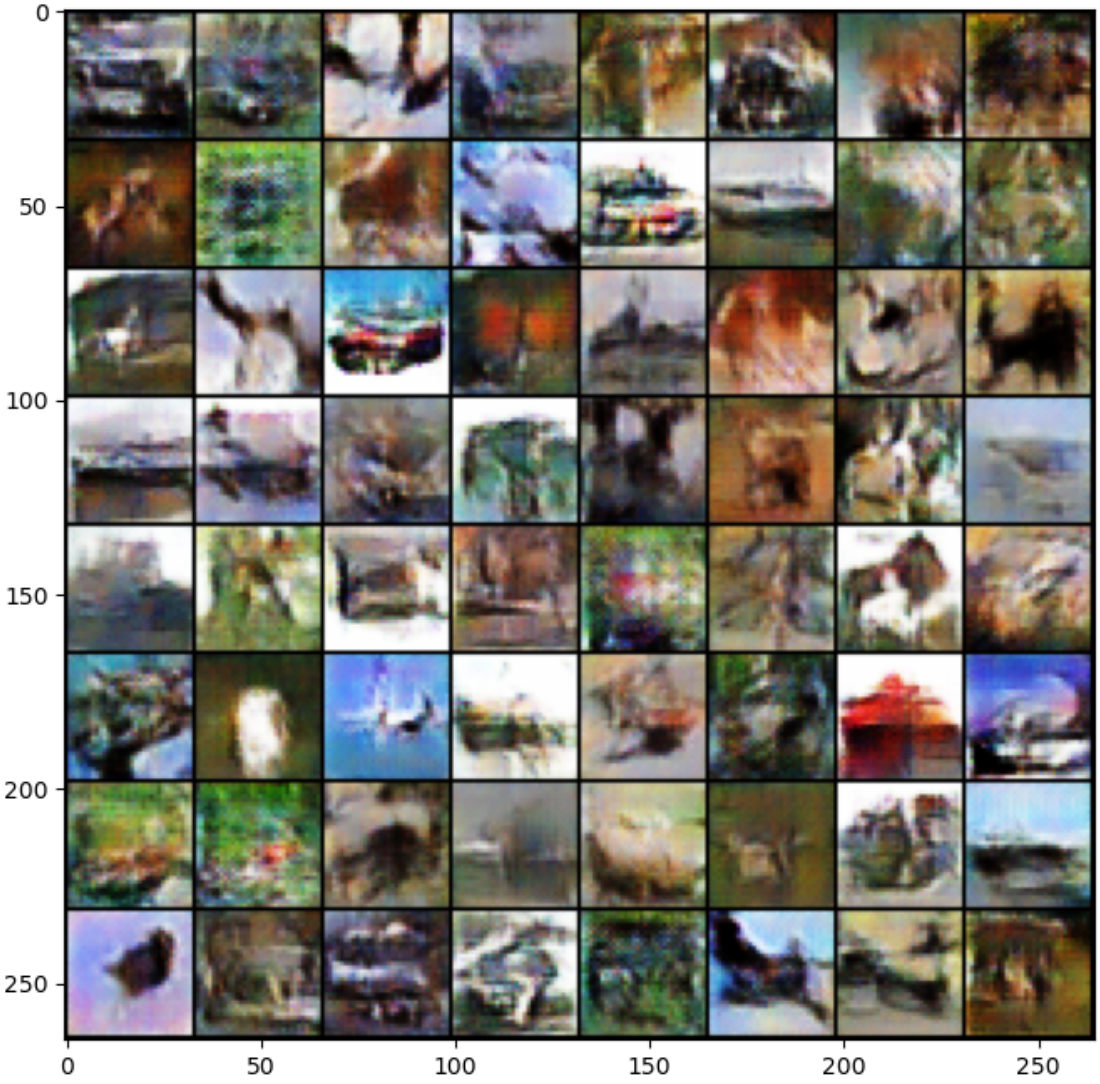

```
Iteration 6700/9750: dis loss = 0.7579, gen loss = 1.3949
Iteration 6800/9750: dis loss = 0.7014, gen loss = 1.2585
Iteration 6900/9750: dis loss = 0.6521, gen loss = 1.9968
Iteration 7000/9750: dis loss = 0.4714, gen loss = 1.6044
Iteration 7100/9750: dis loss = 0.5687, gen loss = 1.4406
Iteration 7200/9750: dis loss = 0.7141, gen loss = 1.2362
Iteration 7300/9750: dis loss = 0.7834, gen loss = 0.8456
Iteration 7400/9750: dis loss = 0.5765, gen loss = 2.1490
```

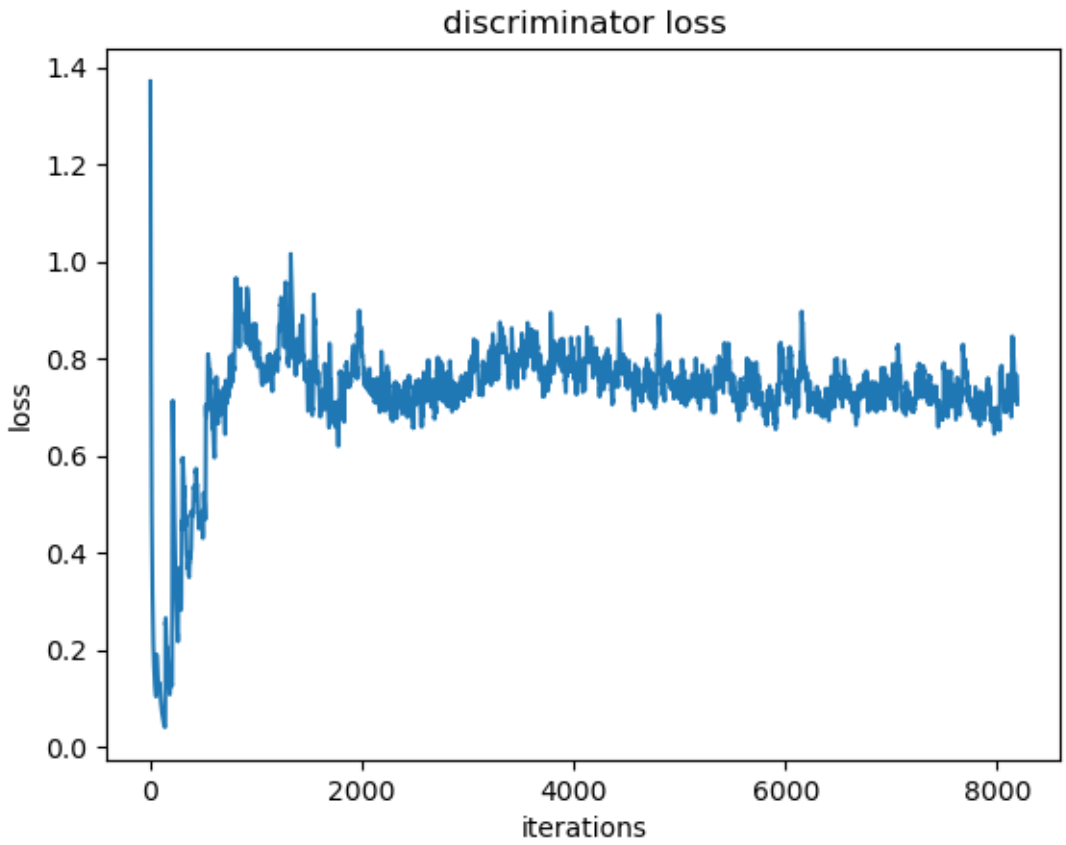

0
50
100
150
200
250
0
50
100
150
200
250

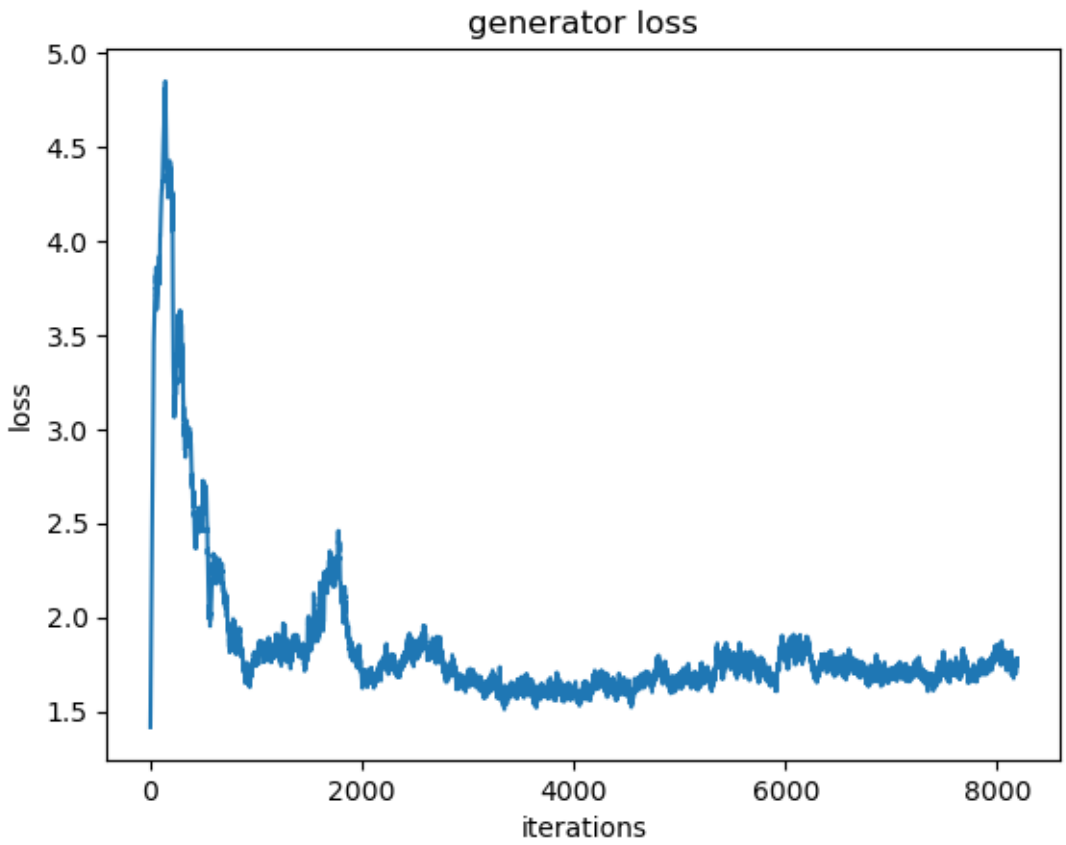

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
1000
2000
3000
4000
5000
6000
7000
iterations



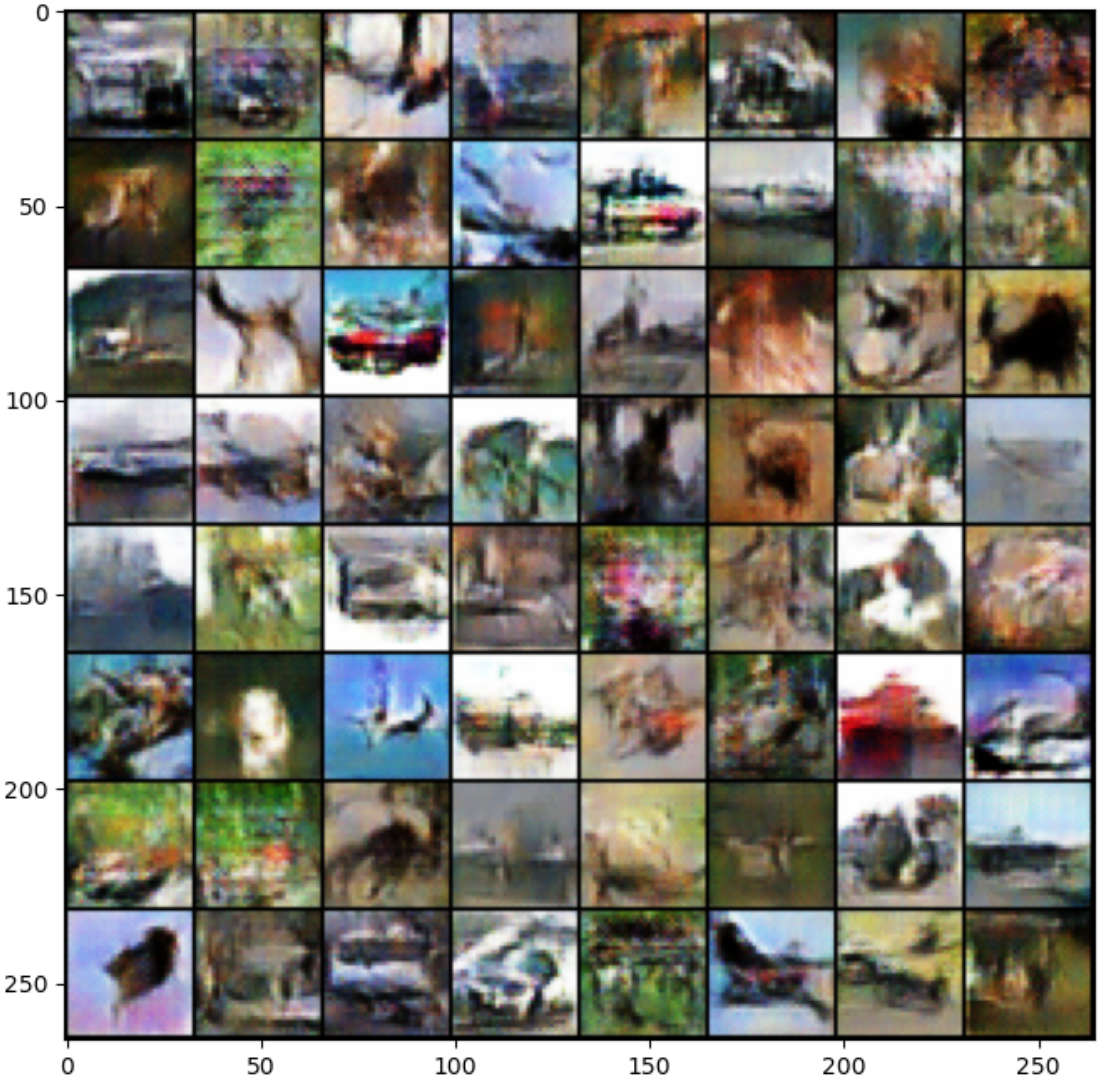

```
Iteration 7500/9750: dis loss = 0.8478, gen loss = 2.2206
Iteration 7600/9750: dis loss = 0.7452, gen loss = 1.9621
Iteration 7700/9750: dis loss = 0.7625, gen loss = 2.1701
Iteration 7800/9750: dis loss = 0.6138, gen loss = 1.8243
Iteration 7900/9750: dis loss = 0.7888, gen loss = 1.3947
Iteration 8000/9750: dis loss = 0.6898, gen loss = 1.2791
Iteration 8100/9750: dis loss = 0.6177, gen loss = 1.5783
```

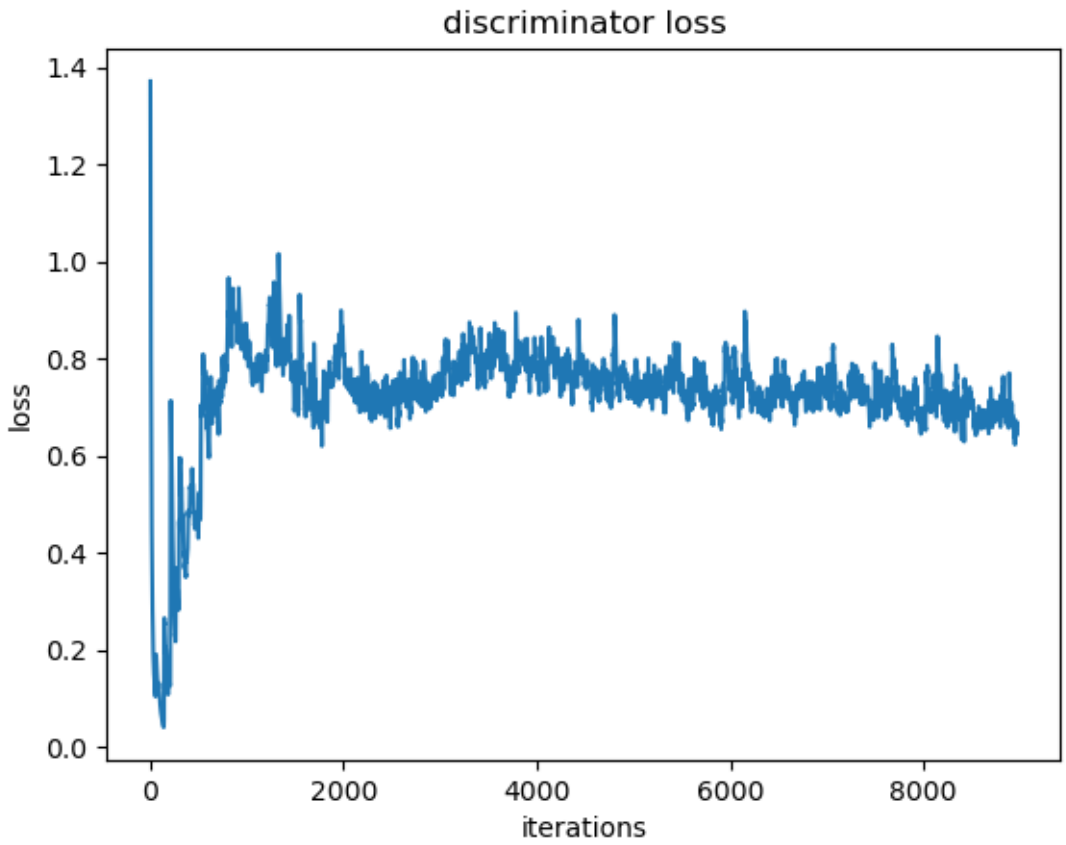

0
50
100
150
200
250
0
50
100
150
200
250

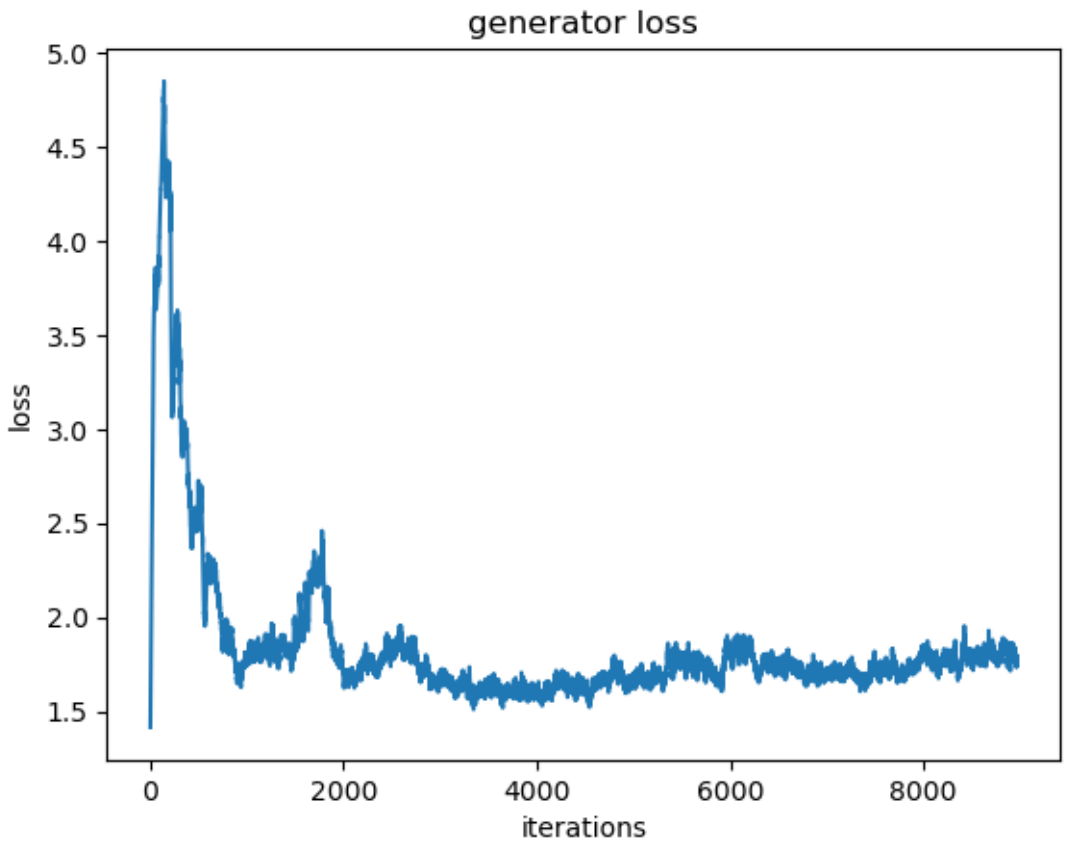

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
2000
4000
6000
8000
iterations



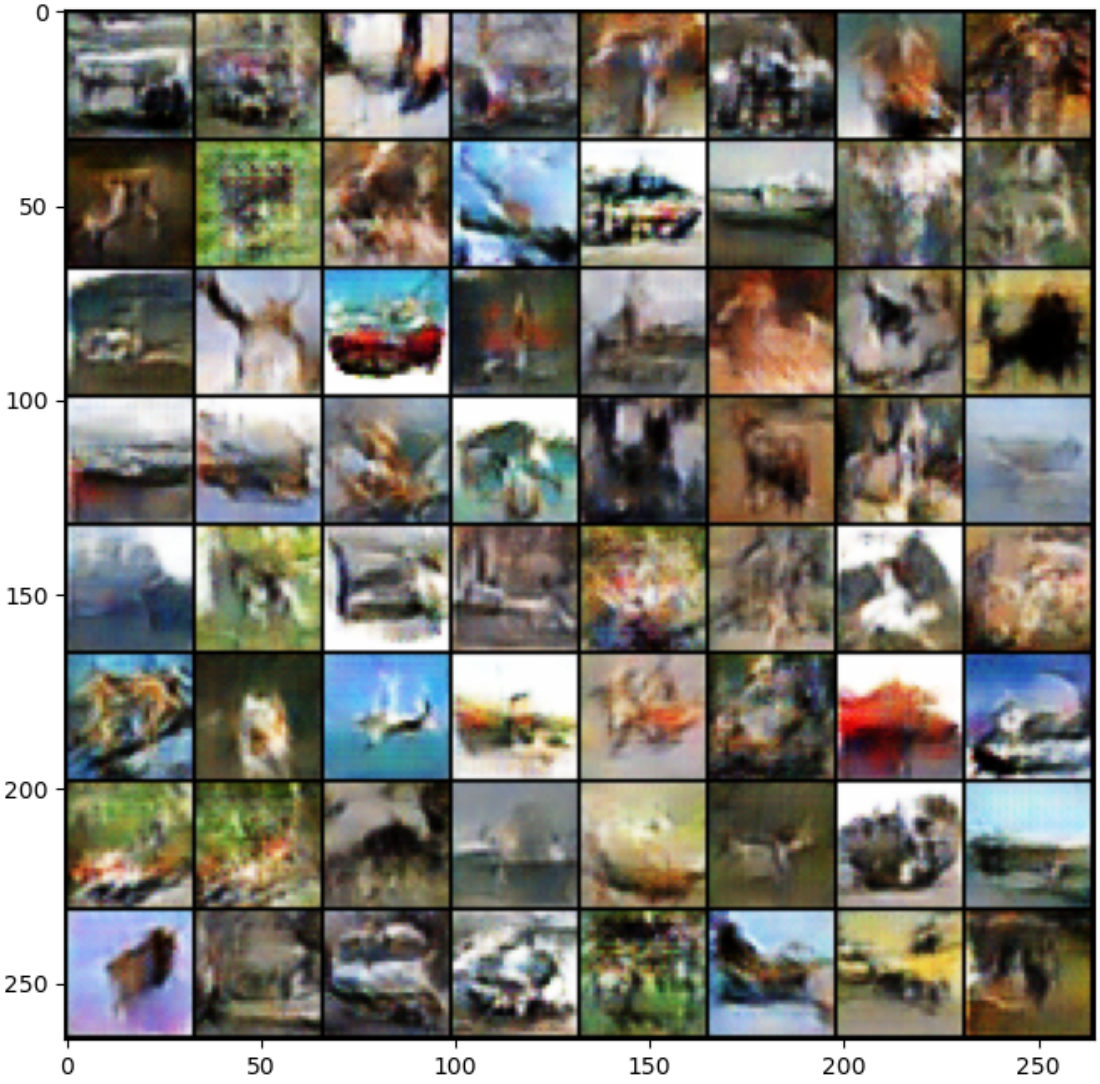

```
Iteration 8200/9750: dis loss = 0.6188, gen loss = 1.8094
Iteration 8300/9750: dis loss = 0.5760, gen loss = 1.2879
Iteration 8400/9750: dis loss = 0.6796, gen loss = 1.3689
Iteration 8500/9750: dis loss = 0.6951, gen loss = 2.5535
Iteration 8600/9750: dis loss = 0.5885, gen loss = 1.8402
Iteration 8700/9750: dis loss = 0.5472, gen loss = 1.3302
Iteration 8800/9750: dis loss = 1.3341, gen loss = 2.2344
Iteration 8900/9750: dis loss = 0.8085, gen loss = 2.6602
```

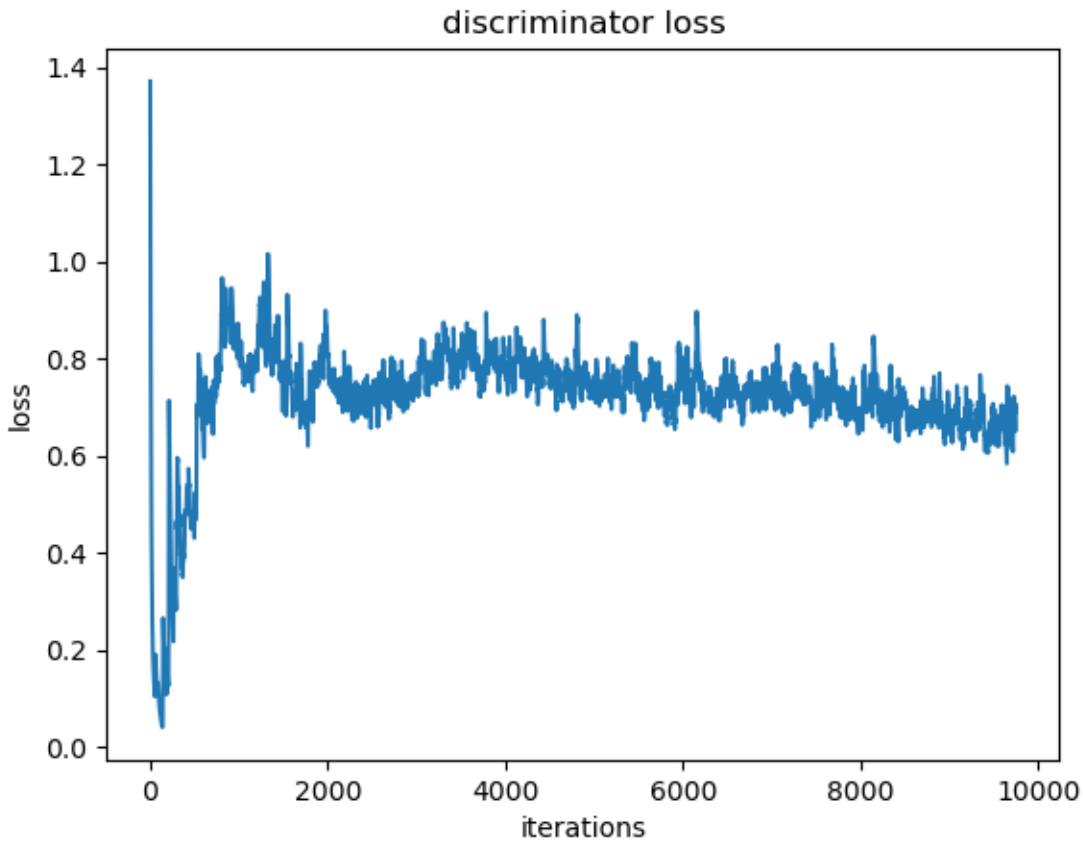

0
50
100
150
200
250
0
50
100
150
200
250

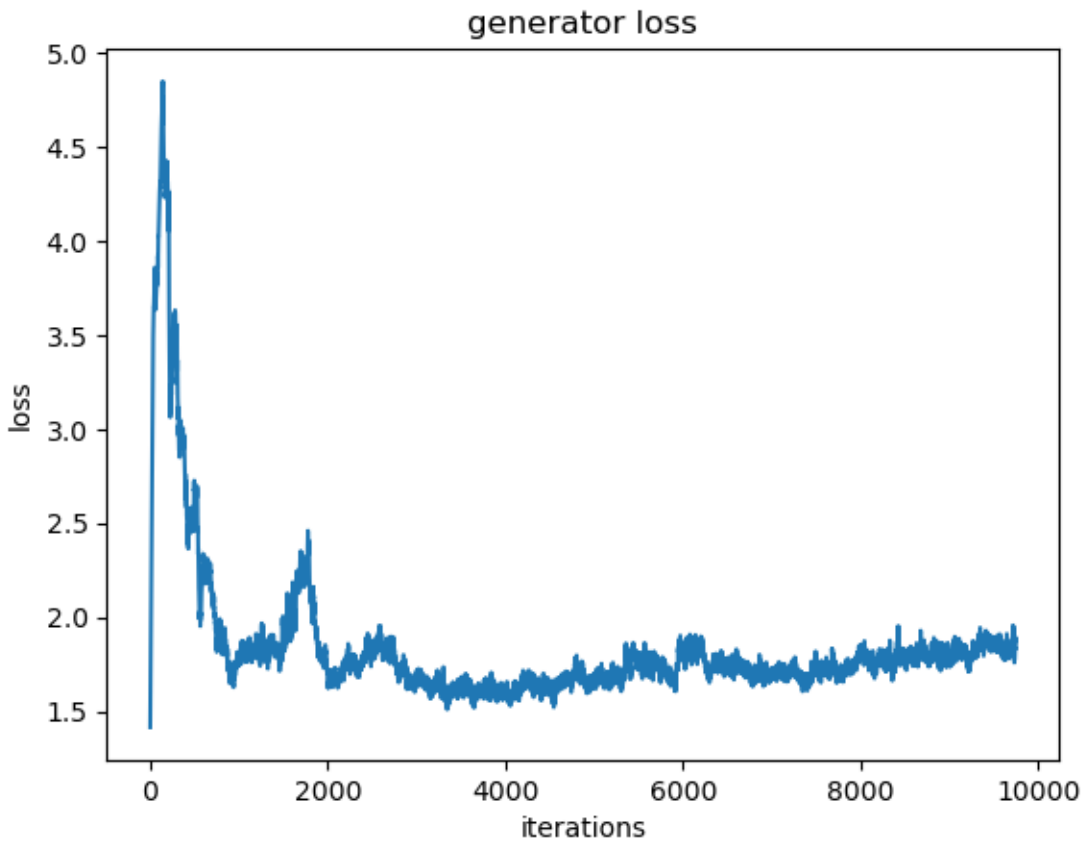

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
2000
4000
6000
8000
iterations




```
Iteration 9000/9750: dis loss = 0.8037, gen loss = 2.2590
Iteration 9100/9750: dis loss = 0.6720, gen loss = 1.4181
Iteration 9200/9750: dis loss = 0.5522, gen loss = 1.9680
Iteration 9300/9750: dis loss = 0.7473, gen loss = 1.3667
Iteration 9400/9750: dis loss = 0.5034, gen loss = 2.4027
Iteration 9500/9750: dis loss = 0.9743, gen loss = 0.8506
Iteration 9600/9750: dis loss = 0.4941, gen loss = 2.3879
Iteration 9700/9750: dis loss = 0.5570, gen loss = 2.0988
```


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


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




```
… Done!
```

4 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: > This would add noise and randomness to the discriminator and make it hard to distinguish real from fake images. Also disrupting the learning by normalizing data in the middle of training, this would disturp the model from learning pixel correlation in the real images.

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

---

5 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)

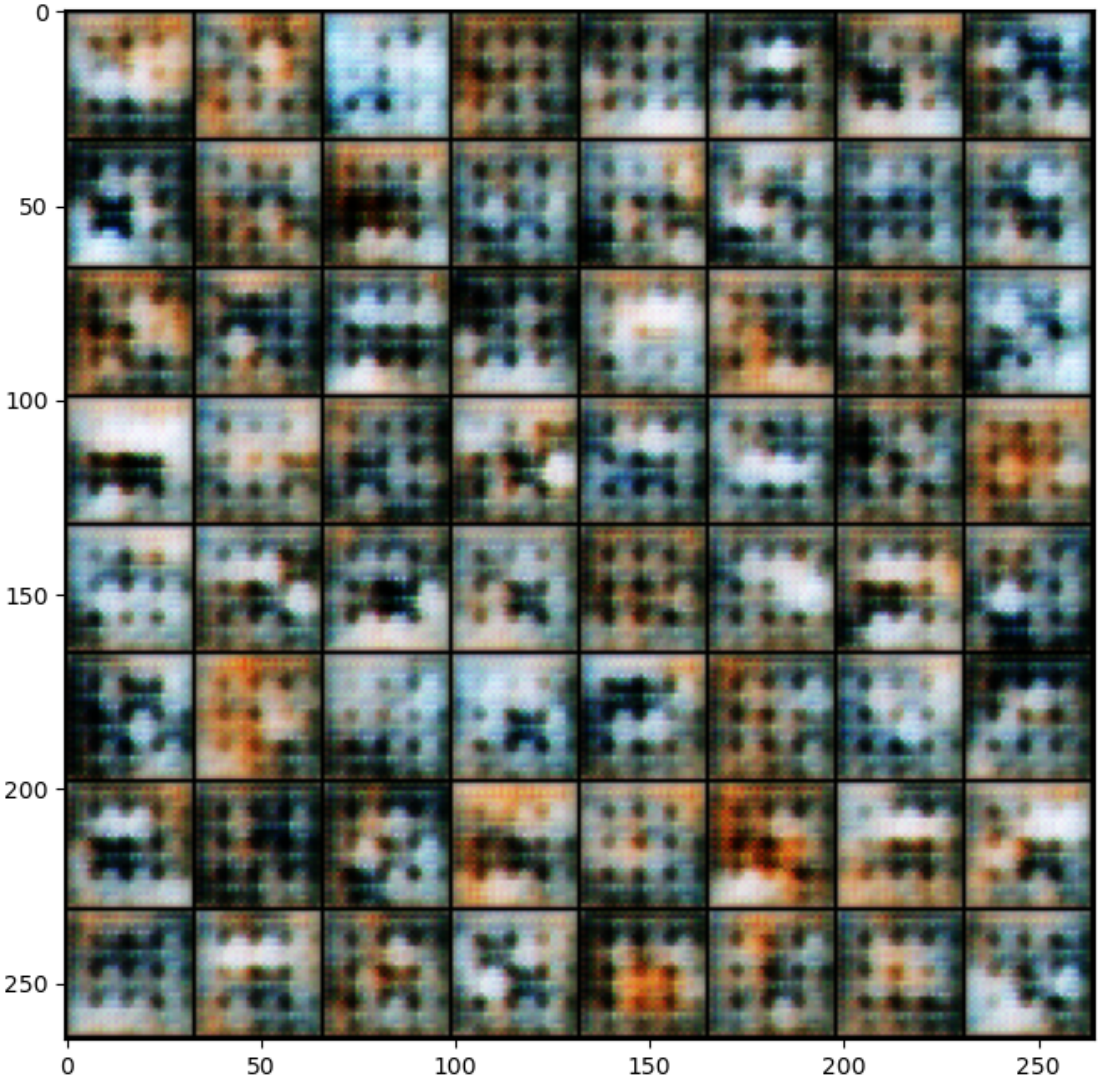
[66]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
       ␣
↪##############################################################################
        # Prob 2-3:                                                            ␣
↪        #
        # adding spectral norm to the discriminator                            ␣
↪        #
       ␣
↪##############################################################################
        self.downsample = nn.Sequential(
            torch.nn.utils.spectral_norm(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),
```

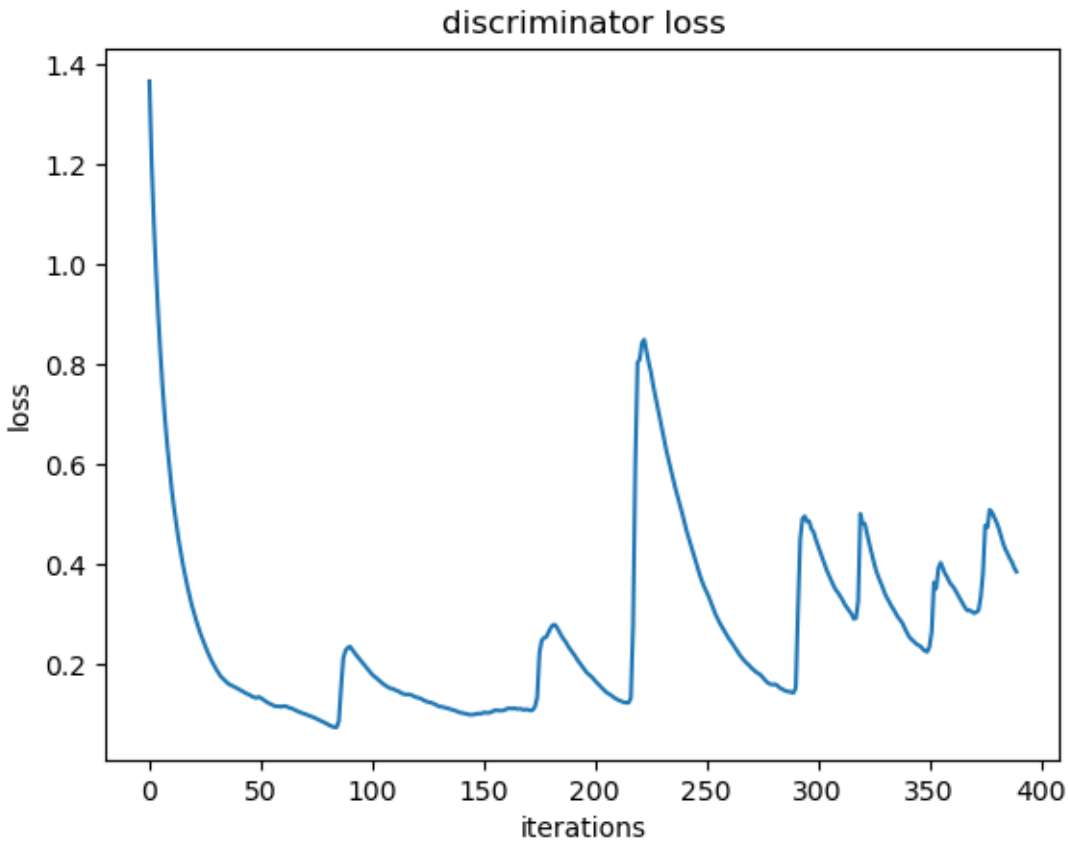
```
            nn.BatchNorm2d(128),
            nn.LeakyReLU(),
        )
        ␣
↪##############################################################################
        # END OF YOUR CODE                                                    ␣
↪        #
        ␣
↪##############################################################################
        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
```

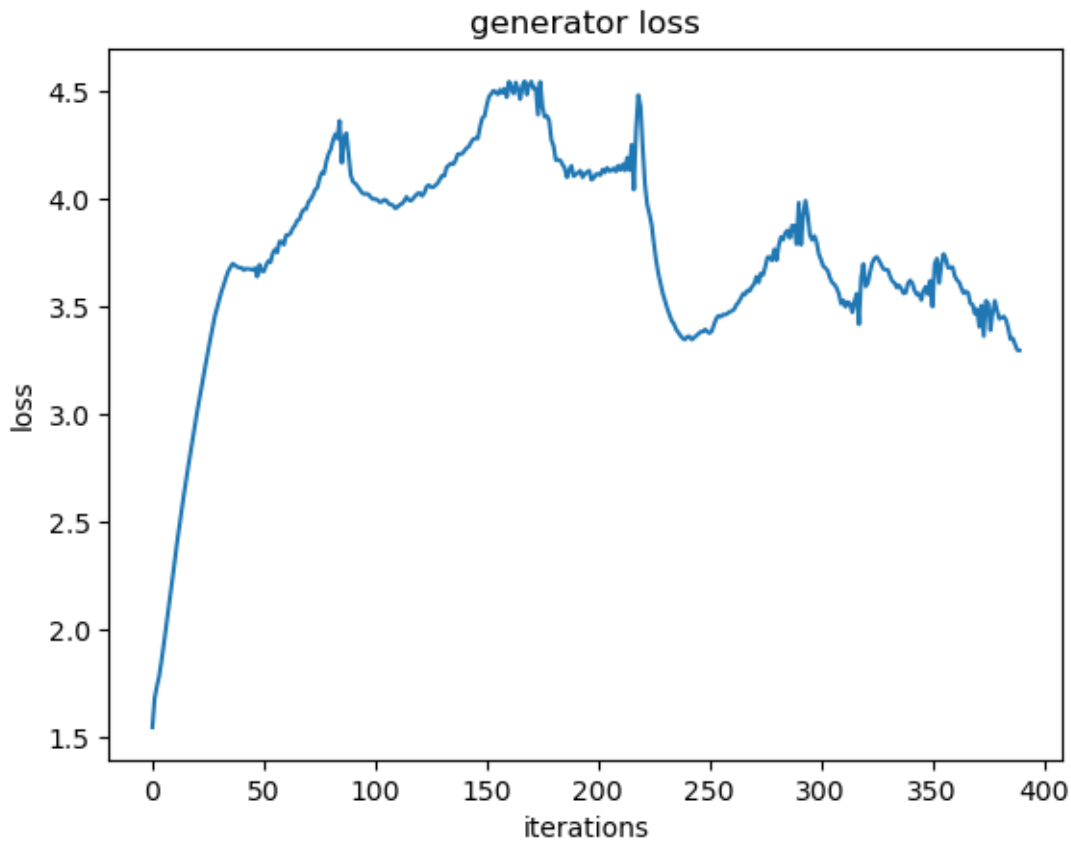
After adding the spectral norm to the discriminator, redo the training block below to see the effects.

[67]:
```
set_seed(42)

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

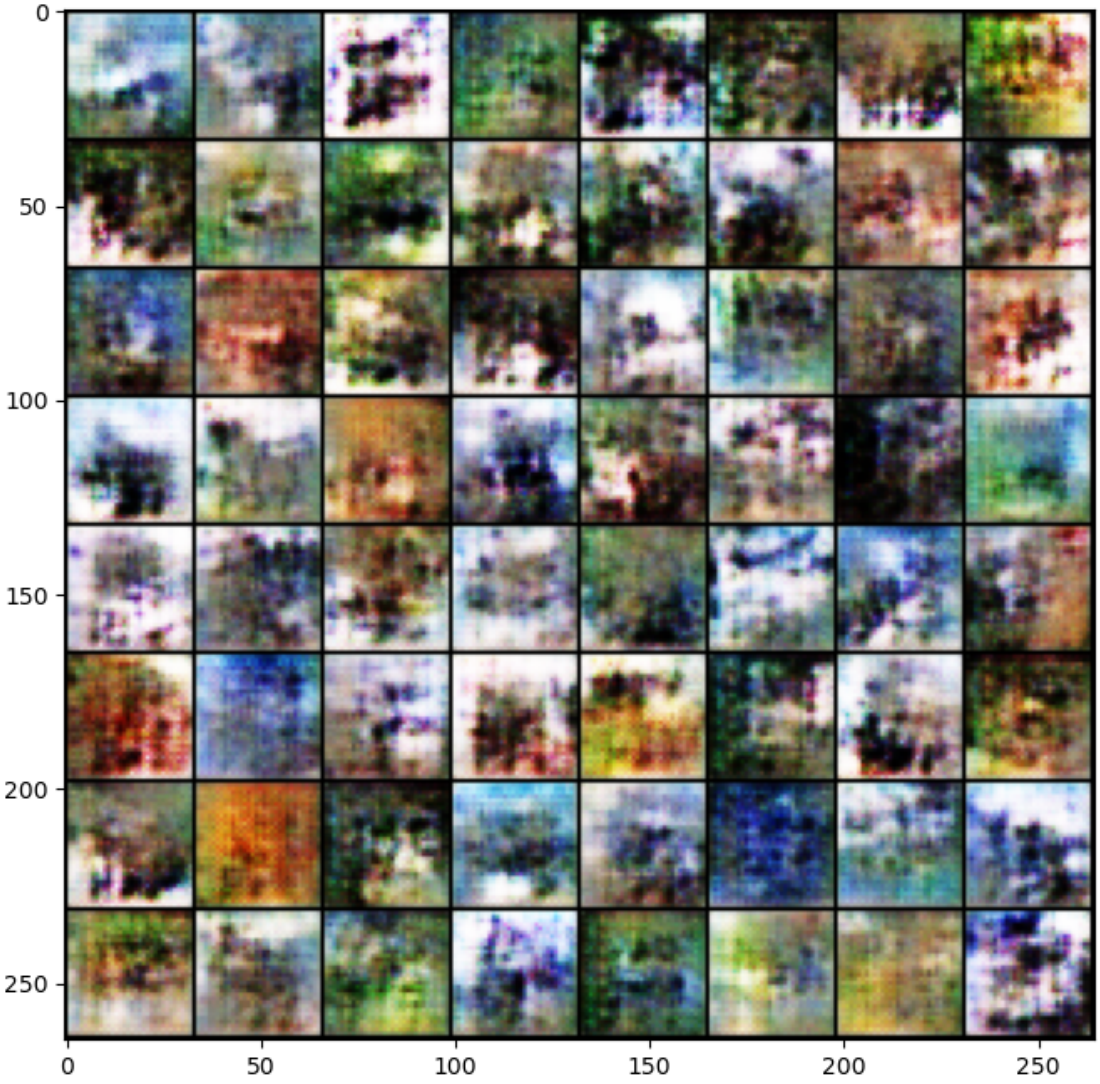
```
Start training …
Iteration 100/9750: dis loss = 0.0818, gen loss = 3.8305
Iteration 200/9750: dis loss = 0.0881, gen loss = 4.4589
Iteration 300/9750: dis loss = 0.1612, gen loss = 2.6867
```

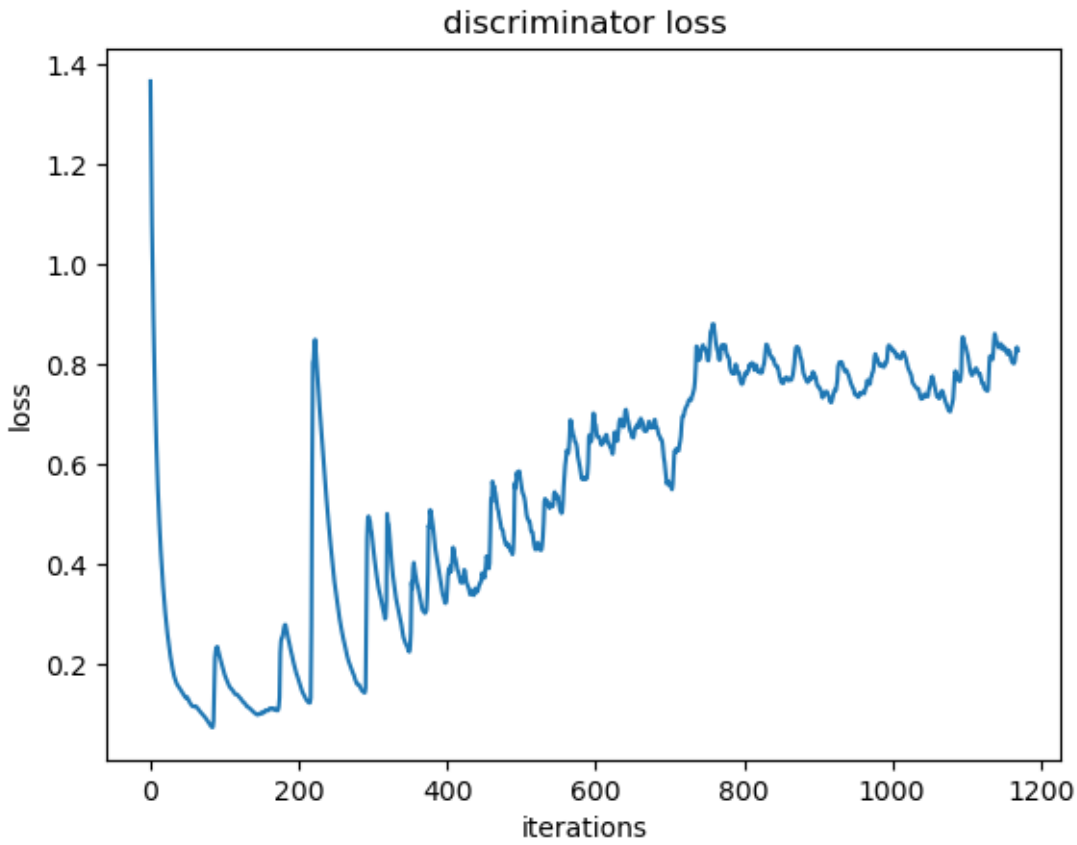

0
50
100
150
200
250
0
50
100
150
200
250

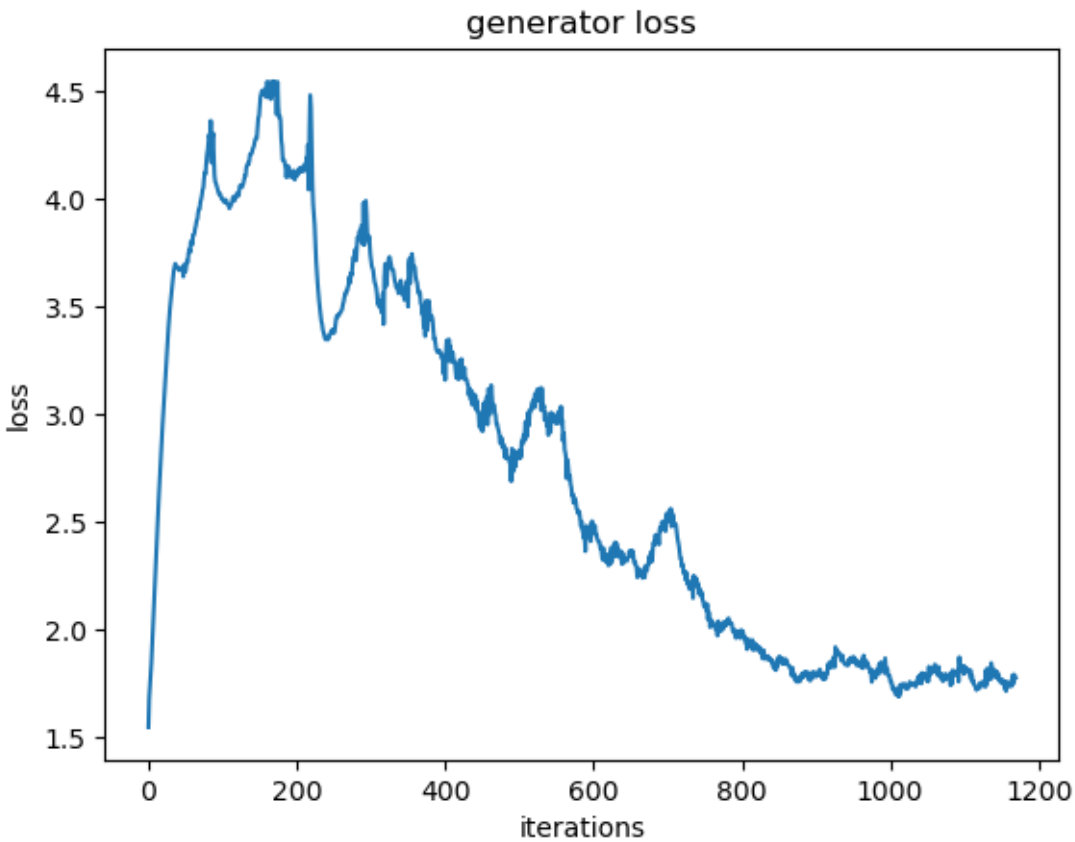

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
50
100
150
200
250
300
350
400
iterations




```
Iteration 400/9750: dis loss = 0.4201, gen loss = 4.9009
Iteration 500/9750: dis loss = 0.3627, gen loss = 3.1273
Iteration 600/9750: dis loss = 0.3206, gen loss = 1.8904
Iteration 700/9750: dis loss = 0.4730, gen loss = 2.9746
Iteration 800/9750: dis loss = 0.8437, gen loss = 2.7130
Iteration 900/9750: dis loss = 0.6580, gen loss = 1.9948
Iteration 1000/9750: dis loss = 0.7740, gen loss = 1.7977
Iteration 1100/9750: dis loss = 0.7661, gen loss = 1.0946
```

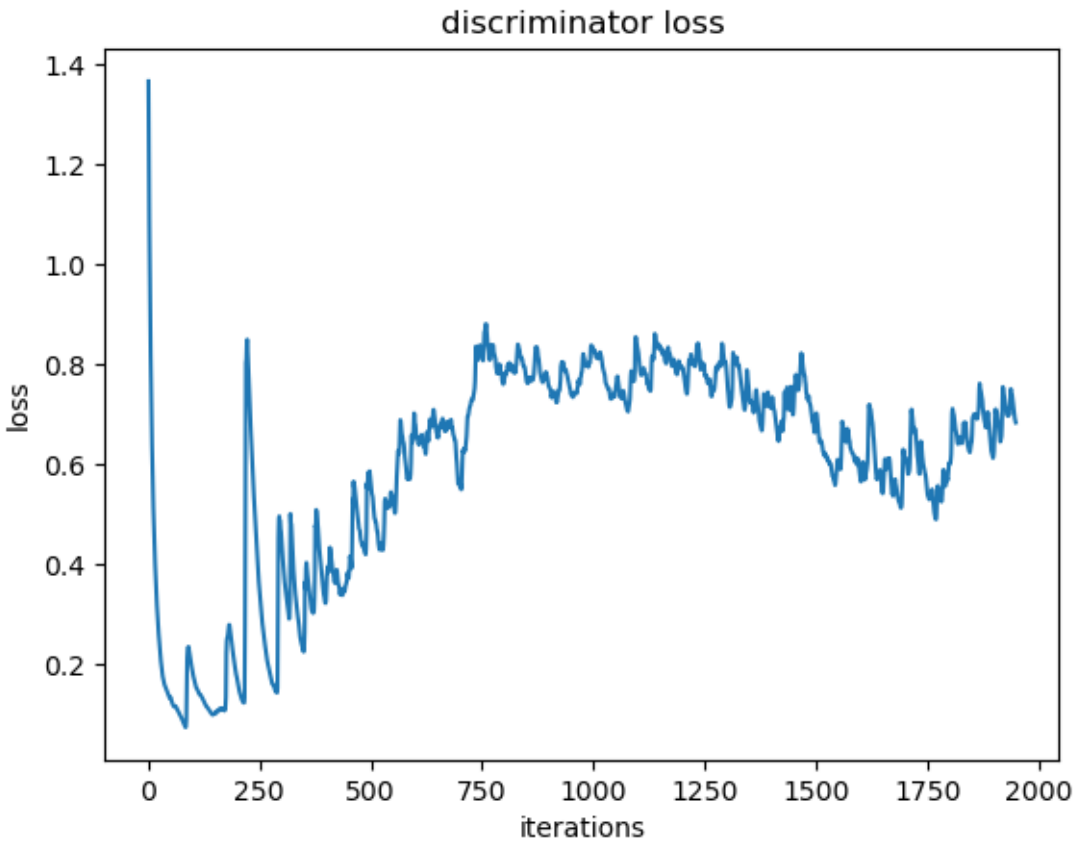

0
50
100
150
200
250
0
50
100
150
200
250

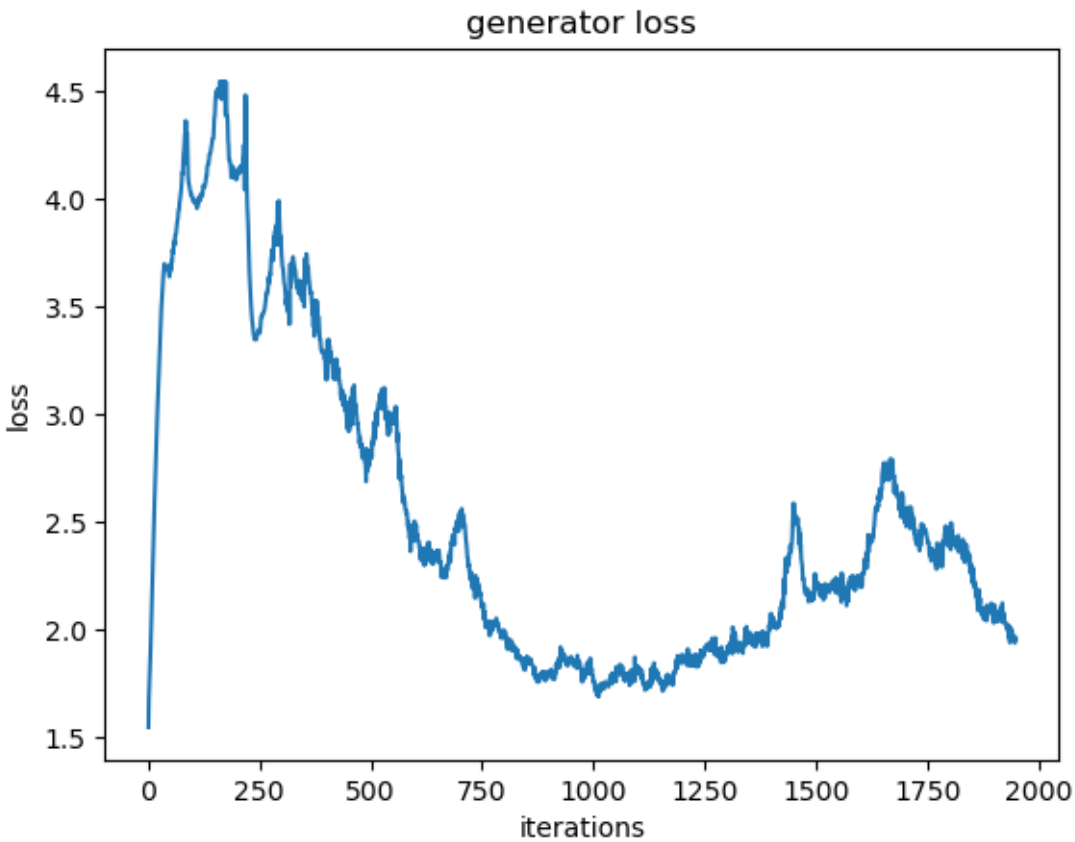

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
200
400
600
800
1000
1200
iterations




```
Iteration 1200/9750: dis loss = 0.7533, gen loss = 2.6387
Iteration 1300/9750: dis loss = 0.6104, gen loss = 1.6700
Iteration 1400/9750: dis loss = 0.6623, gen loss = 3.4033
Iteration 1500/9750: dis loss = 0.9435, gen loss = 1.2326
Iteration 1600/9750: dis loss = 0.3981, gen loss = 1.8059
Iteration 1700/9750: dis loss = 0.4727, gen loss = 2.3809
Iteration 1800/9750: dis loss = 0.8525, gen loss = 1.8101
Iteration 1900/9750: dis loss = 0.6892, gen loss = 2.9701
```

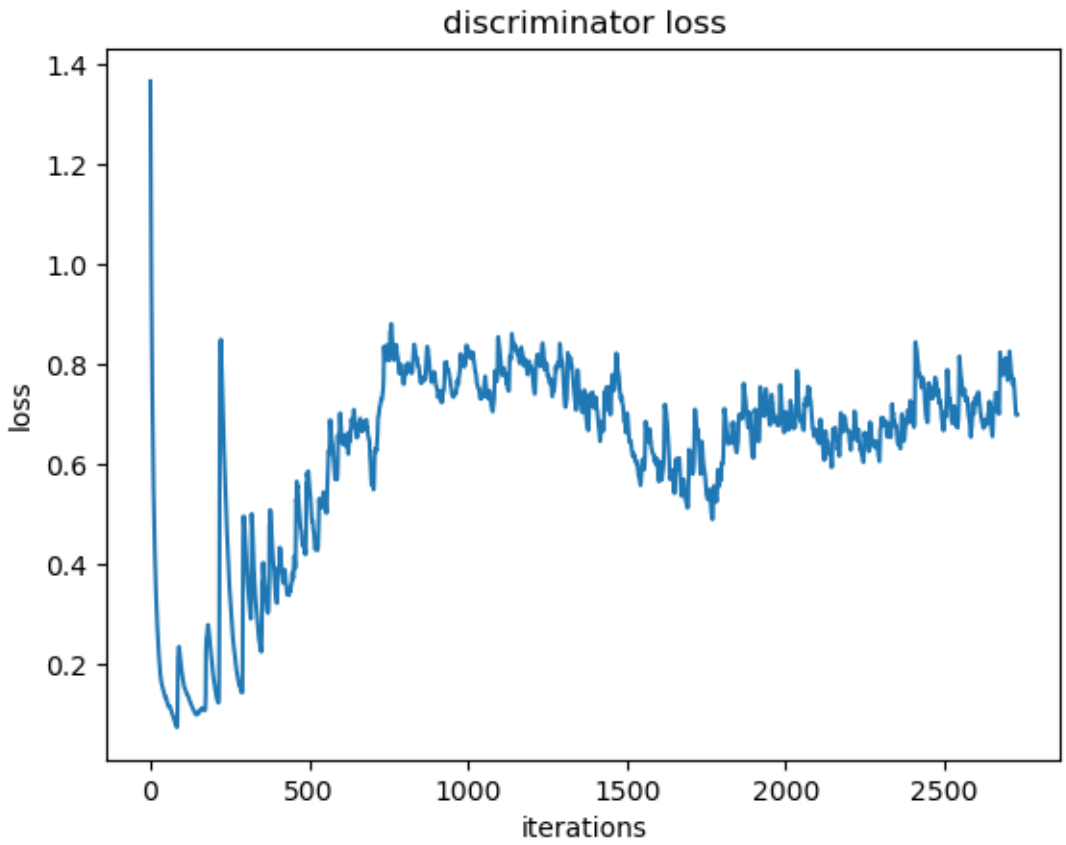
0
50
100
150
200
250
0
50
100
150
200
250

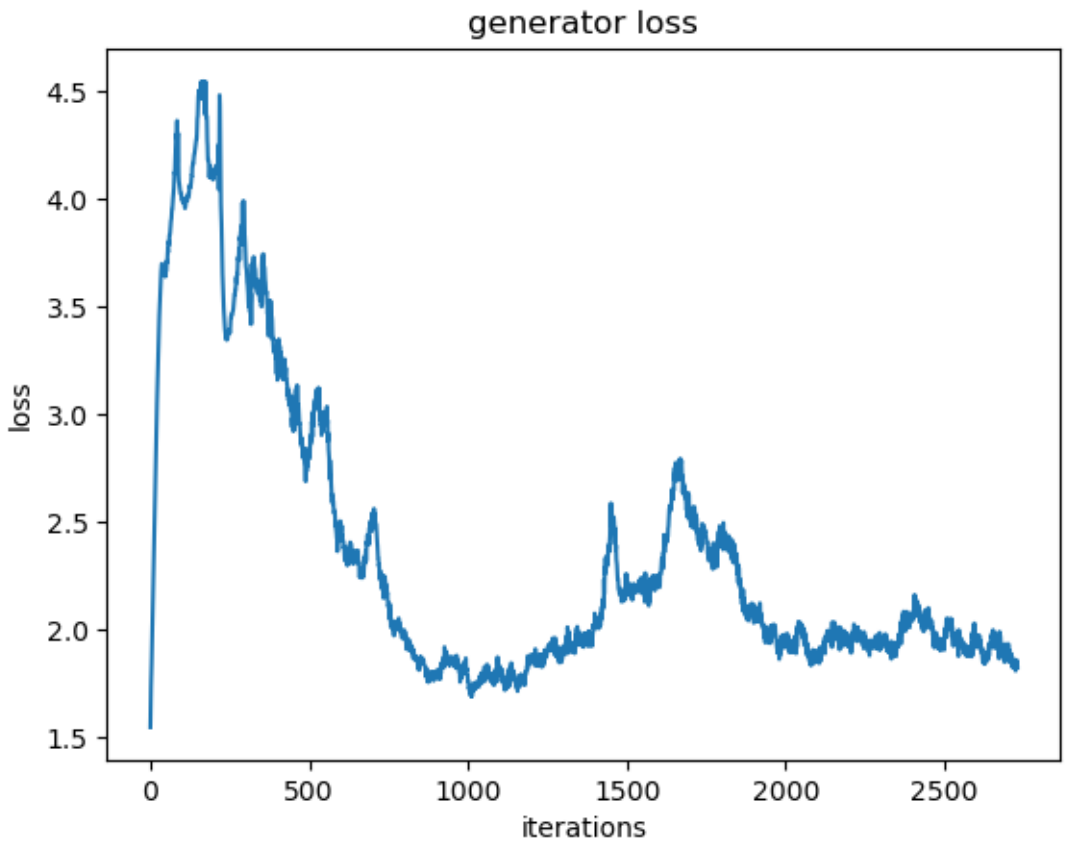

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
250
500
750
1000
1250
1500
1750
2000
iterations




```
Iteration 2000/9750: dis loss = 0.9632, gen loss = 1.1222
Iteration 2100/9750: dis loss = 0.5339, gen loss = 1.4646
Iteration 2200/9750: dis loss = 0.6069, gen loss = 2.0754
Iteration 2300/9750: dis loss = 0.7047, gen loss = 2.4057
Iteration 2400/9750: dis loss = 0.5630, gen loss = 1.7120
Iteration 2500/9750: dis loss = 0.8764, gen loss = 0.9114
Iteration 2600/9750: dis loss = 0.7129, gen loss = 1.9316
Iteration 2700/9750: dis loss = 0.6327, gen loss = 2.7733
```

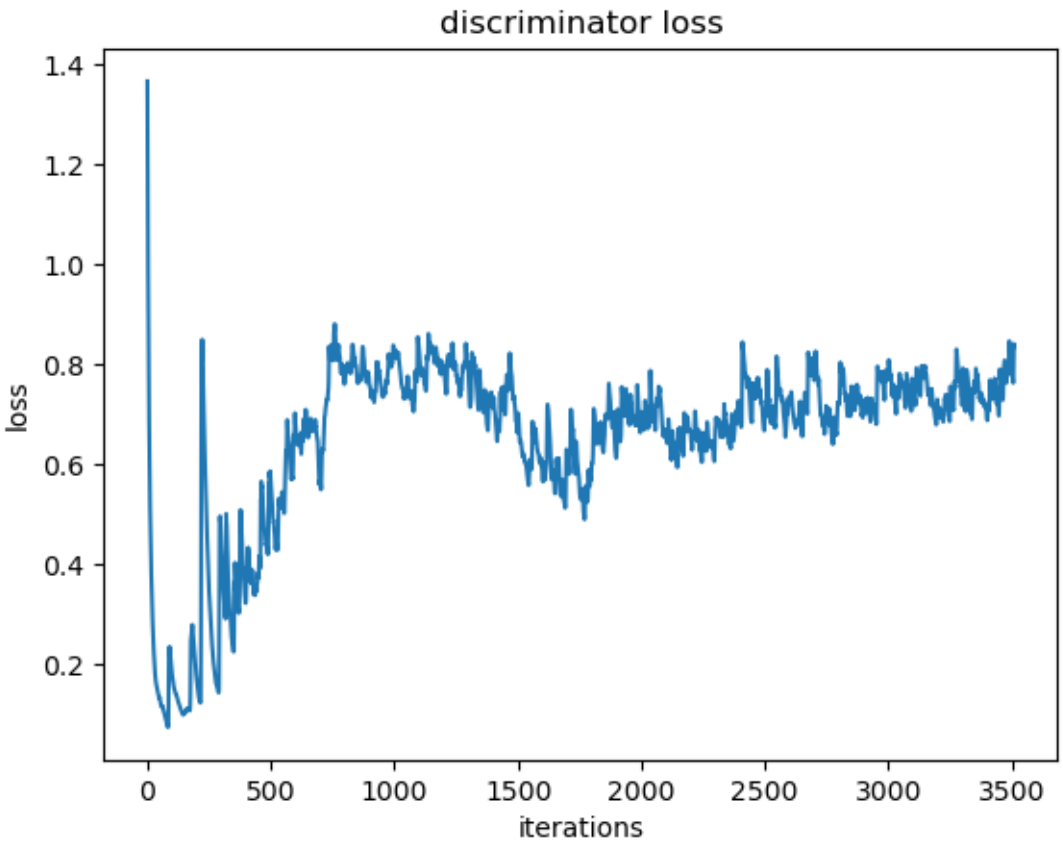
0
50
100
150
200
250
0
50
100
150
200
250

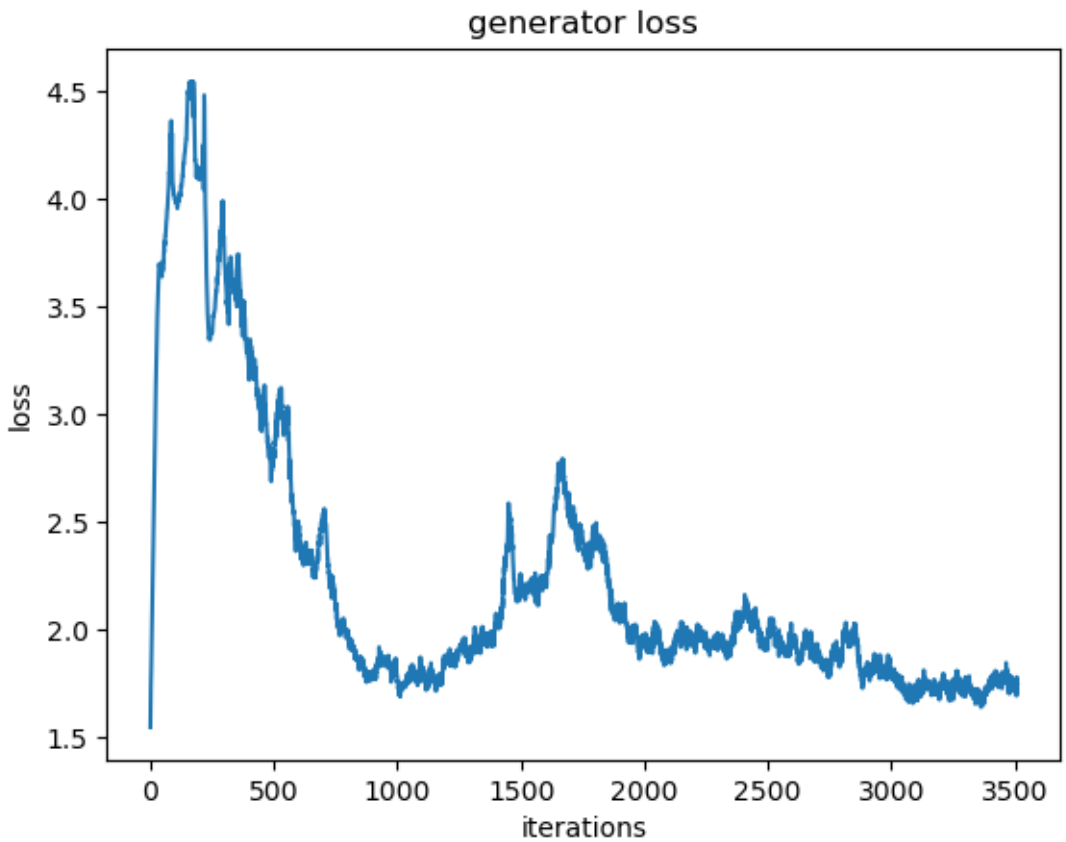

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
500
1000
1500
2000
2500
iterations




```
Iteration 2800/9750: dis loss = 0.6525, gen loss = 2.4392
Iteration 2900/9750: dis loss = 0.6591, gen loss = 2.4827
Iteration 3000/9750: dis loss = 0.8520, gen loss = 1.1340
Iteration 3100/9750: dis loss = 0.5582, gen loss = 1.7106
Iteration 3200/9750: dis loss = 0.5569, gen loss = 1.4218
Iteration 3300/9750: dis loss = 0.9672, gen loss = 0.4734
Iteration 3400/9750: dis loss = 0.6245, gen loss = 2.1532
Iteration 3500/9750: dis loss = 0.6490, gen loss = 2.2165
```

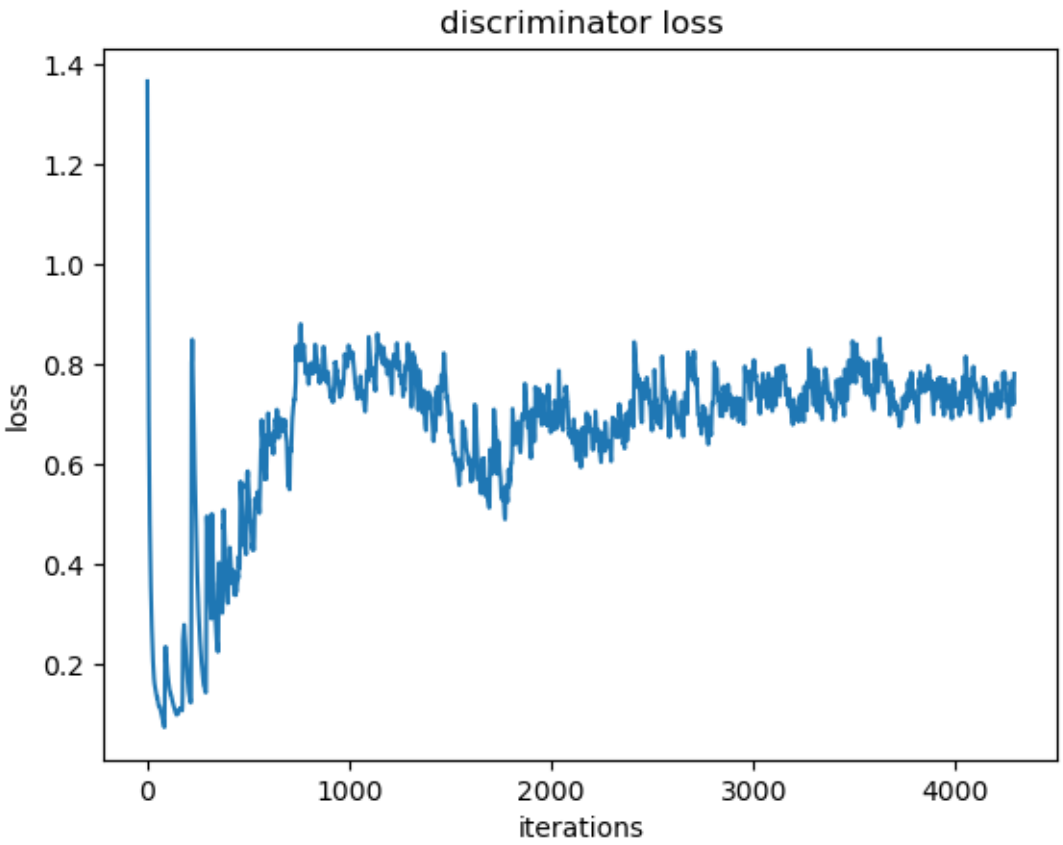
0
50
100
150
200
250
0
50
100
150
200
250

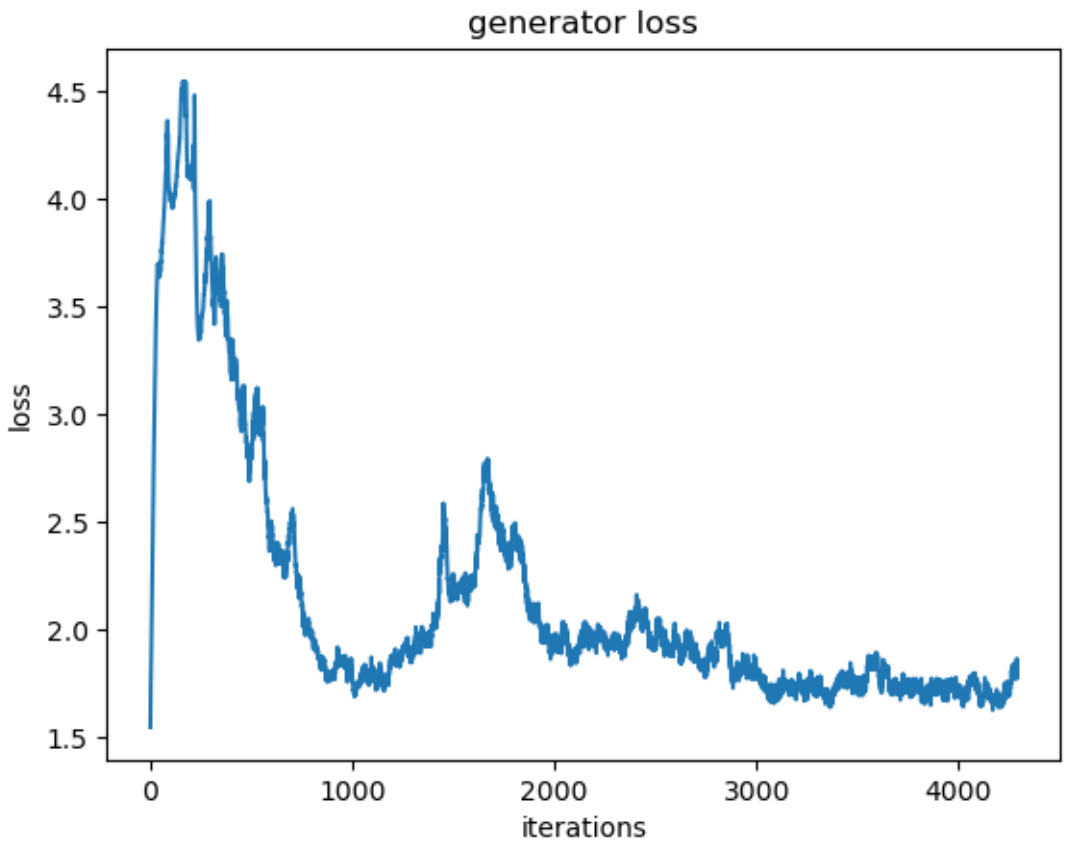

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
500
1000
1500
2000
2500
3000
3500
iterations




```
Iteration 3600/9750: dis loss = 1.1356, gen loss = 1.5110
Iteration 3700/9750: dis loss = 0.6126, gen loss = 1.2840
Iteration 3800/9750: dis loss = 0.7798, gen loss = 0.9167
Iteration 3900/9750: dis loss = 0.7425, gen loss = 1.3343
Iteration 4000/9750: dis loss = 0.6379, gen loss = 1.9903
Iteration 4100/9750: dis loss = 0.7016, gen loss = 1.5404
Iteration 4200/9750: dis loss = 0.8479, gen loss = 1.5989
```

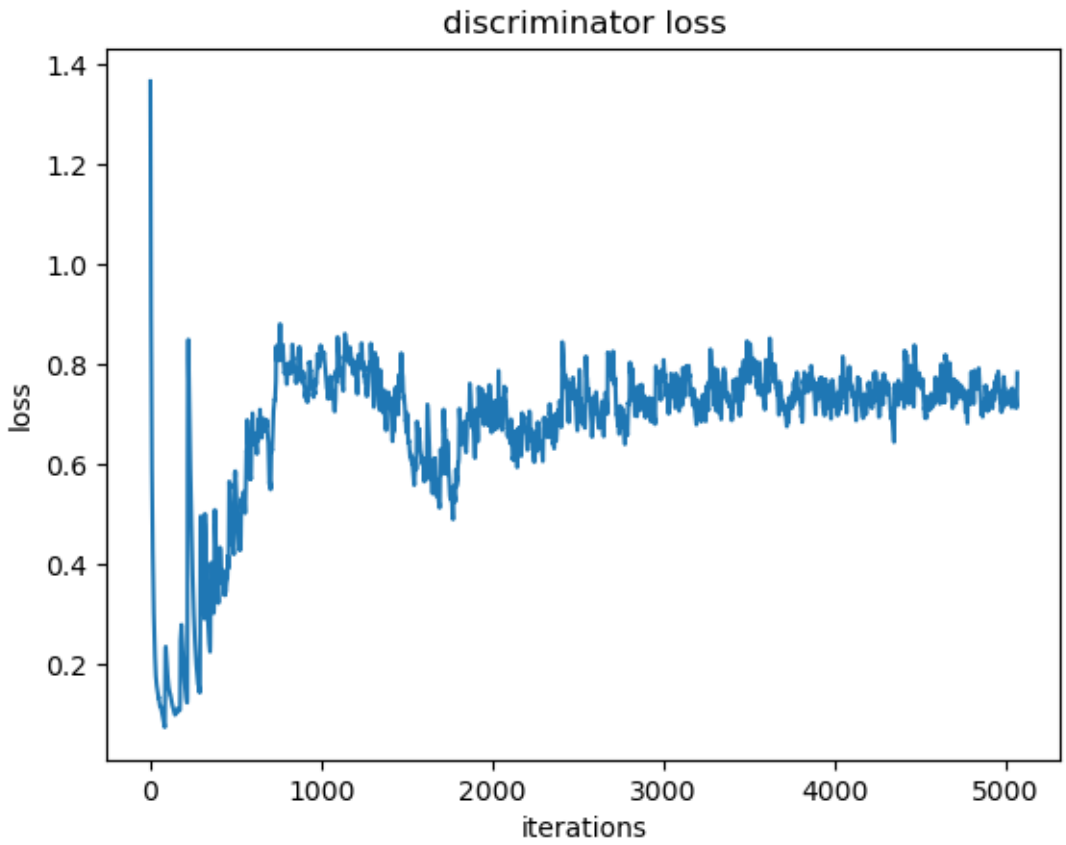
0
50
100
150
200
250
0
50
100
150
200
250

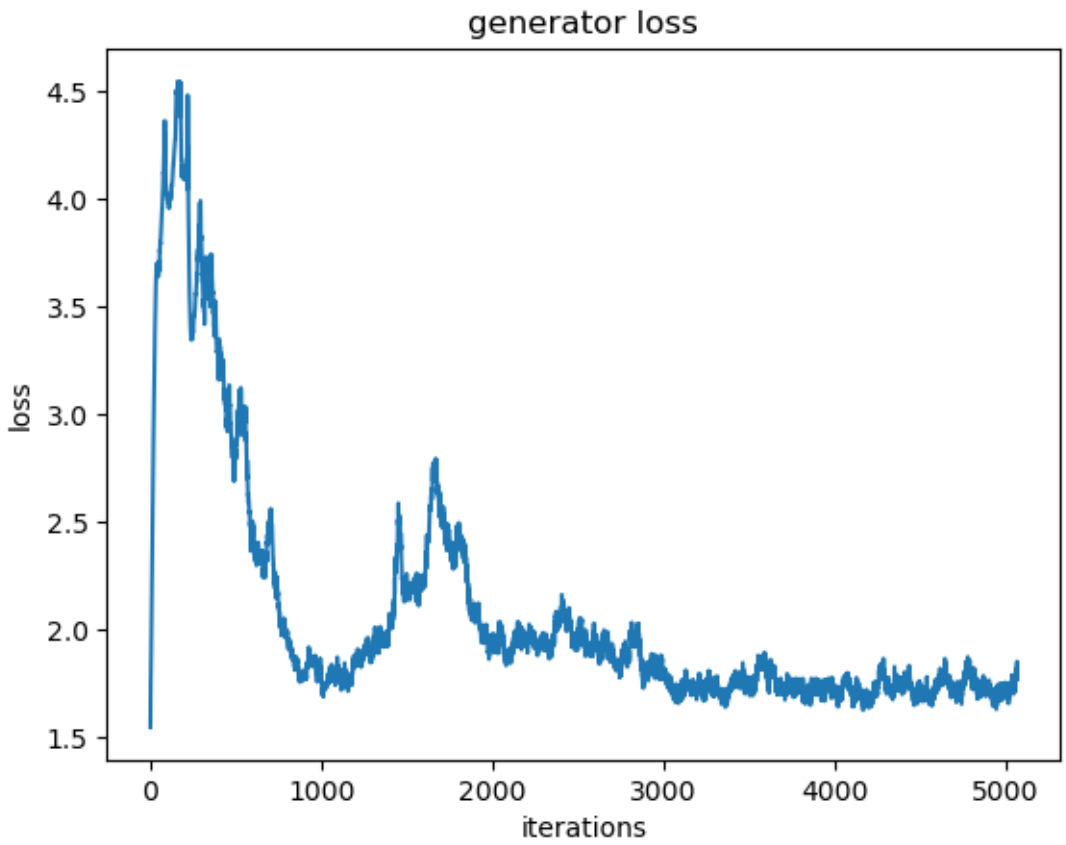

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
1000
2000
3000
4000
iterations




```
Iteration 4300/9750: dis loss = 0.7435, gen loss = 1.3522
Iteration 4400/9750: dis loss = 0.6006, gen loss = 1.6850
Iteration 4500/9750: dis loss = 0.6435, gen loss = 1.3588
Iteration 4600/9750: dis loss = 0.6635, gen loss = 1.0228
Iteration 4700/9750: dis loss = 0.9247, gen loss = 0.8145
Iteration 4800/9750: dis loss = 1.0331, gen loss = 1.1633
Iteration 4900/9750: dis loss = 0.7582, gen loss = 2.4590
Iteration 5000/9750: dis loss = 0.6881, gen loss = 1.0897
```

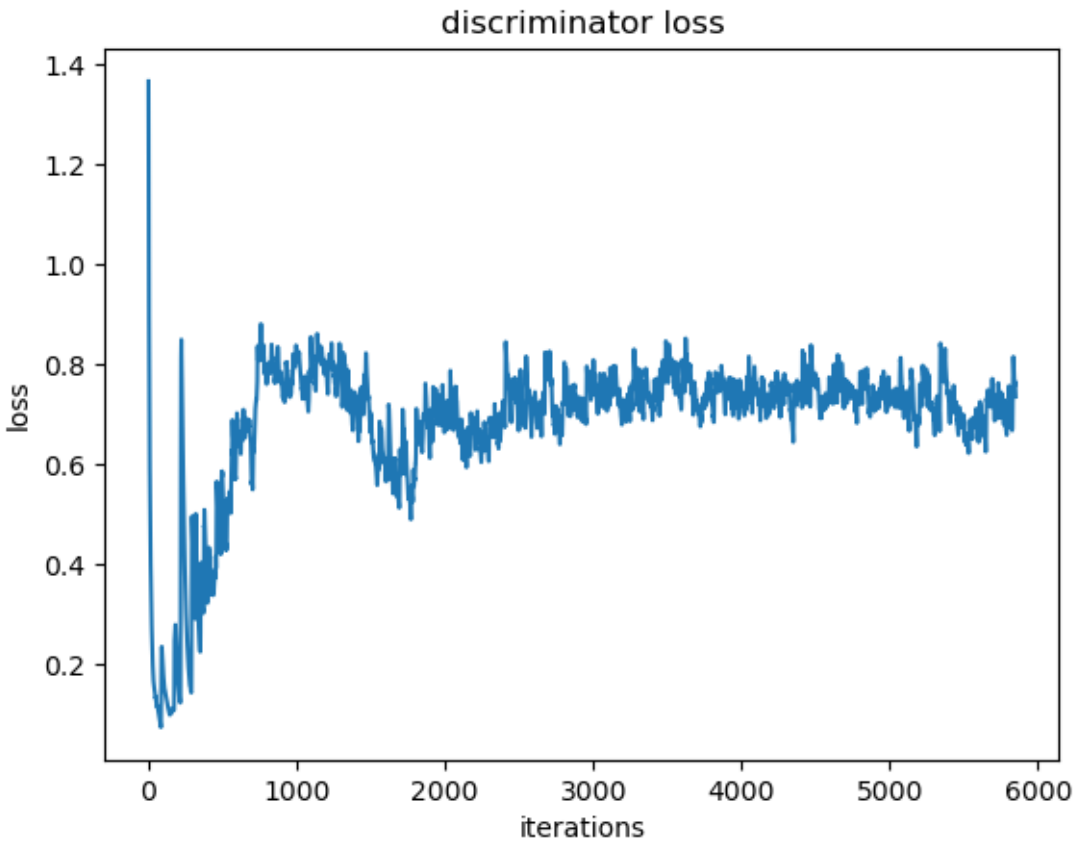
0
50
100
150
200
250
0
50
100
150
200
250

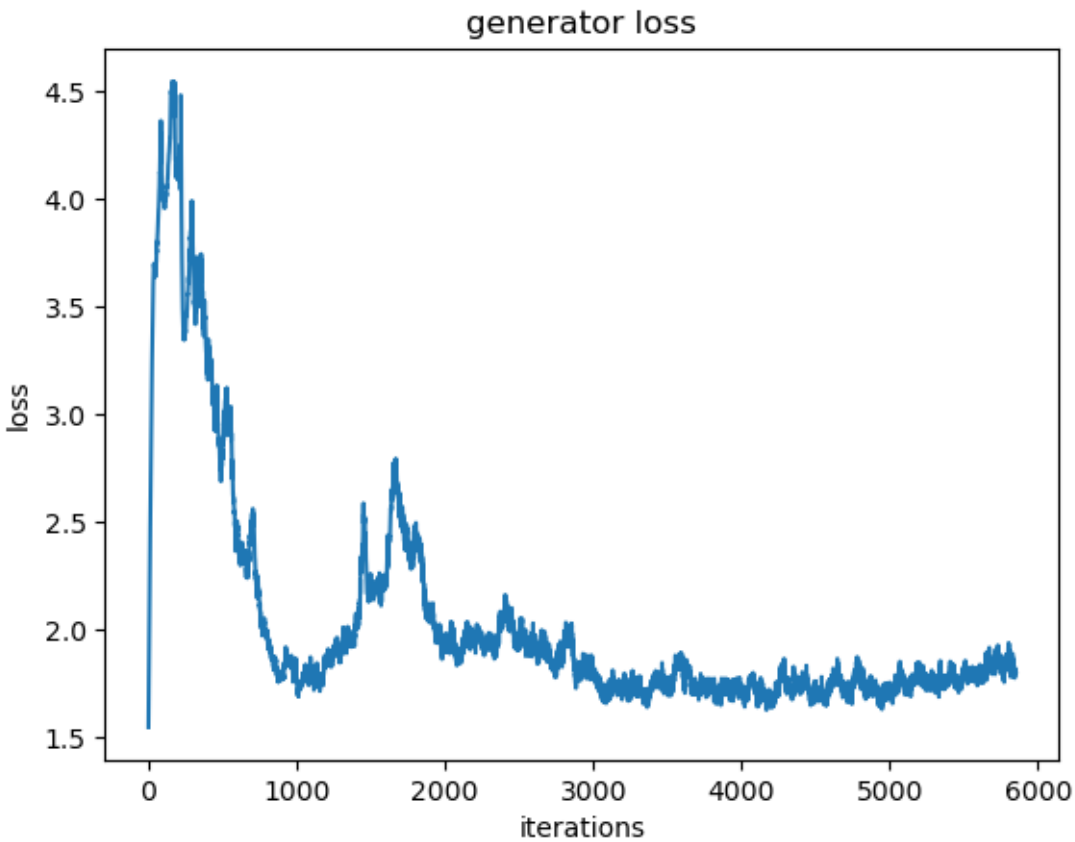

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
1000
2000
3000
4000
5000
iterations




```
Iteration 5100/9750: dis loss = 0.9298, gen loss = 1.2712
Iteration 5200/9750: dis loss = 0.9111, gen loss = 3.6606
Iteration 5300/9750: dis loss = 0.4911, gen loss = 1.9118
Iteration 5400/9750: dis loss = 0.7287, gen loss = 2.3671
Iteration 5500/9750: dis loss = 0.5889, gen loss = 1.5346
Iteration 5600/9750: dis loss = 0.6440, gen loss = 2.0682
Iteration 5700/9750: dis loss = 0.5658, gen loss = 1.9651
Iteration 5800/9750: dis loss = 0.5946, gen loss = 1.6667
```

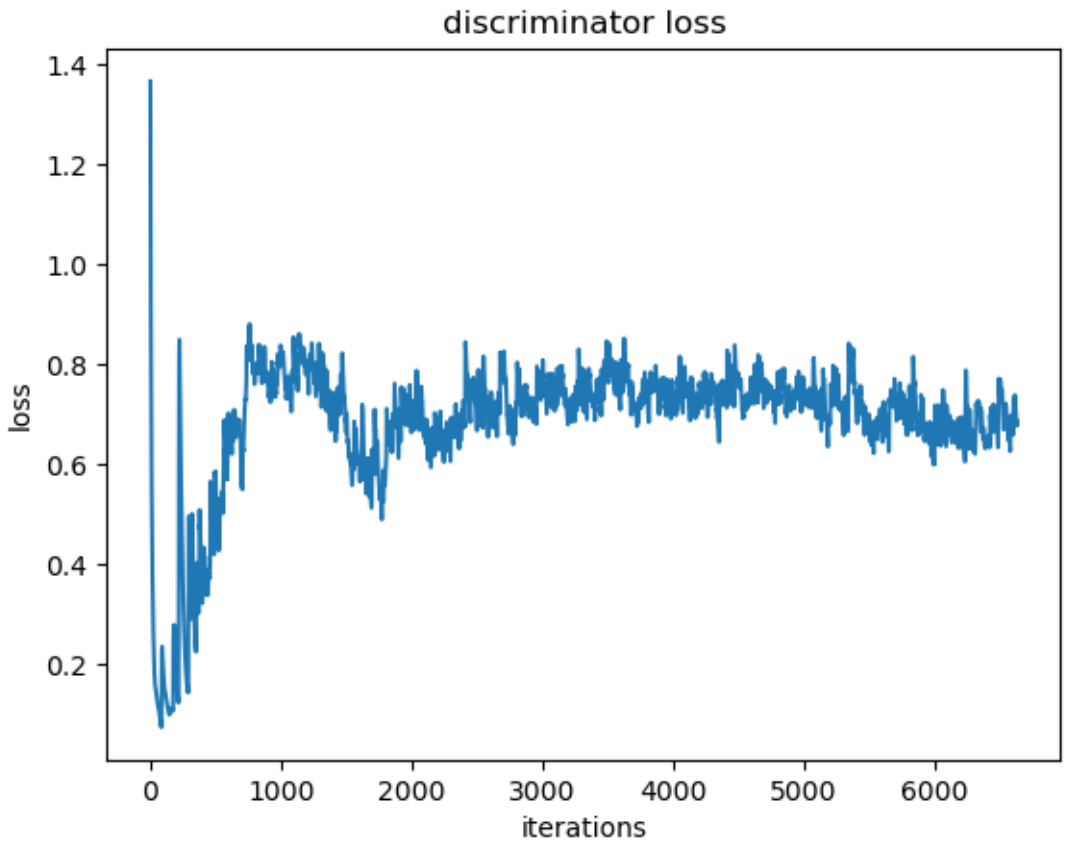
0
50
100
150
200
250
0
50
100
150
200
250

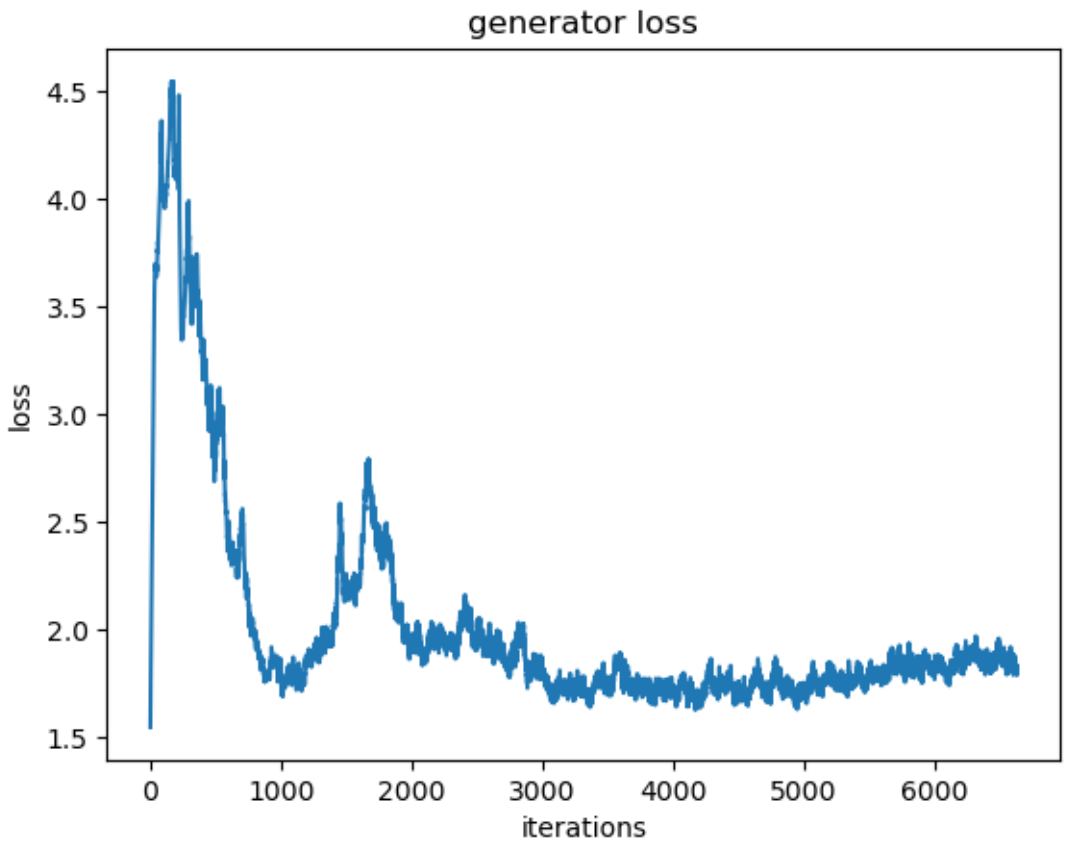

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
1000
2000
3000
4000
5000
6000
iterations



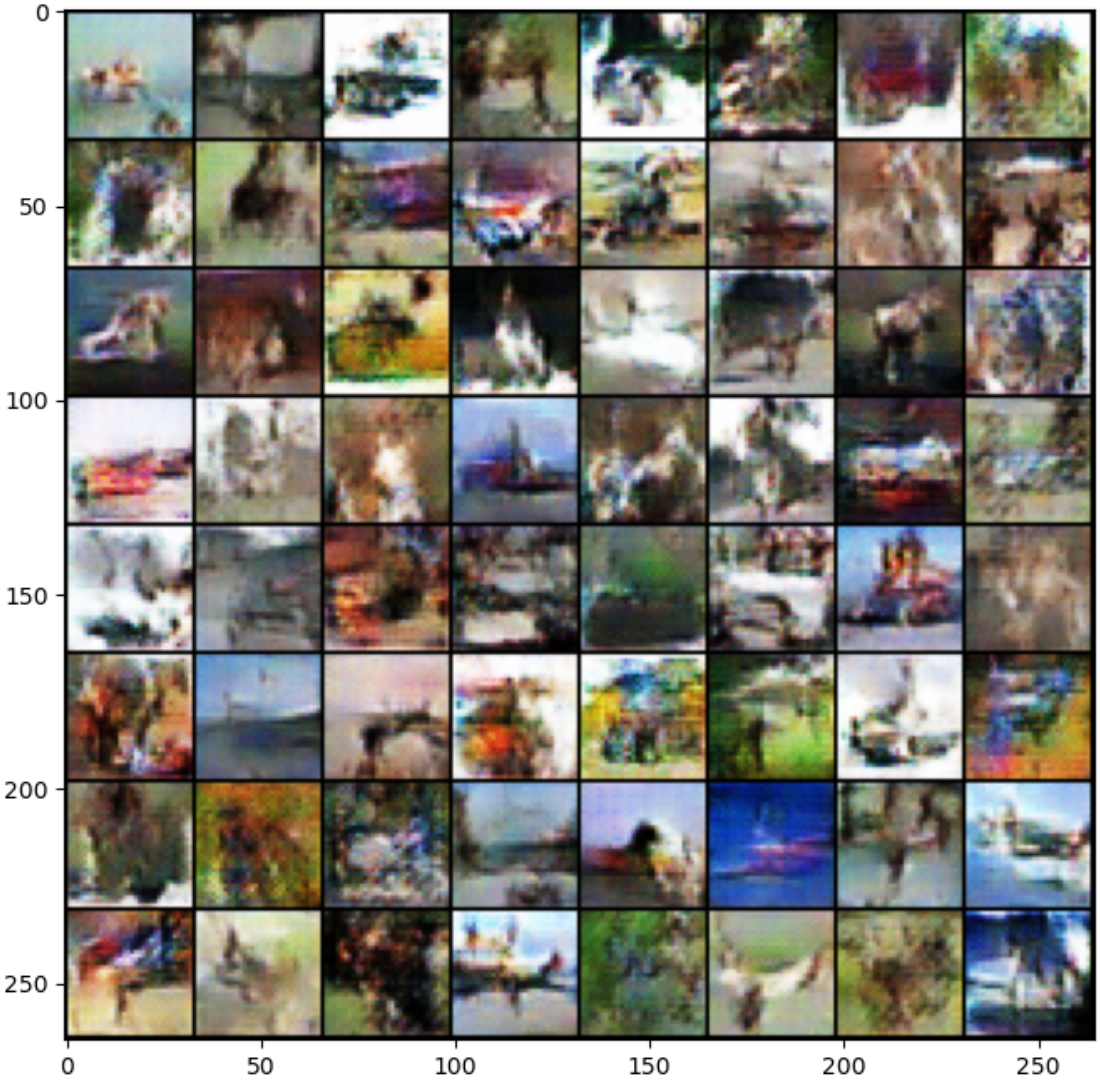

```
Iteration 5900/9750: dis loss = 0.7206, gen loss = 2.1457
Iteration 6000/9750: dis loss = 1.4574, gen loss = 1.1194
Iteration 6100/9750: dis loss = 0.6931, gen loss = 2.6316
Iteration 6200/9750: dis loss = 0.7694, gen loss = 0.7268
Iteration 6300/9750: dis loss = 0.5883, gen loss = 2.4080
Iteration 6400/9750: dis loss = 0.6651, gen loss = 0.9429
Iteration 6500/9750: dis loss = 0.5897, gen loss = 2.3501
Iteration 6600/9750: dis loss = 0.9876, gen loss = 0.3705
```

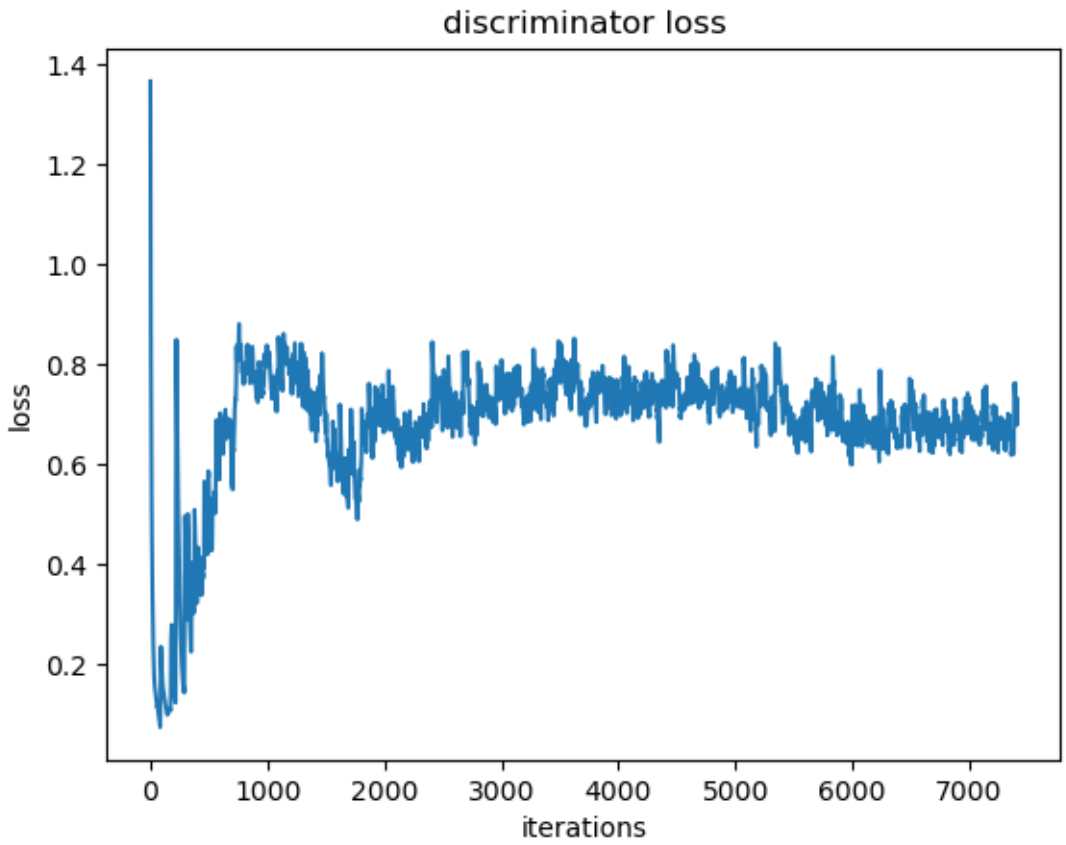
0
50
100
150
200
250
0
50
100
150
200
250

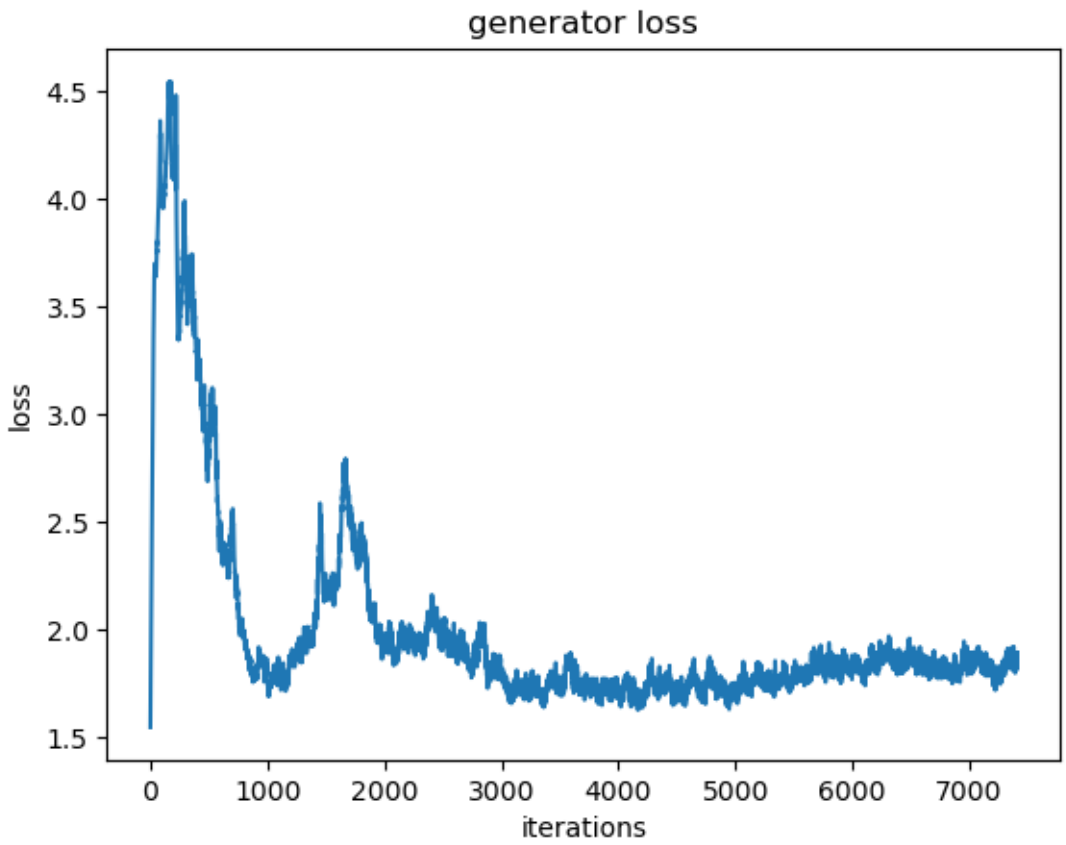

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
1000
2000
3000
4000
5000
6000
iterations




```
Iteration 6700/9750: dis loss = 1.0758, gen loss = 2.9864
Iteration 6800/9750: dis loss = 0.6972, gen loss = 2.4924
Iteration 6900/9750: dis loss = 0.5156, gen loss = 2.1346
Iteration 7000/9750: dis loss = 0.8084, gen loss = 1.2138
Iteration 7100/9750: dis loss = 0.6035, gen loss = 1.9511
Iteration 7200/9750: dis loss = 0.5336, gen loss = 1.7180
Iteration 7300/9750: dis loss = 0.5229, gen loss = 1.5761
Iteration 7400/9750: dis loss = 0.5648, gen loss = 2.7774
```

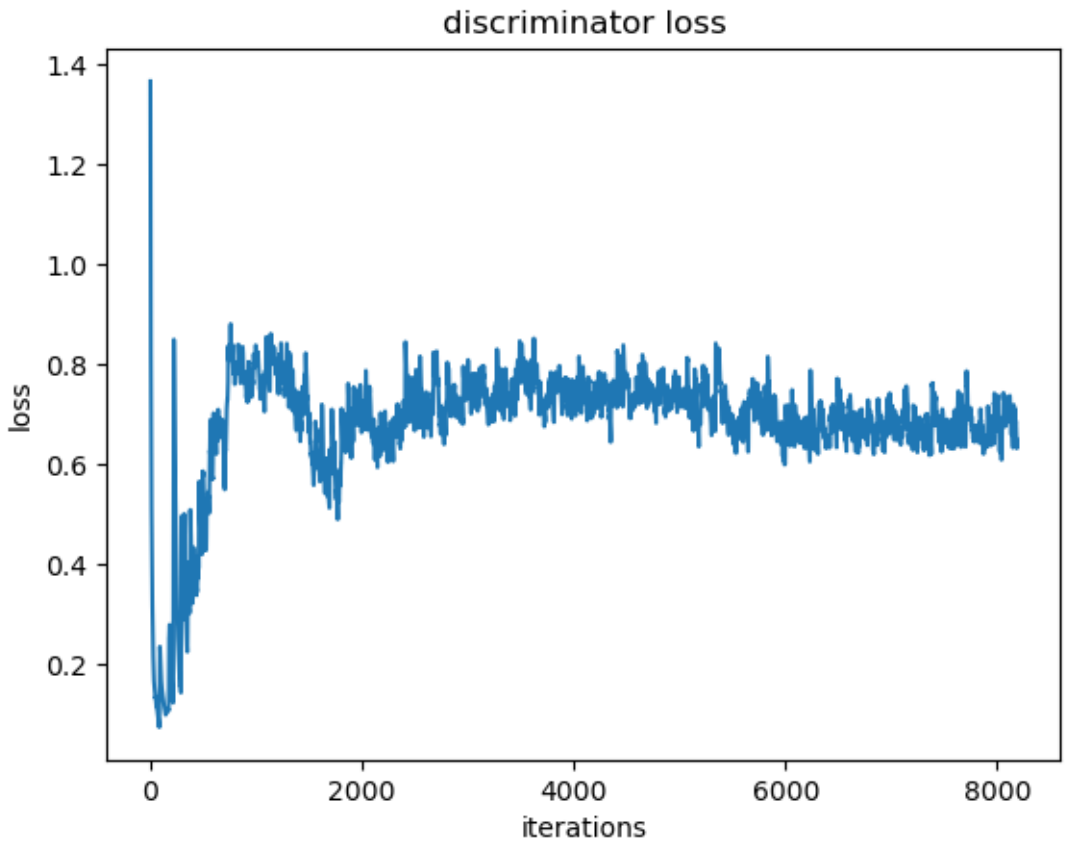

0
50
100
150
200
250
0
50
100
150
200
250

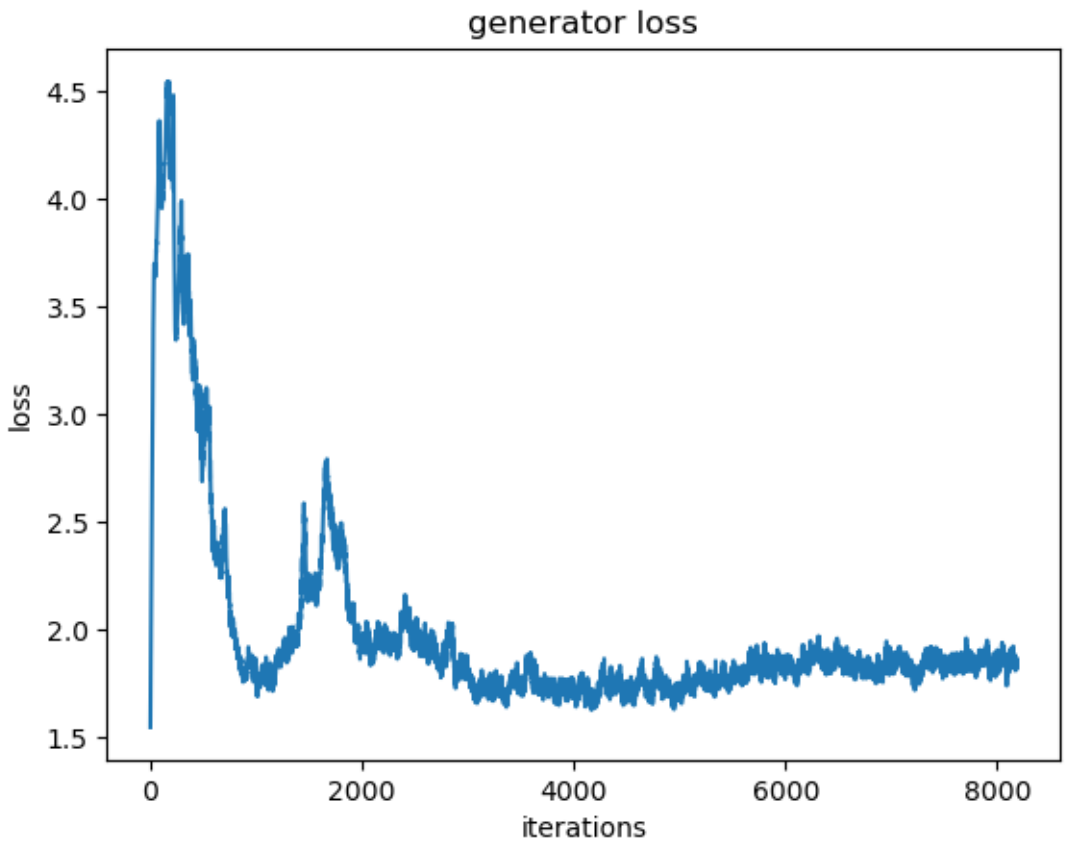

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
1000
2000
3000
4000
5000
6000
7000
iterations




```
Iteration 7500/9750: dis loss = 0.6031, gen loss = 2.3721
Iteration 7600/9750: dis loss = 0.5844, gen loss = 2.0099
Iteration 7700/9750: dis loss = 0.6195, gen loss = 2.0355
Iteration 7800/9750: dis loss = 0.7913, gen loss = 0.9930
Iteration 7900/9750: dis loss = 0.7127, gen loss = 1.3278
Iteration 8000/9750: dis loss = 0.7037, gen loss = 2.9945
Iteration 8100/9750: dis loss = 0.6376, gen loss = 1.3951
```

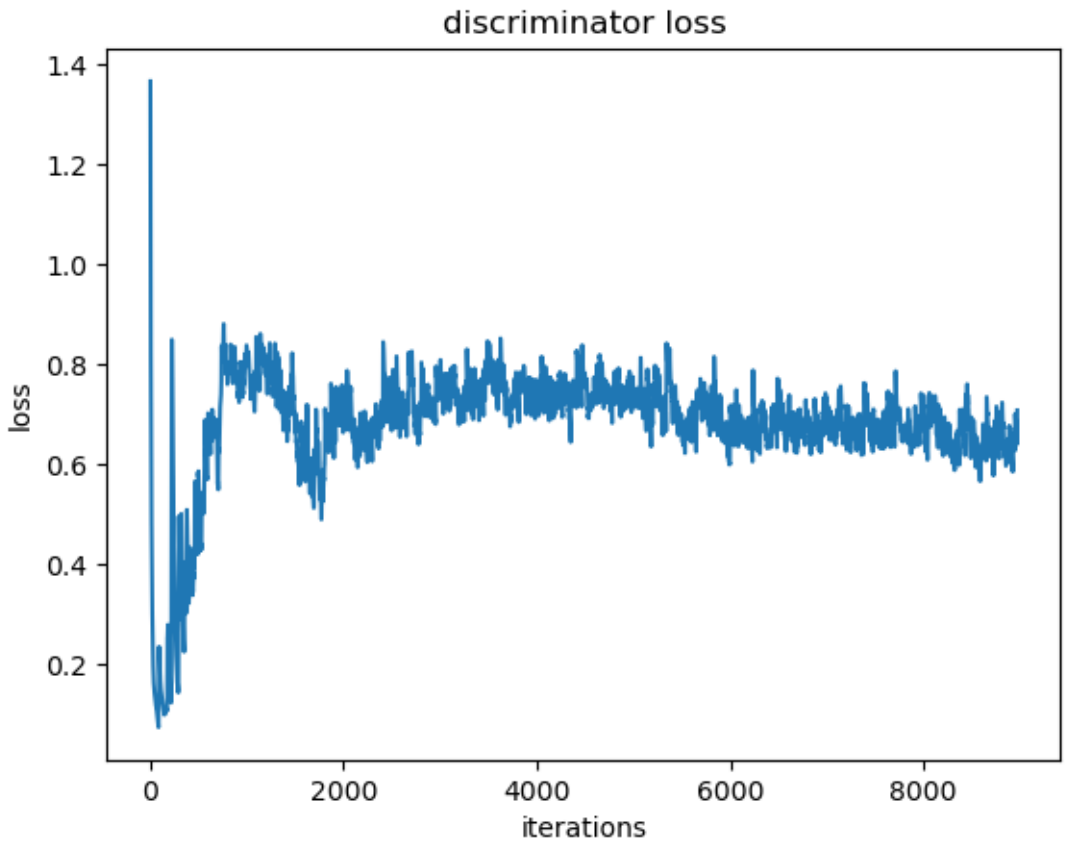
0
50
100
150
200
250
0
50
100
150
200
250

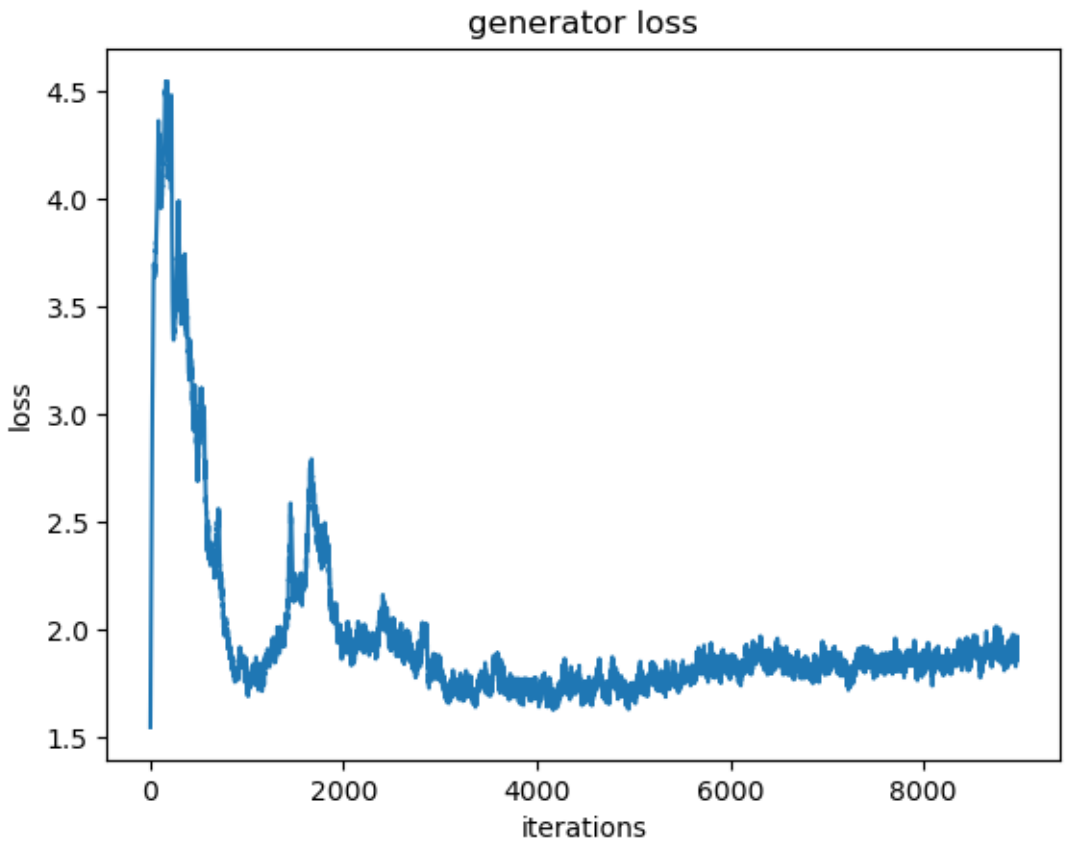

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
2000
4000
6000
8000
iterations




```
Iteration 8200/9750: dis loss = 0.7468, gen loss = 1.5470
Iteration 8300/9750: dis loss = 0.8666, gen loss = 2.5558
Iteration 8400/9750: dis loss = 0.6076, gen loss = 2.7846
Iteration 8500/9750: dis loss = 0.6763, gen loss = 2.6027
Iteration 8600/9750: dis loss = 0.5359, gen loss = 1.8483
Iteration 8700/9750: dis loss = 0.5032, gen loss = 1.9173
Iteration 8800/9750: dis loss = 0.6597, gen loss = 1.4945
Iteration 8900/9750: dis loss = 0.5086, gen loss = 1.5857
```

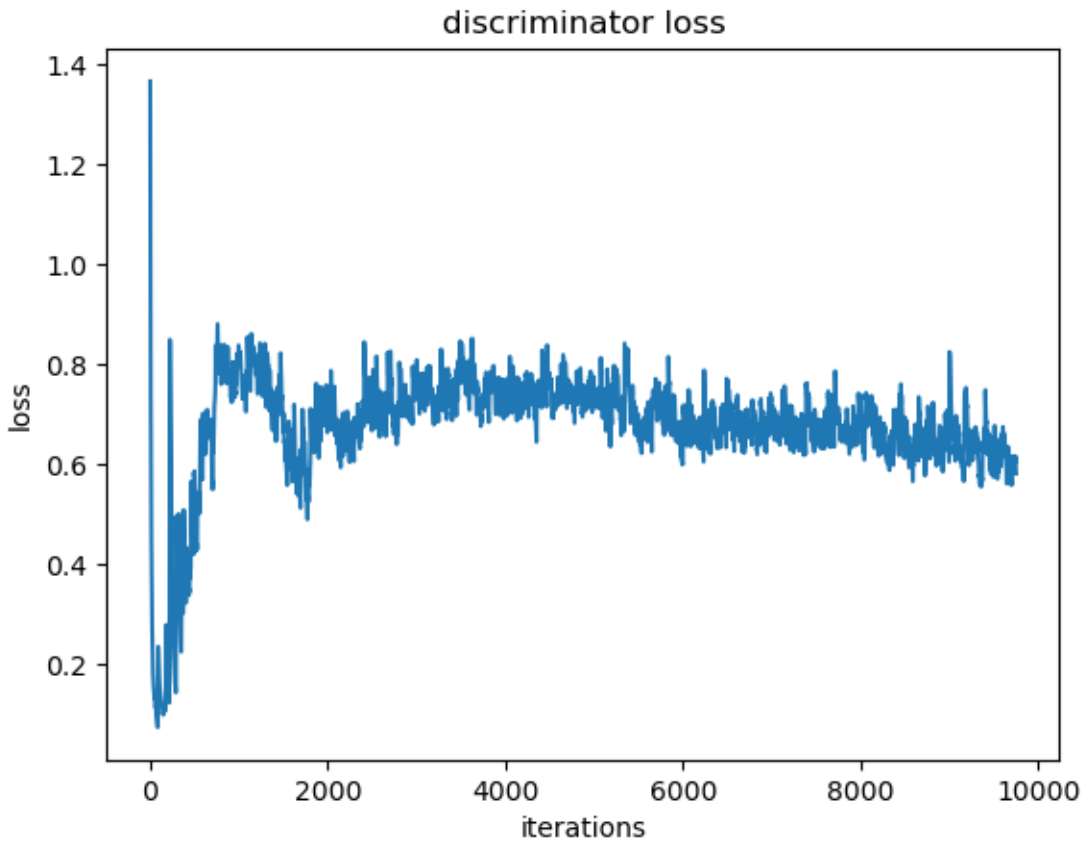
0
50
100
150
200
250
0
50
100
150
200
250

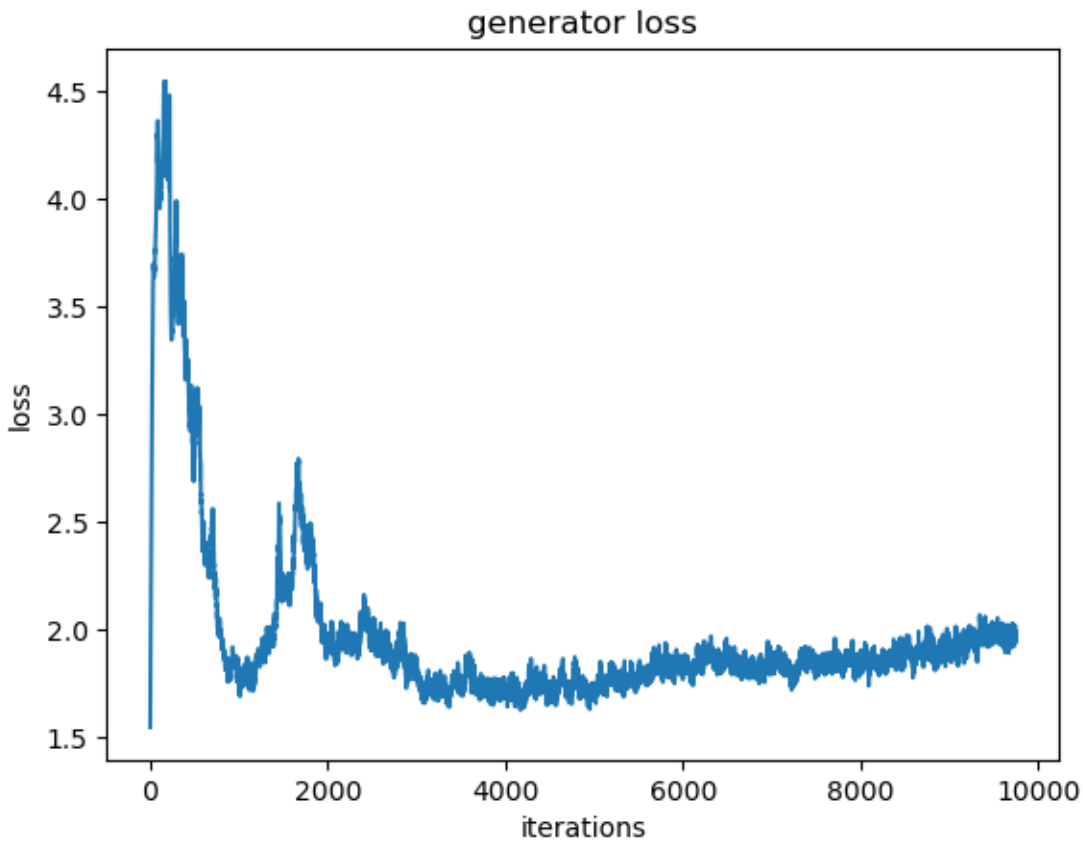

discriminator loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
loss
0
2000
4000
6000
8000
iterations




```
Iteration 9000/9750: dis loss = 1.2977, gen loss = 3.6233
Iteration 9100/9750: dis loss = 0.6815, gen loss = 1.0636
Iteration 9200/9750: dis loss = 0.5618, gen loss = 1.7638
Iteration 9300/9750: dis loss = 0.6286, gen loss = 1.6219
Iteration 9400/9750: dis loss = 0.7615, gen loss = 0.4826
Iteration 9500/9750: dis loss = 0.6816, gen loss = 2.9432
Iteration 9600/9750: dis loss = 0.8113, gen loss = 1.8912
Iteration 9700/9750: dis loss = 0.6045, gen loss = 0.5056
```

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


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




```
… Done!
```

6 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:

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

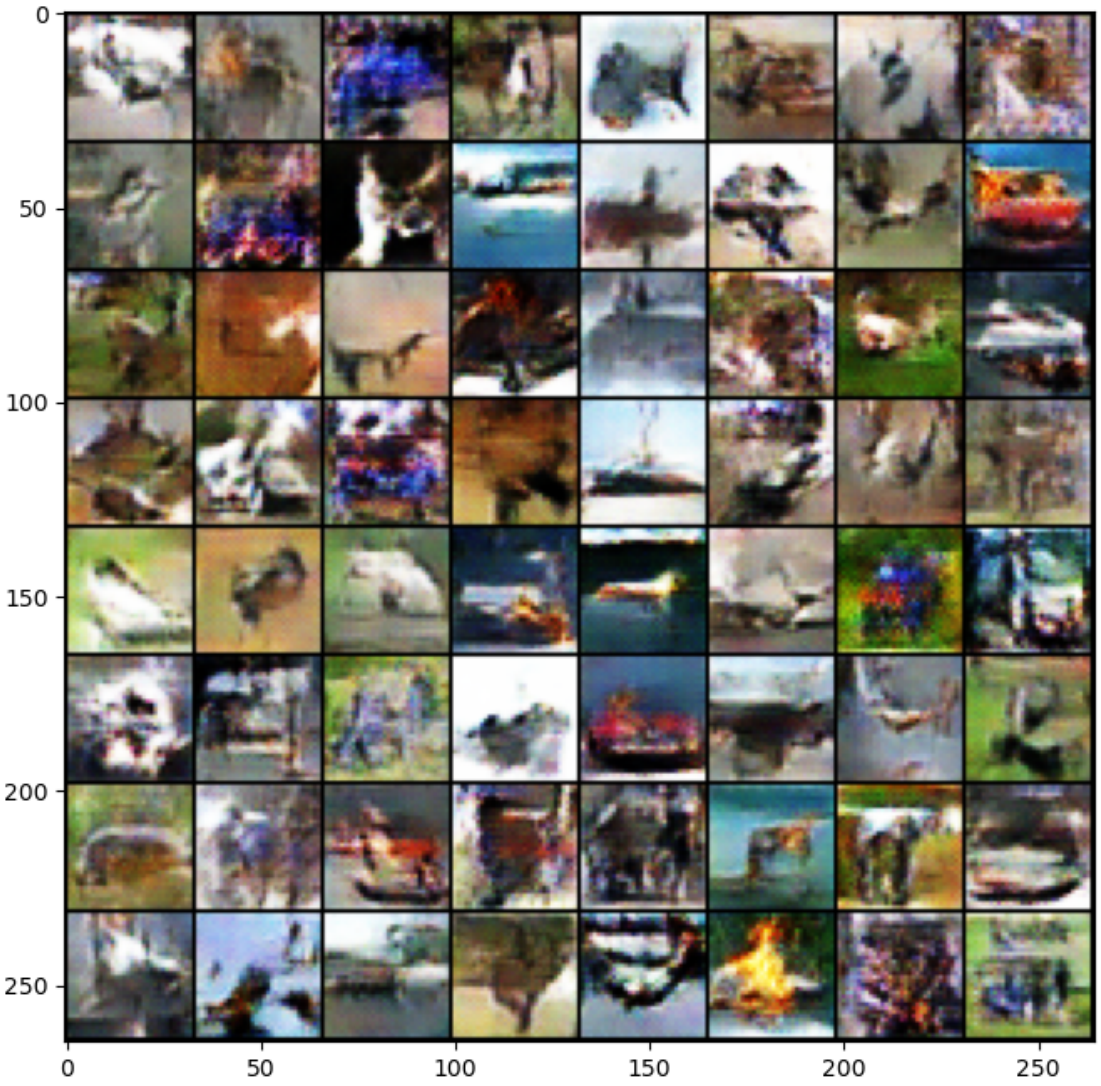
The function to minimize is the same, with the difference being that when training the network we fix a set of input data and find the optimal model parameters, while in activation maximization we fix the model parameters and find the optimal input.

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:

[68]:
```
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:

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

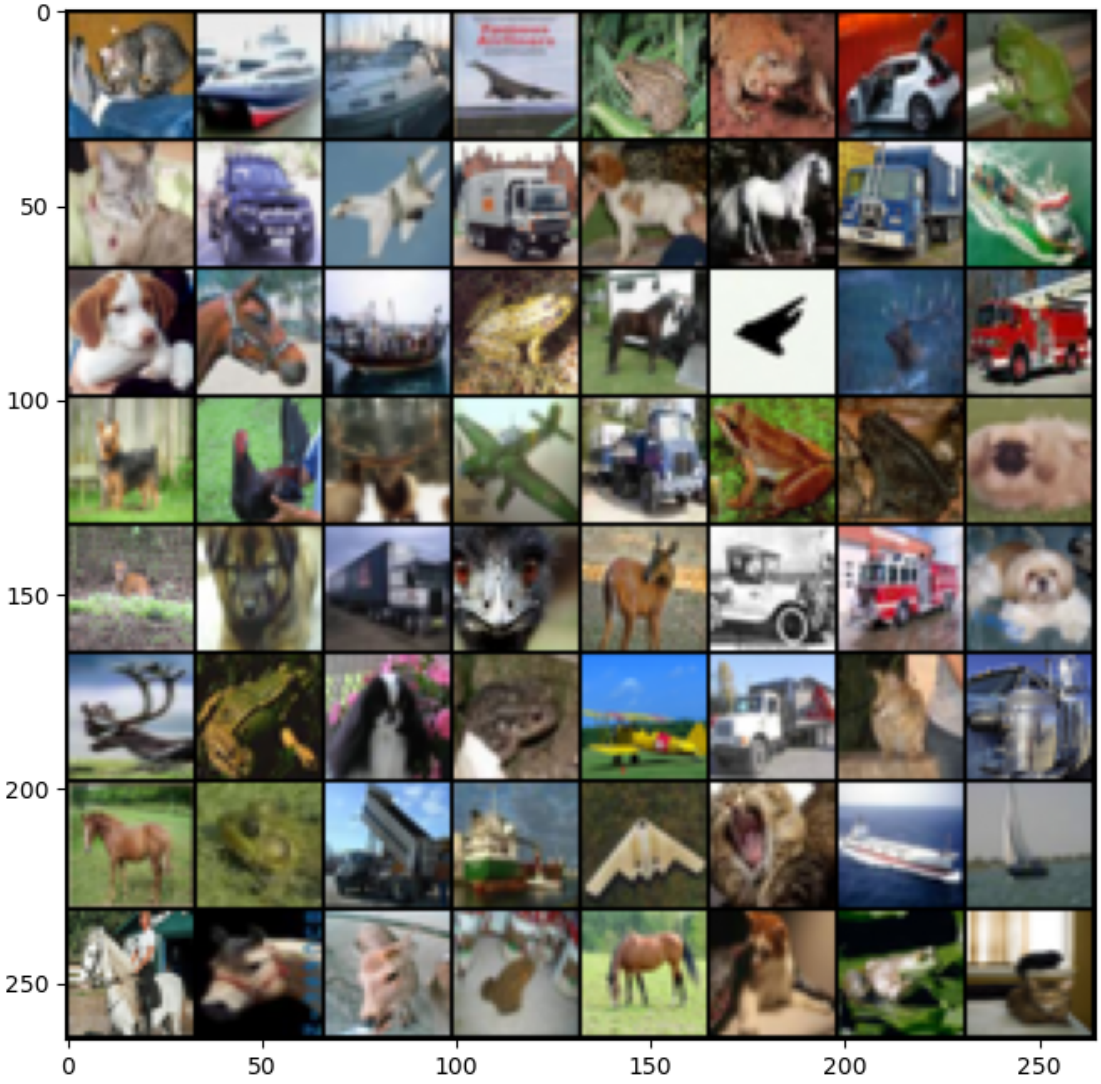
This time, we always start from a zero vector.

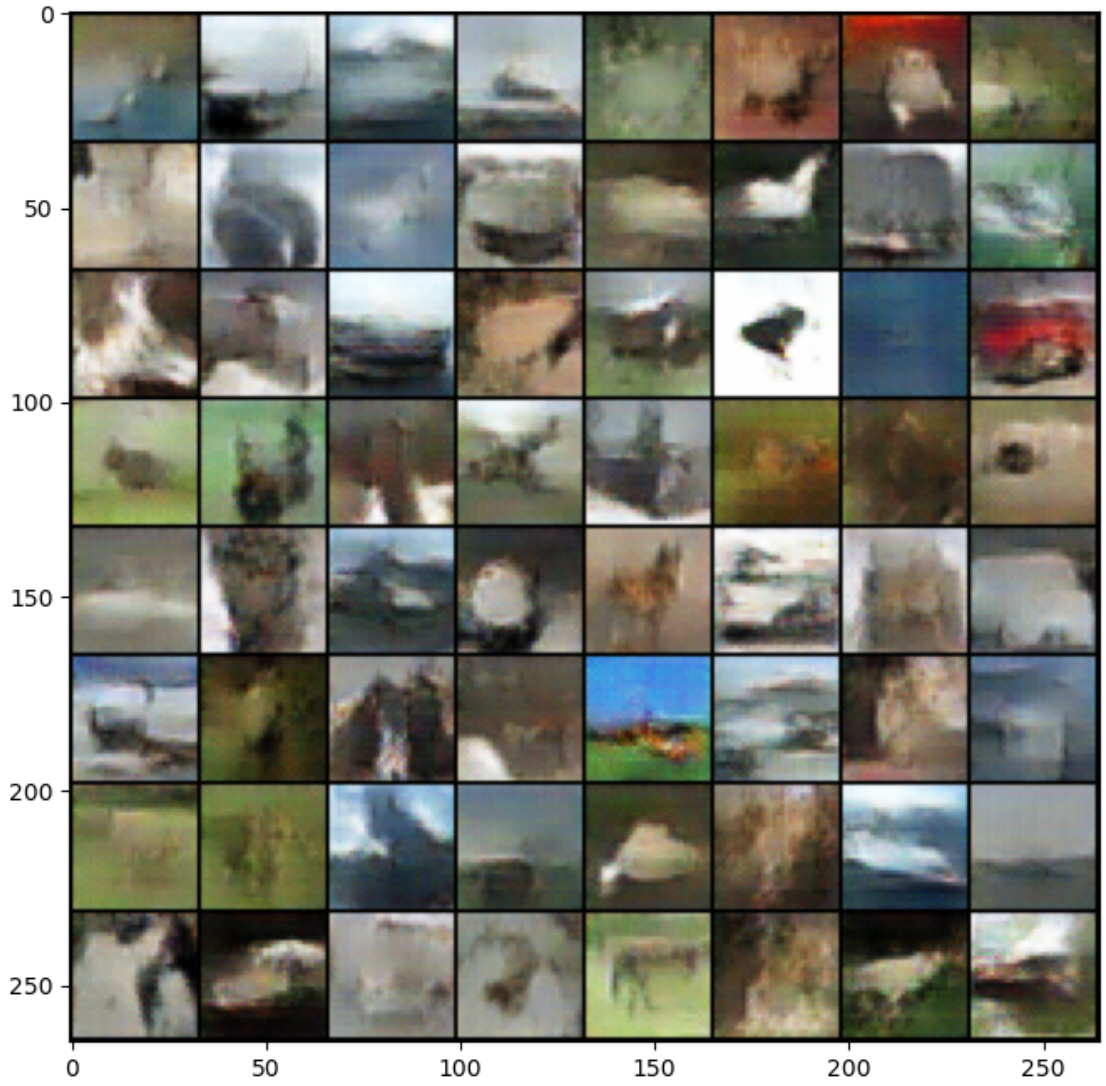
6.1 For this part, you need to complete code blocks marked with "Prob 2-4" above. Then run the following block.

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

[69]:
```
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 = 0.0137
```


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



7 Submission Instruction

See the pinned Piazza post for detailed instruction.

```
[ ]:
```