Problem1_VAE_v0

March 20, 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.

[76]:
```
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
```

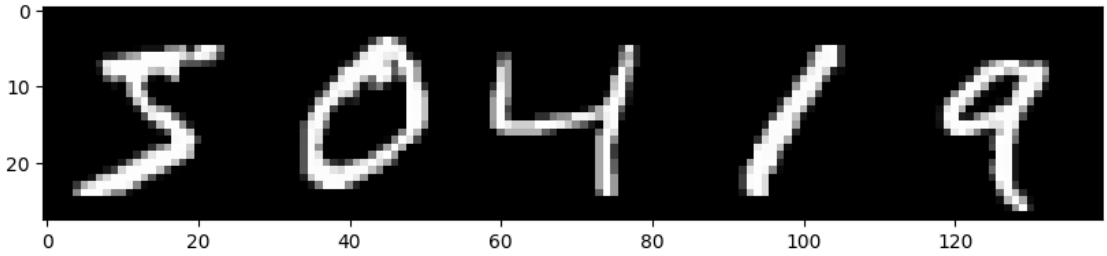
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[77]:
```
import torch
import torchvision
if torch.cuda.is_available():
    dev = 'cuda:0'
# elif torch.backends.mps.is_available():
#     dev = 'mps'
else:
    dev = 'cpu'
device = torch.device(dev)
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!
```

[78]:
```
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]

[93]:
```
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(in_features=self.input_size, out_features=256),
        nn.ReLU(),
        nn.Linear(in_features=256, out_features=64),
        nn.ReLU(),
        nn.Linear(in_features=64, out_features=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(in_features=nz, out_features=64),
        nn.ReLU(),
        nn.Linear(in_features=64, out_features=256),
        nn.ReLU(),
        nn.Linear(in_features=256, out_features=self.output_size),
        nn.Sigmoid()
    )
    ################################## END TODO␣
↪#########################################

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

3.2 Testing the Auto-Encoder Forward Pass

[94]:
```
# 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.

[95]:
```
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.

[99]:
```
# 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.        ␣
 ↪    #
################################################################################
optimizer = torch.optim.Adam(ae_model.parameters(), lr=learning_rate)
# criterion = nn.BCEWithLogitsLoss()
```

```
criterion = nn.BCELoss()

################################### 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.   #
 ␣
 ↪#############################################################################################
  for sample_img, _ in mnist_data_loader:
    sample_img = sample_img.reshape([batch_size, in_size])
    sample_img = sample_img.to(device)
    output = ae_model(sample_img)
    output= output.reshape([output.shape[0], output.shape[2]])
    optimizer.zero_grad()
    rec_loss = criterion(output, sample_img)
    rec_loss.backward()
    optimizer.step()

    if train_it % 100 == 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: 0.69235759973526
It 100: Reconstruction Loss: 0.2707238495349884
It 200: Reconstruction Loss: 0.22942639887332916
It 300: Reconstruction Loss: 0.2040729969739914
```

```
It 400: Reconstruction Loss: 0.15788783133029938
It 500: Reconstruction Loss: 0.14240673184394836
It 600: Reconstruction Loss: 0.14852213859558105
It 700: Reconstruction Loss: 0.13940536975860596
It 800: Reconstruction Loss: 0.14548881351947784
It 900: Reconstruction Loss: 0.14141735434532166
Run Epoch 1
It 1000: Reconstruction Loss: 0.13916432857513428
It 1100: Reconstruction Loss: 0.1290625035762787
It 1200: Reconstruction Loss: 0.1364561766386032
It 1300: Reconstruction Loss: 0.12924596667289734
It 1400: Reconstruction Loss: 0.11268249154090881
It 1500: Reconstruction Loss: 0.1280774474143982
It 1600: Reconstruction Loss: 0.12458617240190506
It 1700: Reconstruction Loss: 0.1258636713027954
It 1800: Reconstruction Loss: 0.1256554275751114
Run Epoch 2
It 1900: Reconstruction Loss: 0.12375355511903763
It 2000: Reconstruction Loss: 0.11442513018846512
It 2100: Reconstruction Loss: 0.11658802628517151
It 2200: Reconstruction Loss: 0.10997845977544785
It 2300: Reconstruction Loss: 0.1099126785993576
It 2400: Reconstruction Loss: 0.11876962333917618
It 2500: Reconstruction Loss: 0.11103267967700958
It 2600: Reconstruction Loss: 0.11182372272014618
It 2700: Reconstruction Loss: 0.10971729457378387
It 2800: Reconstruction Loss: 0.11381809413433075
Run Epoch 3
It 2900: Reconstruction Loss: 0.10194595158100128
It 3000: Reconstruction Loss: 0.10225298255681992
It 3100: Reconstruction Loss: 0.10894297808408737
It 3200: Reconstruction Loss: 0.10934257507324219
It 3300: Reconstruction Loss: 0.10538375377655029
It 3400: Reconstruction Loss: 0.09904099255800247
It 3500: Reconstruction Loss: 0.09635597467422485
It 3600: Reconstruction Loss: 0.10076792538166046
It 3700: Reconstruction Loss: 0.0997028723359108
Run Epoch 4
It 3800: Reconstruction Loss: 0.09869039058685303
It 3900: Reconstruction Loss: 0.10684093087911606
It 4000: Reconstruction Loss: 0.09906283020973206
It 4100: Reconstruction Loss: 0.09264710545539856
It 4200: Reconstruction Loss: 0.0966084748506546
It 4300: Reconstruction Loss: 0.09792937338352203
It 4400: Reconstruction Loss: 0.0982743501663208
It 4500: Reconstruction Loss: 0.09316025674343109
It 4600: Reconstruction Loss: 0.10063009709119797
Run Epoch 5
```

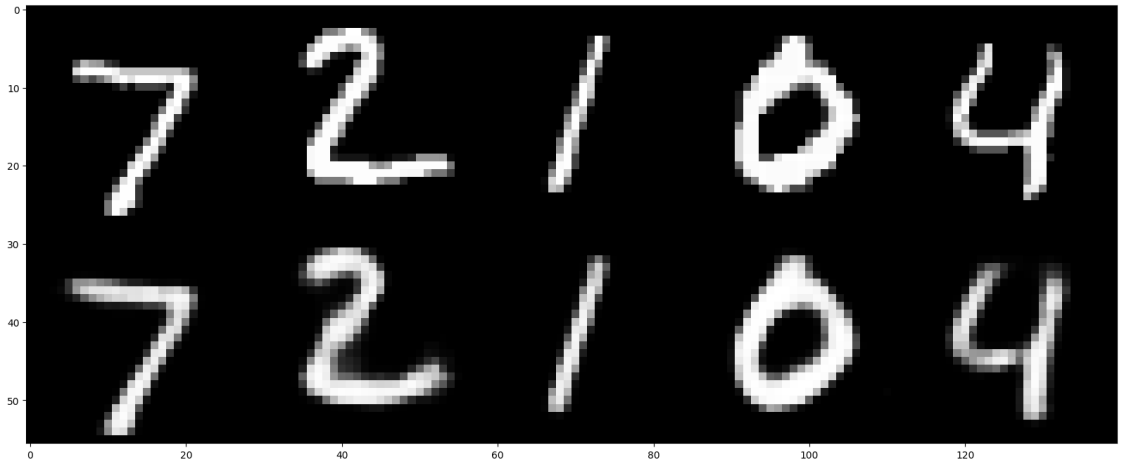
```
It 4700: Reconstruction Loss: 0.09812107682228088
It 4800: Reconstruction Loss: 0.09799942374229431
It 4900: Reconstruction Loss: 0.0964757576584816
It 5000: Reconstruction Loss: 0.10120406746864319
It 5100: Reconstruction Loss: 0.09603039175271988
It 5200: Reconstruction Loss: 0.09620886296033859
It 5300: Reconstruction Loss: 0.09865379333496094
It 5400: Reconstruction Loss: 0.09416192770004272
It 5500: Reconstruction Loss: 0.09518168121576309
It 5600: Reconstruction Loss: 0.09319744259119034
Run Epoch 6
It 5700: Reconstruction Loss: 0.09588009864091873
It 5800: Reconstruction Loss: 0.09223124384880066
It 5900: Reconstruction Loss: 0.08610569685697556
It 6000: Reconstruction Loss: 0.08553403615951538
It 6100: Reconstruction Loss: 0.09873869270086288
It 6200: Reconstruction Loss: 0.09776858240365982
It 6300: Reconstruction Loss: 0.08525542914867401
It 6400: Reconstruction Loss: 0.0951789915561676
It 6500: Reconstruction Loss: 0.08746118098497391
Run Epoch 7
It 6600: Reconstruction Loss: 0.08518008887767792
It 6700: Reconstruction Loss: 0.0974484235048294
It 6800: Reconstruction Loss: 0.09418900310993195
It 6900: Reconstruction Loss: 0.08461111783981323
It 7000: Reconstruction Loss: 0.08952615410089493
It 7100: Reconstruction Loss: 0.08548752218484879
It 7200: Reconstruction Loss: 0.09197342395782471
It 7300: Reconstruction Loss: 0.09620743989944458
It 7400: Reconstruction Loss: 0.09509843587875366
Run Epoch 8
It 7500: Reconstruction Loss: 0.08526157587766647
It 7600: Reconstruction Loss: 0.0931568443775177
It 7700: Reconstruction Loss: 0.08814407140016556
It 7800: Reconstruction Loss: 0.09263410419225693
It 7900: Reconstruction Loss: 0.08812088519334793
It 8000: Reconstruction Loss: 0.08883696049451828
It 8100: Reconstruction Loss: 0.09538339078426361
It 8200: Reconstruction Loss: 0.08404916524887085
It 8300: Reconstruction Loss: 0.08520786464214325
It 8400: Reconstruction Loss: 0.08797740936279297
Run Epoch 9
It 8500: Reconstruction Loss: 0.08675126731395721
It 8600: Reconstruction Loss: 0.08737777918577194
It 8700: Reconstruction Loss: 0.09372072666883469
It 8800: Reconstruction Loss: 0.08399766683578491
It 8900: Reconstruction Loss: 0.09331043064594269
It 9000: Reconstruction Loss: 0.08676797151565552
```

```
It 9100: Reconstruction Loss: 0.08886002749204636
It 9200: Reconstruction Loss: 0.09216055274009705
It 9300: Reconstruction Loss: 0.09069150686264038
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.**

[100]:
```
# 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
```

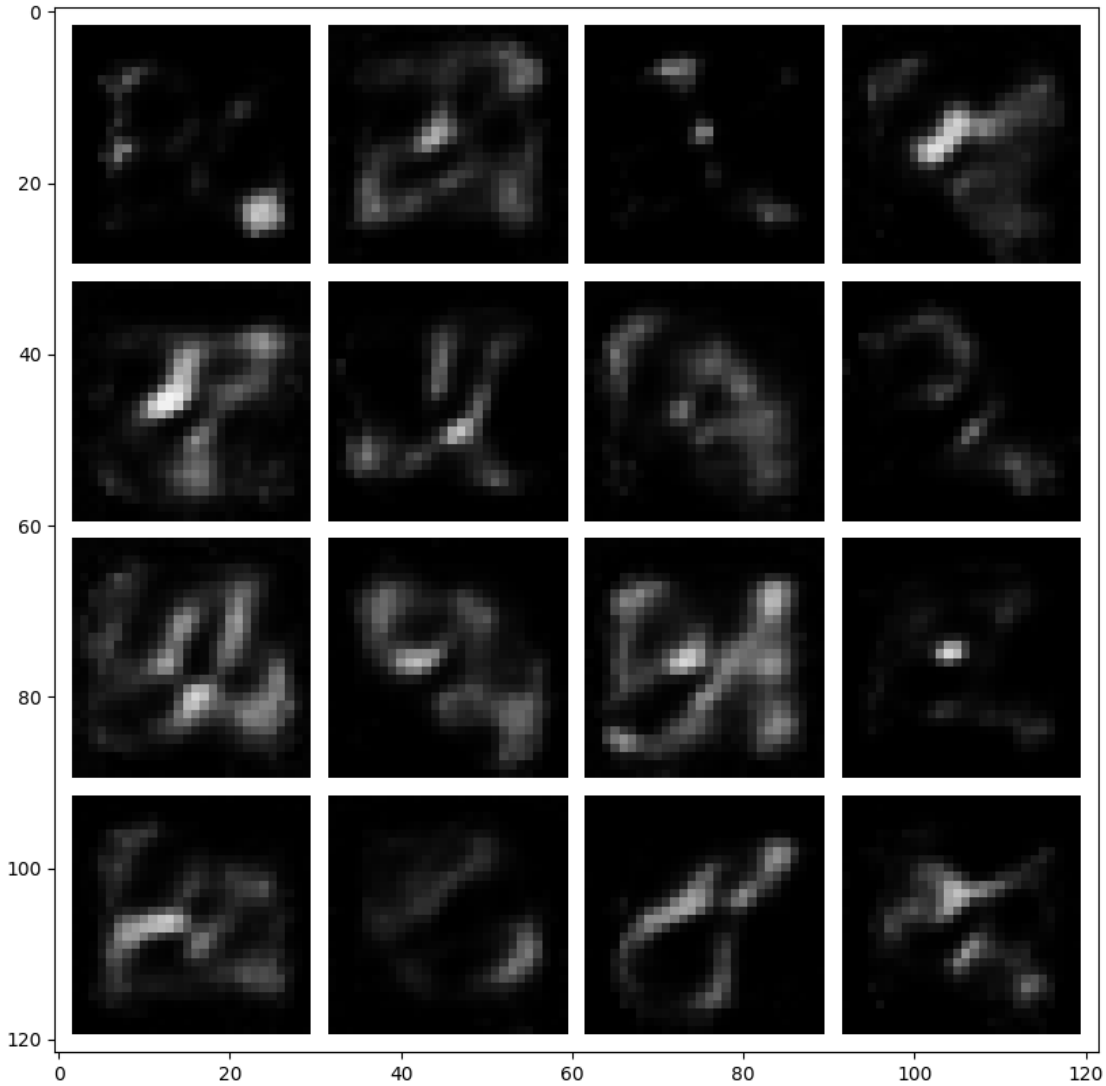


3.5 Sampling from the Auto-Encoder [2pt]

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.

[138]:
```
# 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.
  #      #

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

```
  sampled_embeddings = torch.randn(batch_size, nz).to(device)
  decoded_samples = model.decoder(sampled_embeddings)
  decoded_samples= decoded_samples.reshape(decoded_samples.
 ↪shape[0],decoded_samples.shape[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)
```



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

Answer: While the reconstructions are very realistic, these generated samples don't look like any of the 10 digits that we are trying to generate, and instead look like all of the 10 digits pieced together. This is due to the fact that AE is purely deterministic, and since our inputs are sampled from a gaussian distribution, these images also directly reflect the randomness in addtition to the training data they are based on.

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):

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

Here, $D_{\text{KL}}(q, p)$ 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:

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

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]

[172]:
```
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(nz * 2, 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)
    mean = q[:,:self.nz]
    log_sigma = q[:,self.nz:]
    weight = torch.randn(log_sigma.shape).to(device)
    z = mean + torch.exp(log_sigma) * weight
    reconstruction = self.decoder(z)
    ##################################### 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)                         ␣
↪       #
  ␣
↪##############################################################################
    criterion = nn.BCELoss()
    outputs['rec']= outputs['rec'].reshape([outputs['rec'].shape[0],␣
↪outputs['rec'].shape[2]])
    rec_loss = criterion(outputs['rec'], x)

    q = outputs['q']
    q_mean, q_ls = q[:,:self.nz], q[:,self.nz:]
    p_mean, p_ls = torch.zeros(q_mean.shape).to(device), torch.zeros(q_ls.
↪shape).to(device)
    kl_loss = torch.mean(torch.sum(kl_divergence(q_mean, q_ls, p_mean, p_ls),␣
↪dim=1), dim=0)
    ##################################### 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.                                               ␣
↪       #
  ␣
↪##############################################################################
    q = self.encoder(x)
    mean = q[:self.nz]
```

```
    image = self.decoder(mean)
    image = image.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 $\beta = 0$.

[173]:
```
# Prob1-5 VAE training loop
learning_rate = 1e-3
nz = 32
beta = 0

######################################### TODO␣
 ↪#######################################
epochs = 5       # 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.                    ␣
 ↪    #
#############################################################################################
optimizer = torch.optim.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␣
 ↪#######################################
```

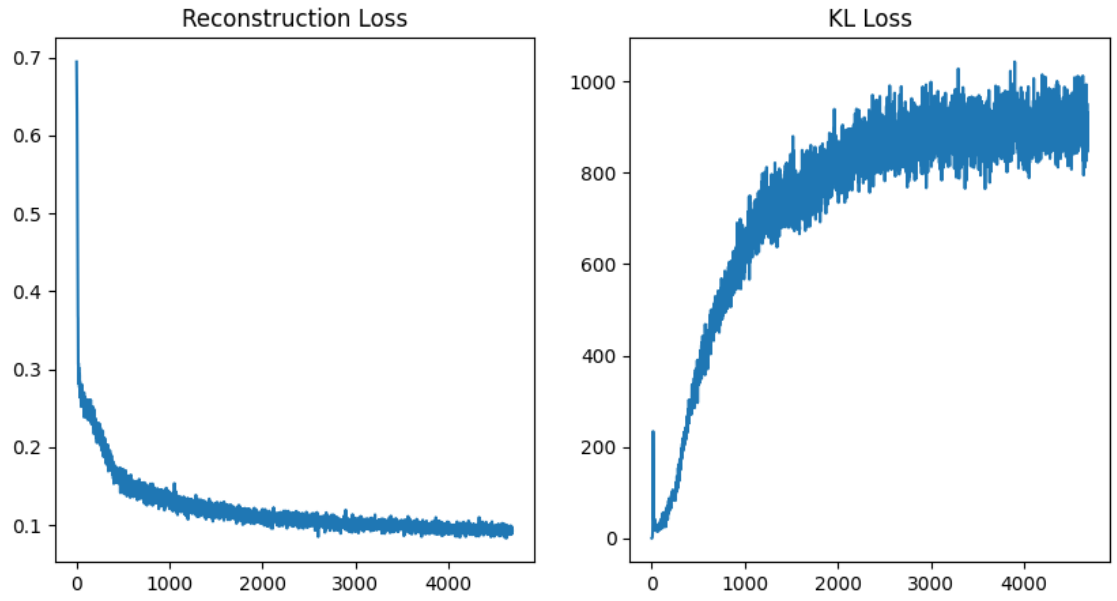
```
  # Implement the main training loop for the VAE model.                    ␣
 ↪     #
  # HINT: Your training loop should sample batches from the data loader, run␣
 ↪the     #
  #       forward pass of the VAE, compute the loss, perform the backward pass␣
 ↪and   #
  #       perform one gradient 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 sample_img, _ in mnist_data_loader:
    sample_img = sample_img.reshape([batch_size, in_size])
    sample_img = sample_img.to(device)
    output = vae_model(sample_img)
    optimizer.zero_grad()
    total_loss, losses = vae_model.loss(sample_img, output)
    total_loss.backward()
    optimizer.step()


    rec_loss.append(losses['rec_loss']); kl_loss.append(losses['kl_loss'])
    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
Run Epoch 0
It 0: Total Loss: 0.6943735480308533,    Rec Loss: 0.6943735480308533,   KL
Loss: 0.35501763224601746
It 100: Total Loss: 0.2577057480812073,          Rec Loss: 0.2577057480812073,
KL Loss: 34.26350021362305
It 200: Total Loss: 0.23379792273044586,         Rec Loss: 0.23379792273044586,
KL Loss: 78.69709014892578
It 300: Total Loss: 0.19293224811553955,         Rec Loss: 0.19293224811553955,
KL Loss: 170.7700958251953
It 400: Total Loss: 0.1532227247953415,          Rec Loss: 0.1532227247953415,
KL Loss: 285.20458984375
It 500: Total Loss: 0.16210757195949554,         Rec Loss: 0.16210757195949554,
KL Loss: 342.786865234375
It 600: Total Loss: 0.15782546997070312,         Rec Loss: 0.15782546997070312,
KL Loss: 370.58203125
It 700: Total Loss: 0.14507314562797546,         Rec Loss: 0.14507314562797546,
KL Loss: 527.5662841796875
It 800: Total Loss: 0.1423225849866867,          Rec Loss: 0.1423225849866867,
KL Loss: 508.6618347167969
It 900: Total Loss: 0.13328149914741516,         Rec Loss: 0.13328149914741516,
KL Loss: 597.4258422851562
Run Epoch 1
It 1000: Total Loss: 0.12876084446907043,        Rec Loss: 0.12876084446907043,
KL Loss: 594.1229248046875
It 1100: Total Loss: 0.1244005635380745,         Rec Loss: 0.1244005635380745,
KL Loss: 661.9530639648438
It 1200: Total Loss: 0.12936484813690186,        Rec Loss: 0.12936484813690186,
KL Loss: 656.5836791992188
It 1300: Total Loss: 0.1271369308233261,         Rec Loss: 0.1271369308233261,
KL Loss: 726.16064453125
It 1400: Total Loss: 0.1218051165342331,         Rec Loss: 0.1218051165342331,
KL Loss: 738.068603515625
It 1500: Total Loss: 0.1221933364868164,         Rec Loss: 0.1221933364868164,
KL Loss: 779.0858154296875
It 1600: Total Loss: 0.11261708289384842,        Rec Loss: 0.11261708289384842,
KL Loss: 756.8787841796875
It 1700: Total Loss: 0.11508123576641083,        Rec Loss: 0.11508123576641083,
KL Loss: 755.899169921875
It 1800: Total Loss: 0.11034765094518661,        Rec Loss: 0.11034765094518661,
KL Loss: 776.6708984375
```

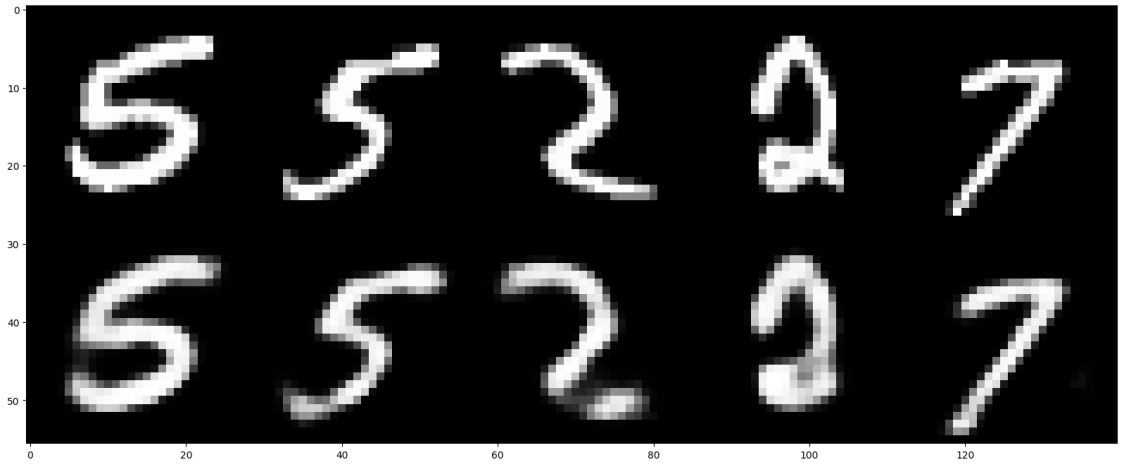
```
Run Epoch 2
It 1900: Total Loss: 0.11909809708595276,        Rec Loss: 0.11909809708595276,
KL Loss: 840.705322265625
It 2000: Total Loss: 0.1136472299695015,         Rec Loss: 0.1136472299695015,
KL Loss: 779.4839477539062
It 2100: Total Loss: 0.10919971764087677,        Rec Loss: 0.10919971764087677,
KL Loss: 894.626708984375
It 2200: Total Loss: 0.10251037776470184,        Rec Loss: 0.10251037776470184,
KL Loss: 939.8167114257812
It 2300: Total Loss: 0.10934096574783325,        Rec Loss: 0.10934096574783325,
KL Loss: 860.928466796875
It 2400: Total Loss: 0.10357362031936646,        Rec Loss: 0.10357362031936646,
KL Loss: 826.7959594726562
It 2500: Total Loss: 0.10548515617847443,        Rec Loss: 0.10548515617847443,
KL Loss: 848.298095703125
It 2600: Total Loss: 0.10425085574388504,        Rec Loss: 0.10425085574388504,
KL Loss: 956.17822265625
It 2700: Total Loss: 0.10498742014169693,        Rec Loss: 0.10498742014169693,
KL Loss: 888.339111328125
It 2800: Total Loss: 0.10247649997472763,        Rec Loss: 0.10247649997472763,
KL Loss: 881.1948852539062
Run Epoch 3
It 2900: Total Loss: 0.09440292418003082,        Rec Loss: 0.09440292418003082,
KL Loss: 882.683837890625
It 3000: Total Loss: 0.10270338505506516,        Rec Loss: 0.10270338505506516,
KL Loss: 998.82958984375
It 3100: Total Loss: 0.1022912859916687,         Rec Loss: 0.1022912859916687,
KL Loss: 879.8482055664062
It 3200: Total Loss: 0.10358843207359314,        Rec Loss: 0.10358843207359314,
KL Loss: 833.2611083984375
It 3300: Total Loss: 0.09074004739522934,        Rec Loss: 0.09074004739522934,
KL Loss: 874.718994140625
It 3400: Total Loss: 0.10464642941951752,        Rec Loss: 0.10464642941951752,
KL Loss: 839.3648071289062
It 3500: Total Loss: 0.09718982875347137,        Rec Loss: 0.09718982875347137,
KL Loss: 852.91259765625
It 3600: Total Loss: 0.09772277623414993,        Rec Loss: 0.09772277623414993,
KL Loss: 946.2738647460938
It 3700: Total Loss: 0.10220272839069366,        Rec Loss: 0.10220272839069366,
KL Loss: 987.5896606445312
Run Epoch 4
It 3800: Total Loss: 0.09290503710508347,        Rec Loss: 0.09290503710508347,
KL Loss: 884.8182983398438
It 3900: Total Loss: 0.09533505886793137,        Rec Loss: 0.09533505886793137,
KL Loss: 1031.7857666015625
It 4000: Total Loss: 0.09258952736854553,        Rec Loss: 0.09258952736854553,
KL Loss: 912.0250854492188
It 4100: Total Loss: 0.09851173311471939,        Rec Loss: 0.09851173311471939,
```

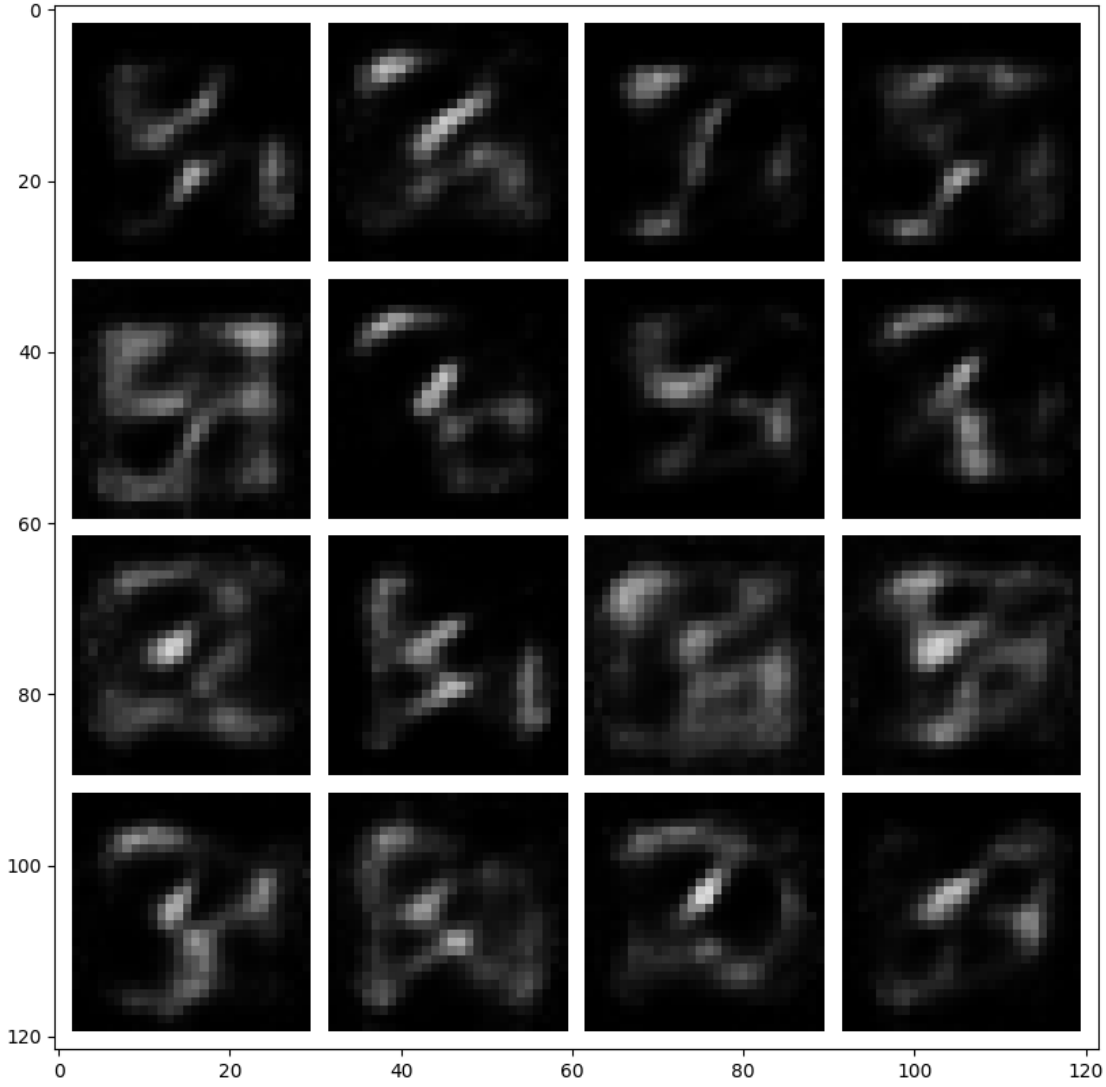
```
KL Loss: 896.8978881835938
It 4200: Total Loss: 0.09715785086154938,        Rec Loss: 0.09715785086154938,
KL Loss: 891.7191162109375
It 4300: Total Loss: 0.09517809748649597,        Rec Loss: 0.09517809748649597,
KL Loss: 890.8844604492188
It 4400: Total Loss: 0.09387403726577759,        Rec Loss: 0.09387403726577759,
KL Loss: 947.7421875
It 4500: Total Loss: 0.09571227431297302,        Rec Loss: 0.09571227431297302,
KL Loss: 901.41357421875
It 4600: Total Loss: 0.09931296110153198,        Rec Loss: 0.09931296110153198,
KL Loss: 937.7604370117188
Done!
```



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

[175]:
```
# visualize VAE reconstructions and samples from the generative model
vis_reconstruction(vae_model, randomize=True)
vis_samples(vae_model)
```





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.

[182]:
```
# Prob1-5 VAE training loop
learning_rate = 1e-3
nz = 32
beta = 10

######################################## TODO␣
↪#######################################
epochs = 5       # 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.                    ␣
↪    #
################################################################################
optimizer = torch.optim.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     #
```

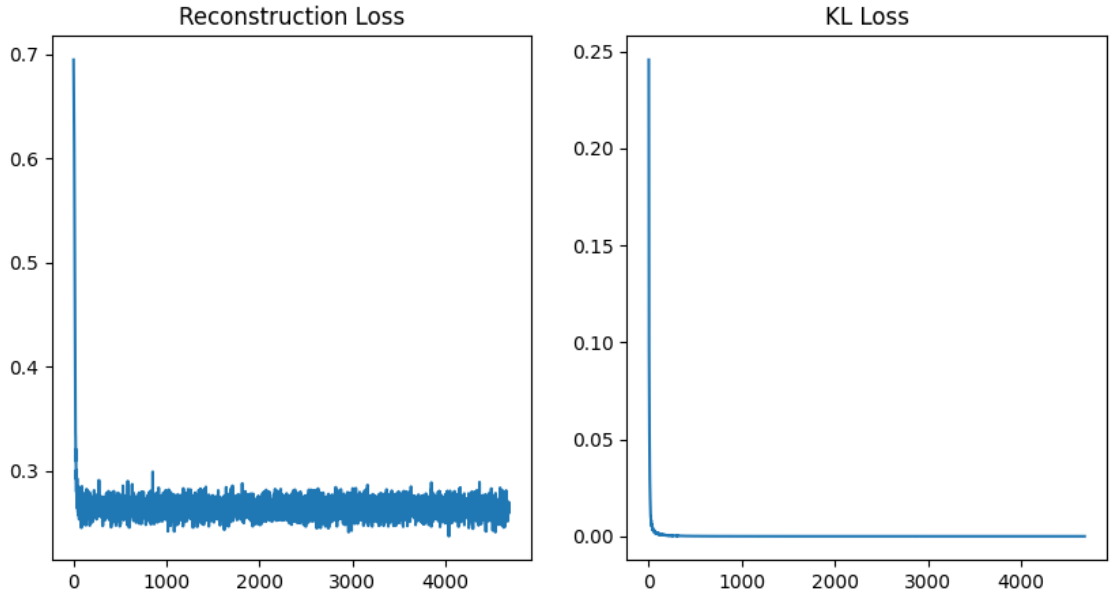
```
  #       forward pass of the VAE, compute the loss, perform the backward pass␣
 ↪and   #
  #       perform one gradient step with the optimizer.                        ␣
 ↪     #
  # HINT: Don't forget to erase old gradients before performing the backward␣
 ↪pass.   #
  # HINT: This time we will 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 sample_img, _ in mnist_data_loader:
    sample_img = sample_img.reshape([batch_size, in_size])
    sample_img = sample_img.to(device)
    output = vae_model(sample_img)
    optimizer.zero_grad()
    total_loss, losses = vae_model.loss(sample_img, output)
    total_loss.backward()
    optimizer.step()


    rec_loss.append(losses['rec_loss']); kl_loss.append(losses['kl_loss'])
    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()
```

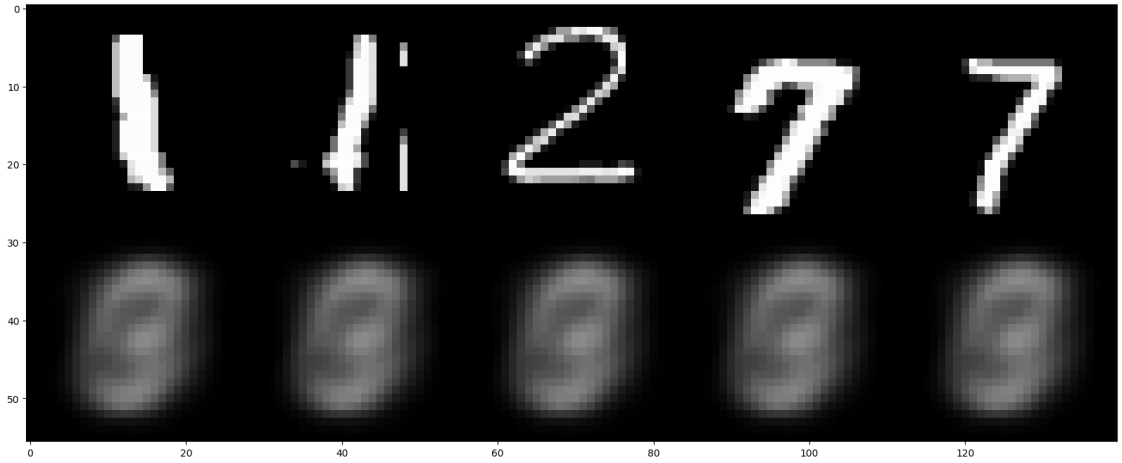
```
Running 5 epochs with beta=10
Run Epoch 0
It 0: Total Loss: 3.151681423187256,     Rec Loss: 0.6947468519210815,   KL
Loss: 0.24569346010684967
It 100: Total Loss: 0.28901875019073486,         Rec Loss: 0.27230772376060486,
KL Loss: 0.0016711014322936535
It 200: Total Loss: 0.2589009702205658,          Rec Loss: 0.2545331120491028,
KL Loss: 0.00043678656220436096
It 300: Total Loss: 0.25815775990486145,         Rec Loss: 0.2555581033229828,
KL Loss: 0.0002599665895104408
It 400: Total Loss: 0.2644473612308502,          Rec Loss: 0.26243096590042114,
KL Loss: 0.00020163832232356071
It 500: Total Loss: 0.26545441150665283,         Rec Loss: 0.26456505060195923,
KL Loss: 8.893711492419243e-05
It 600: Total Loss: 0.257619708776474,   Rec Loss: 0.2569688856601715,   KL
Loss: 6.508128717541695e-05
It 700: Total Loss: 0.2561585307121277,          Rec Loss: 0.2557874321937561,
KL Loss: 3.711041063070297e-05
It 800: Total Loss: 0.2628690302371979,          Rec Loss: 0.26249995827674866,
KL Loss: 3.6908313632011414e-05
It 900: Total Loss: 0.2504001259803772,          Rec Loss: 0.25013235211372375,
KL Loss: 2.677692100405693e-05
Run Epoch 1
It 1000: Total Loss: 0.26486942172050476,        Rec Loss: 0.2646584212779999,
KL Loss: 2.110050991177559e-05
It 1100: Total Loss: 0.269294410943985,          Rec Loss: 0.2691652774810791,
KL Loss: 1.2914184480905533e-05
It 1200: Total Loss: 0.2573128640651703,         Rec Loss: 0.25719955563545227,
KL Loss: 1.1329539120197296e-05
It 1300: Total Loss: 0.26282015442848206,        Rec Loss: 0.26271188259124756,
KL Loss: 1.0826624929904938e-05
It 1400: Total Loss: 0.25814616680145264,        Rec Loss: 0.2580588161945343,
KL Loss: 8.734408766031265e-06
It 1500: Total Loss: 0.2556641101837158,         Rec Loss: 0.2555943727493286,
KL Loss: 6.972812116146088e-06
It 1600: Total Loss: 0.2690756320953369,         Rec Loss: 0.26900821924209595,
KL Loss: 6.7427754402160645e-06
It 1700: Total Loss: 0.2546180486679077,         Rec Loss: 0.2545745372772217,
KL Loss: 4.349742084741592e-06
It 1800: Total Loss: 0.26088616251945496,        Rec Loss: 0.26082611083984375,
KL Loss: 6.003770977258682e-06
Run Epoch 2
It 1900: Total Loss: 0.2664547562599182,         Rec Loss: 0.26642292737960815,
KL Loss: 3.1813979148864746e-06
It 2000: Total Loss: 0.25304853916168213,        Rec Loss: 0.2530236542224884,
```

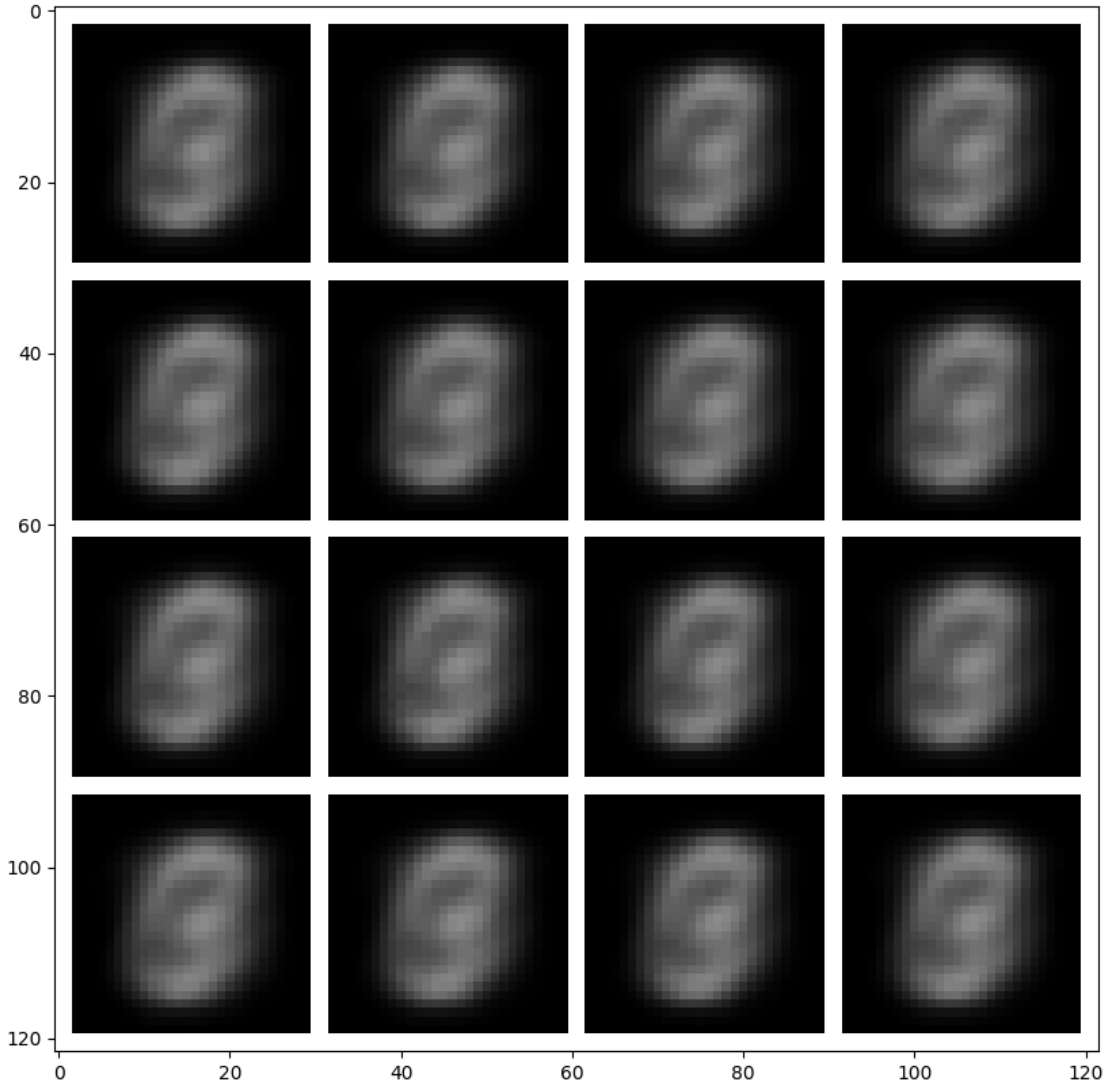
KL Loss: 2.4884939193725586e-06
It 2100: Total Loss: 0.2664833068847656, Rec Loss: 0.26645201444625854,
KL Loss: 3.1278468668460846e-06
It 2200: Total Loss: 0.26312708854675293, Rec Loss: 0.2631032466888428,
KL Loss: 2.385117135902405e-06
It 2300: Total Loss: 0.27295392751693726, Rec Loss: 0.27290958166122437,
KL Loss: 4.434958100318909e-06
It 2400: Total Loss: 0.27062860131263733, Rec Loss: 0.27054381370544434,
KL Loss: 8.477363735437393e-06
It 2500: Total Loss: 0.2677670419216156, Rec Loss: 0.2677440941333771,
KL Loss: 2.2961758077144623e-06
It 2600: Total Loss: 0.2716025114059448, Rec Loss: 0.2715895473957062,
KL Loss: 1.2973323464393616e-06
It 2700: Total Loss: 0.25893276929855347, Rec Loss: 0.25892260670661926,
KL Loss: 1.0151416063308716e-06
It 2800: Total Loss: 0.2594742774963379, Rec Loss: 0.25944191217422485,
KL Loss: 3.237742930650711e-06
Run Epoch 3
It 2900: Total Loss: 0.25203514099121094, Rec Loss: 0.25196486711502075,
KL Loss: 7.028225809335709e-06
It 3000: Total Loss: 0.2828879654407501, Rec Loss: 0.2828701138496399,
KL Loss: 1.7844140529632568e-06
It 3100: Total Loss: 0.26038092374801636, Rec Loss: 0.26035648584365845,
KL Loss: 2.443324774503708e-06
It 3200: Total Loss: 0.26154014468193054, Rec Loss: 0.26152005791664124,
KL Loss: 2.008862793445587e-06
It 3300: Total Loss: 0.27806276082992554, Rec Loss: 0.2780208885669708,
KL Loss: 4.186294972896576e-06
It 3400: Total Loss: 0.26738911867141724, Rec Loss: 0.2673821449279785,
KL Loss: 6.980262696743011e-07
It 3500: Total Loss: 0.25050199031829834, Rec Loss: 0.250466912984848,
KL Loss: 3.5087577998638153e-06
It 3600: Total Loss: 0.2619476318359375, Rec Loss: 0.26193997263908386,
KL Loss: 7.664784789085388e-07
It 3700: Total Loss: 0.27052050828933716, Rec Loss: 0.2705138027667999,
KL Loss: 6.710179150104523e-07
Run Epoch 4
It 3800: Total Loss: 0.25857630372047424, Rec Loss: 0.2585737705230713,
KL Loss: 2.523884177207947e-07
It 3900: Total Loss: 0.26287367939949036, Rec Loss: 0.2628489136695862,
KL Loss: 2.476386725902557e-06
It 4000: Total Loss: 0.2548661231994629, Rec Loss: 0.2548176646232605,
KL Loss: 4.845205694437027e-06
It 4100: Total Loss: 0.2678057849407196, Rec Loss: 0.26778653264045715,
KL Loss: 1.9245781004428864e-06
It 4200: Total Loss: 0.27792420983314514, Rec Loss: 0.2779073119163513,
KL Loss: 1.6889534890651703e-06
It 4300: Total Loss: 0.2676580250263214, Rec Loss: 0.267654687166214,

```
KL Loss: 3.3248215913772583e-07
It 4400: Total Loss: 0.25811997056007385,        Rec Loss: 0.25811347365379333,
KL Loss: 6.505288183689117e-07
It 4500: Total Loss: 0.25751399993896484,        Rec Loss: 0.25744128227233887,
KL Loss: 7.270369678735733e-06
It 4600: Total Loss: 0.25455811619758606,        Rec Loss: 0.25452056527137756,
KL Loss: 3.753695636987686e-06
Done!
```



[183]:
```
# visualize VAE reconstructions and samples from the generative model
vis_reconstruction(vae_model, randomize=True)
vis_samples(vae_model)
```





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

Answer: When beta = 0, the reconstructions look highly realistic and similar to the training data, while the samples look nothing like the 10 digits yet they share a similar hand-written style, similar to what we observed with AE. This is due to beta being 0, meaning that the regularization, or KL loss is completely ignored, making the VAE function like the AE. When beta = 10, the KL loss is so heavily weighted that the reconstruction loss is almost completely muted. The VAE's parameters are optimized to minimize KL loss and completely ignore the reconstruction quality, which made both the reconstructions and the samples blurry and do not resemble any of the digits or look hand-written at all.

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**: A well-tuned model should have a good balance between reconstructions and samples, meaning that for reconstructions, the generated images should look similar to the inputs, and the samples should look like one of the 10 digits it's trained with.

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 0.002 [2pt]

Narrowed down to 0.001 with logarithmic seach and fine-tuend to 0.002 for better performance.

[336]:
```
# Prob1-5 VAE training loop
learning_rate = 1e-3
nz = 32
beta = 0.002

######################################## TODO␣
 ↪#######################################
epochs = 10    # 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.                    ␣
 ↪   #
################################################################################
optimizer = torch.optim.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 sample_img, _ in mnist_data_loader:
    sample_img = sample_img.reshape([batch_size, in_size])
    sample_img = sample_img.to(device)
    output = vae_model(sample_img)
    optimizer.zero_grad()
    total_loss, losses = vae_model.loss(sample_img, output)
    total_loss.backward()
    optimizer.step()


    rec_loss.append(losses['rec_loss']); kl_loss.append(losses['kl_loss'])
    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()
```

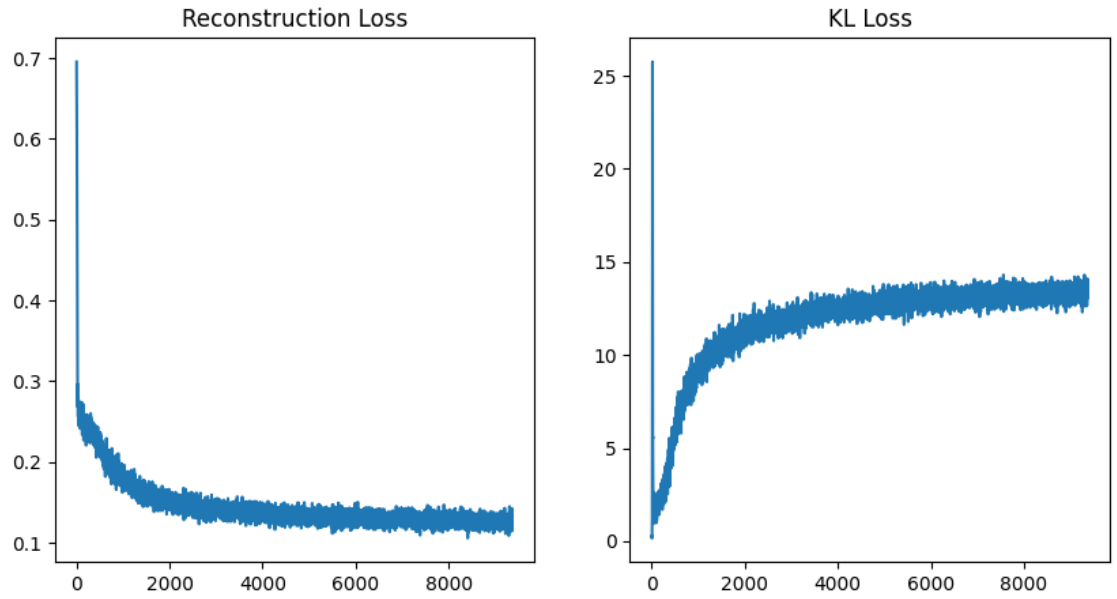
```
Running 10 epochs with beta=0.002
Run Epoch 0
It 0: Total Loss: 0.6957665085792542,    Rec Loss: 0.6952295303344727,   KL
Loss: 0.26847967505455017
It 100: Total Loss: 0.2485547512769699,          Rec Loss: 0.24517822265625,
KL Loss: 1.6882671117782593
It 200: Total Loss: 0.24409446120262146,         Rec Loss: 0.24041977524757385,
KL Loss: 1.8373417854309082
It 300: Total Loss: 0.2418866604566574,          Rec Loss: 0.23542316257953644,
KL Loss: 3.23175048828125
It 400: Total Loss: 0.24236233532428741,         Rec Loss: 0.23341743648052216,
KL Loss: 4.472447872161865
It 500: Total Loss: 0.23645462095737457,         Rec Loss: 0.22599440813064575,
KL Loss: 5.230107307434082
It 600: Total Loss: 0.22482778131961823,         Rec Loss: 0.21120937168598175,
KL Loss: 6.809201717376709
It 700: Total Loss: 0.20759090781211853,         Rec Loss: 0.19207386672496796,
KL Loss: 7.758520126342773
It 800: Total Loss: 0.19642899930477142,         Rec Loss: 0.1807851940393448,
KL Loss: 7.821900367736816
It 900: Total Loss: 0.2147337794303894,          Rec Loss: 0.1971135437488556,
KL Loss: 8.810115814208984
Run Epoch 1
It 1000: Total Loss: 0.19779512286186218,        Rec Loss: 0.180032417178154,
KL Loss: 8.881349563598633
It 1100: Total Loss: 0.20138205587863922,        Rec Loss: 0.18424451351165771,
KL Loss: 8.568772315979004
It 1200: Total Loss: 0.19606170058250427,        Rec Loss: 0.1770782470703125,
KL Loss: 9.491728782653809
It 1300: Total Loss: 0.1876171976327896,         Rec Loss: 0.1671079397201538,
KL Loss: 10.254631042480469
It 1400: Total Loss: 0.17411121726036072,        Rec Loss: 0.15408124029636383,
KL Loss: 10.014985084533691
It 1500: Total Loss: 0.18321916460990906,        Rec Loss: 0.1630917340517044,
```

```
KL Loss: 10.063714027404785
It 1600: Total Loss: 0.18185724318027496,        Rec Loss: 0.1591539829969406,
KL Loss: 11.351628303527832
It 1700: Total Loss: 0.17986159026622772,        Rec Loss: 0.1577969193458557,
KL Loss: 11.03233814239502
It 1800: Total Loss: 0.1773936003446579,         Rec Loss: 0.15543106198310852,
KL Loss: 10.981268882751465
Run Epoch 2
It 1900: Total Loss: 0.16827239096164703,        Rec Loss: 0.14620846509933472,
KL Loss: 11.031959533691406
It 2000: Total Loss: 0.17672178149223328,        Rec Loss: 0.15559184551239014,
KL Loss: 10.56496810913086
It 2100: Total Loss: 0.18245595693588257,        Rec Loss: 0.1588871031999588,
KL Loss: 11.784424781799316
It 2200: Total Loss: 0.17777203023433685,        Rec Loss: 0.1522059291601181,
KL Loss: 12.78304672241211
It 2300: Total Loss: 0.16764357686042786,        Rec Loss: 0.14455607533454895,
KL Loss: 11.54375171661377
It 2400: Total Loss: 0.17261818051338196,        Rec Loss: 0.15007542073726654,
KL Loss: 11.271381378173828
It 2500: Total Loss: 0.1669297069311142,         Rec Loss: 0.14393894374370575,
KL Loss: 11.495378494262695
It 2600: Total Loss: 0.17258626222610474,        Rec Loss: 0.14886094629764557,
KL Loss: 11.862655639648438
It 2700: Total Loss: 0.17128798365592957,        Rec Loss: 0.1476251482963562,
KL Loss: 11.831419944763184
It 2800: Total Loss: 0.1660655438899994,         Rec Loss: 0.14335684478282928,
KL Loss: 11.35434627532959
Run Epoch 3
It 2900: Total Loss: 0.16981816291809082,        Rec Loss: 0.14548733830451965,
KL Loss: 12.165409088134766
It 3000: Total Loss: 0.16087810695171356,        Rec Loss: 0.13702677190303802,
KL Loss: 11.925666809082031
It 3100: Total Loss: 0.15287110209465027,        Rec Loss: 0.12973517179489136,
KL Loss: 11.567962646484375
It 3200: Total Loss: 0.15837781131267548,        Rec Loss: 0.13391605019569397,
KL Loss: 12.230880737304688
It 3300: Total Loss: 0.16922244429588318,        Rec Loss: 0.1453309804201126,
KL Loss: 11.94572925567627
It 3400: Total Loss: 0.16200967133045197,        Rec Loss: 0.13787049055099487,
KL Loss: 12.069587707519531
It 3500: Total Loss: 0.16475129127502441,        Rec Loss: 0.13909371197223663,
KL Loss: 12.828788757324219
It 3600: Total Loss: 0.1625312864780426,         Rec Loss: 0.13828834891319275,
KL Loss: 12.121471405029297
It 3700: Total Loss: 0.16041173040866852,        Rec Loss: 0.13518694043159485,
KL Loss: 12.612397193908691
Run Epoch 4
```

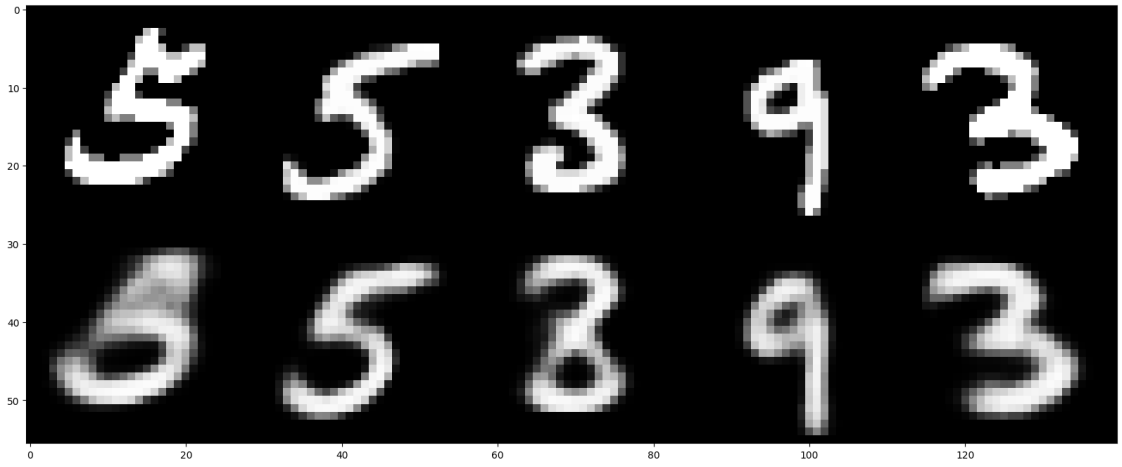
```
It 3800: Total Loss: 0.16296081244945526,       Rec Loss: 0.13869191706180573,
KL Loss: 12.134449005126953
It 3900: Total Loss: 0.16943617165088654,       Rec Loss: 0.1438417285680771,
KL Loss: 12.797220230102539
It 4000: Total Loss: 0.1603834480047226,        Rec Loss: 0.13562500476837158,
KL Loss: 12.379222869873047
It 4100: Total Loss: 0.15817715227603912,       Rec Loss: 0.1331249326467514,
KL Loss: 12.526108741760254
It 4200: Total Loss: 0.15087342262268066,       Rec Loss: 0.12649162113666534,
KL Loss: 12.190901756286621
It 4300: Total Loss: 0.163469597697258,         Rec Loss: 0.13927331566810608,
KL Loss: 12.098138809204102
It 4400: Total Loss: 0.1514604389667511,        Rec Loss: 0.12701961398124695,
KL Loss: 12.220409393310547
It 4500: Total Loss: 0.15795263648033142,       Rec Loss: 0.13269522786140442,
KL Loss: 12.628704071044922
It 4600: Total Loss: 0.1597938984632492,        Rec Loss: 0.13439464569091797,
KL Loss: 12.699626922607422
Run Epoch 5
It 4700: Total Loss: 0.15738122165203094,       Rec Loss: 0.13232587277889252,
KL Loss: 12.527676582336426
It 4800: Total Loss: 0.15896213054656982,       Rec Loss: 0.13295873999595642,
KL Loss: 13.001693725585938
It 4900: Total Loss: 0.1493818163871765,        Rec Loss: 0.12383746355772018,
KL Loss: 12.772173881530762
It 5000: Total Loss: 0.15726254880428314,       Rec Loss: 0.13124476373195648,
KL Loss: 13.008890151977539
It 5100: Total Loss: 0.16908308863639832,       Rec Loss: 0.14399294555187225,
KL Loss: 12.545074462890625
It 5200: Total Loss: 0.1688636690378189,        Rec Loss: 0.14222095906734467,
KL Loss: 13.321353912353516
It 5300: Total Loss: 0.15626178681850433,       Rec Loss: 0.1305096447467804,
KL Loss: 12.87607192993164
It 5400: Total Loss: 0.15244179964065552,       Rec Loss: 0.12716571986675262,
KL Loss: 12.638040542602539
It 5500: Total Loss: 0.15462525188922882,       Rec Loss: 0.12841804325580597,
KL Loss: 13.103601455688477
It 5600: Total Loss: 0.15043462812900543,       Rec Loss: 0.1246681734919548,
KL Loss: 12.88322925567627
Run Epoch 6
It 5700: Total Loss: 0.15229396522045135,       Rec Loss: 0.12657800316810608,
KL Loss: 12.857977867126465
It 5800: Total Loss: 0.1555713266134262,        Rec Loss: 0.12984582781791687,
KL Loss: 12.862751960754395
It 5900: Total Loss: 0.16567766666412354,       Rec Loss: 0.13991127908229828,
KL Loss: 12.883193969726562
It 6000: Total Loss: 0.15920419991016388,       Rec Loss: 0.1332777589559555,
KL Loss: 12.96321964263916
```

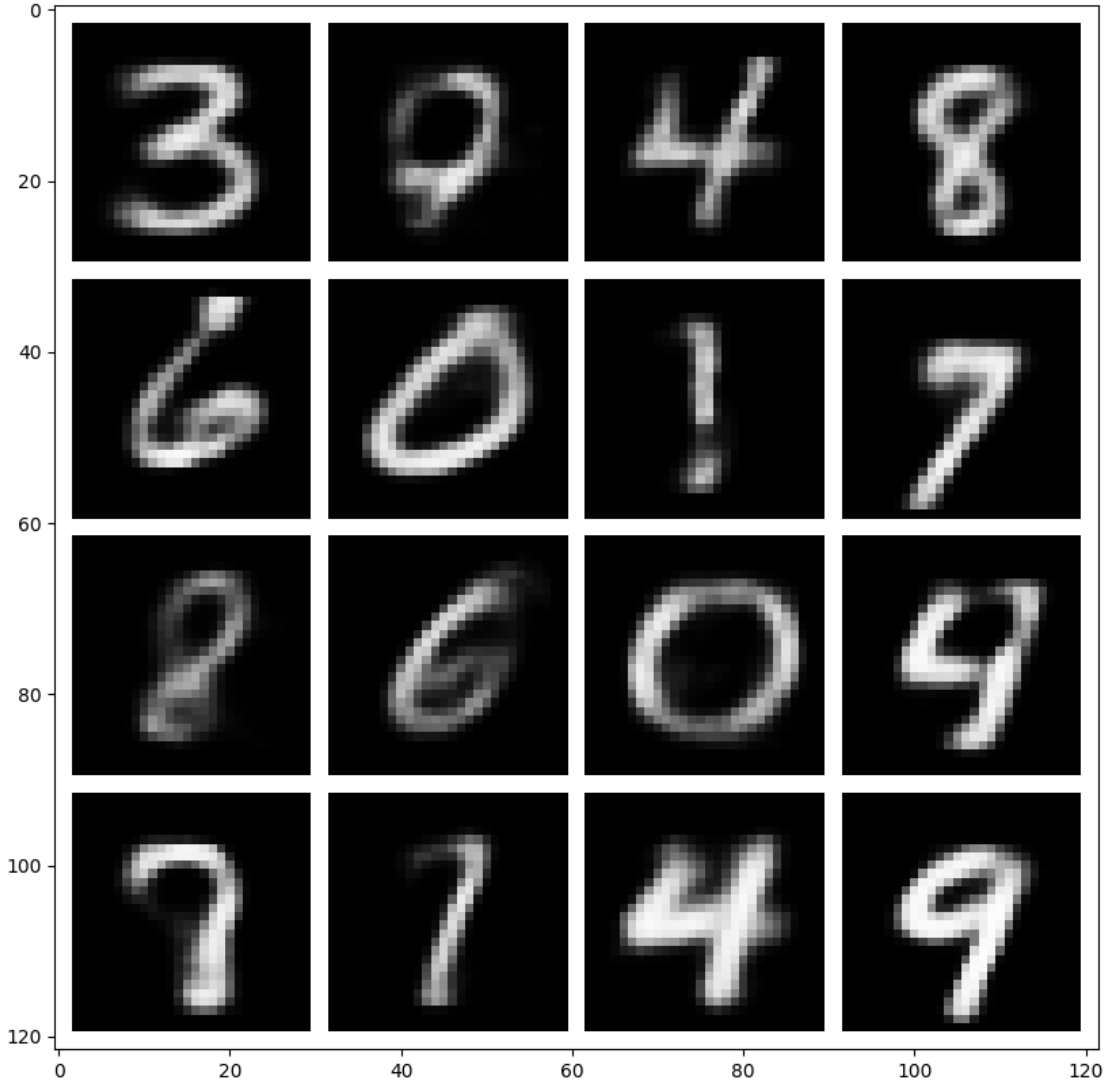
```
It 6100: Total Loss: 0.1548752784729004,         Rec Loss: 0.1291223019361496,
KL Loss: 12.876486778259277
It 6200: Total Loss: 0.15749076008796692,        Rec Loss: 0.1329222172498703,
KL Loss: 12.284273147583008
It 6300: Total Loss: 0.1550011783838272,         Rec Loss: 0.12831348180770874,
KL Loss: 13.34384822845459
It 6400: Total Loss: 0.15568199753761292,        Rec Loss: 0.13018463551998138,
KL Loss: 12.74868106842041
It 6500: Total Loss: 0.15468311309814453,        Rec Loss: 0.12920811772346497,
KL Loss: 12.737497329711914
Run Epoch 7
It 6600: Total Loss: 0.1678178608417511,         Rec Loss: 0.14209896326065063,
KL Loss: 12.859444618225098
It 6700: Total Loss: 0.15597744286060333,        Rec Loss: 0.13028942048549652,
KL Loss: 12.844013214111328
It 6800: Total Loss: 0.16251803934574127,        Rec Loss: 0.1352640688419342,
KL Loss: 13.626985549926758
It 6900: Total Loss: 0.15533877909183502,        Rec Loss: 0.12958276271820068,
KL Loss: 12.878010749816895
It 7000: Total Loss: 0.16182245314121246,        Rec Loss: 0.13502530753612518,
KL Loss: 13.398569107055664
It 7100: Total Loss: 0.1558108925819397,         Rec Loss: 0.1297721415758133,
KL Loss: 13.019378662109375
It 7200: Total Loss: 0.16064223647117615,        Rec Loss: 0.13452553749084473,
KL Loss: 13.058351516723633
It 7300: Total Loss: 0.15352044999599457,        Rec Loss: 0.12744806706905365,
KL Loss: 13.036188125610352
It 7400: Total Loss: 0.16176848113536835,        Rec Loss: 0.13518869876861572,
KL Loss: 13.289892196655273
Run Epoch 8
It 7500: Total Loss: 0.15314677357673645,        Rec Loss: 0.12781809270381927,
KL Loss: 12.664340019226074
It 7600: Total Loss: 0.14969688653945923,        Rec Loss: 0.12310627847909927,
KL Loss: 13.29530143737793
It 7700: Total Loss: 0.1519896537065506,         Rec Loss: 0.12491798400878906,
KL Loss: 13.535833358764648
It 7800: Total Loss: 0.14038190245628357,        Rec Loss: 0.11482832580804825,
KL Loss: 12.77678394317627
It 7900: Total Loss: 0.1471591591835022,         Rec Loss: 0.12031269818544388,
KL Loss: 13.42323112487793
It 8000: Total Loss: 0.1576087772846222,         Rec Loss: 0.1307988315820694,
KL Loss: 13.40496826171875
It 8100: Total Loss: 0.1573343276977539,         Rec Loss: 0.13059452176094055,
KL Loss: 13.369904518127441
It 8200: Total Loss: 0.16518796980381012,        Rec Loss: 0.13766346871852875,
KL Loss: 13.762248992919922
It 8300: Total Loss: 0.1619499921798706,         Rec Loss: 0.13427689671516418,
KL Loss: 13.83654499053955
```

```
It 8400: Total Loss: 0.1542864441871643,         Rec Loss: 0.12775816023349762,
KL Loss: 13.264140129089355
Run Epoch 9
It 8500: Total Loss: 0.14863772690296173,        Rec Loss: 0.12226247042417526,
KL Loss: 13.187629699707031
It 8600: Total Loss: 0.15906944870948792,        Rec Loss: 0.13115306198596954,
KL Loss: 13.958196640014648
It 8700: Total Loss: 0.15172751247882843,        Rec Loss: 0.12513241171836853,
KL Loss: 13.297548294067383
It 8800: Total Loss: 0.15206597745418549,        Rec Loss: 0.12594468891620636,
KL Loss: 13.06064510345459
It 8900: Total Loss: 0.15917757153511047,        Rec Loss: 0.13332225382328033,
KL Loss: 12.927661895751953
It 9000: Total Loss: 0.1548408716917038,         Rec Loss: 0.12891948223114014,
KL Loss: 12.960697174072266
It 9100: Total Loss: 0.14908535778522491,        Rec Loss: 0.12366688251495361,
KL Loss: 12.709237098693848
It 9200: Total Loss: 0.15866954624652863,        Rec Loss: 0.13189749419689178,
KL Loss: 13.386026382446289
It 9300: Total Loss: 0.15092292428016663,        Rec Loss: 0.1238553375005722,
KL Loss: 13.533790588378906
Done!
```



[373]:
```
# visualize VAE reconstructions and samples from the generative model
vis_reconstruction(vae_model, randomize=True)
vis_samples(vae_model)
```



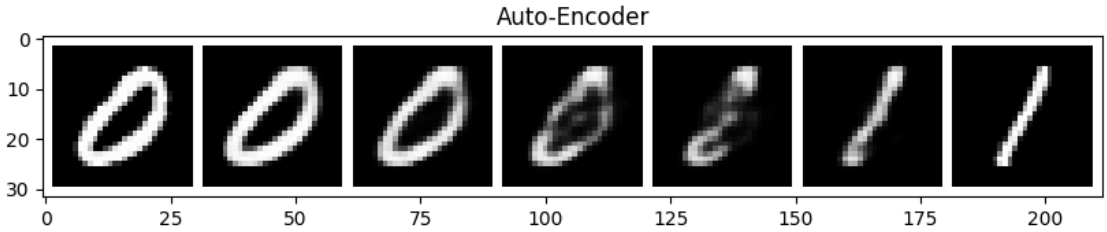


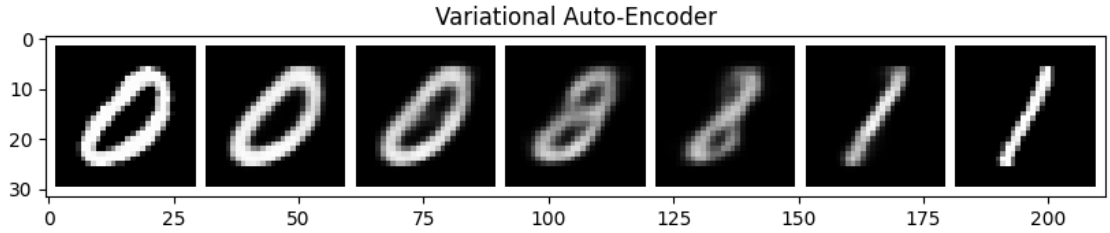
5 4. Embedding Space Interpolation [3pt]

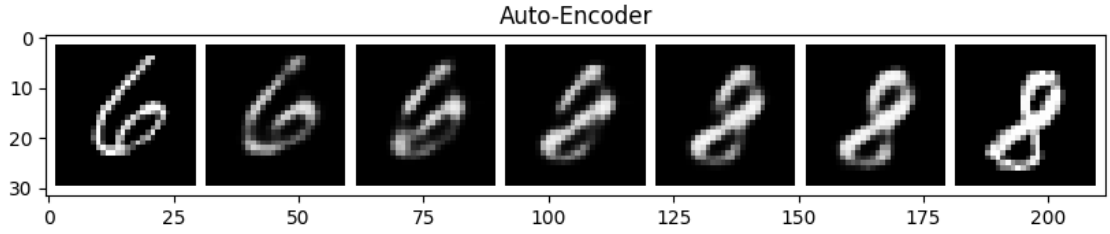
As mentioned in the introduction, AEs and VAEs cannot only be used to generate images, but also to learn low-dimensional representations of their inputs. In this final section we will investigate the representations we learned with both models by **interpolating in embedding space** between different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

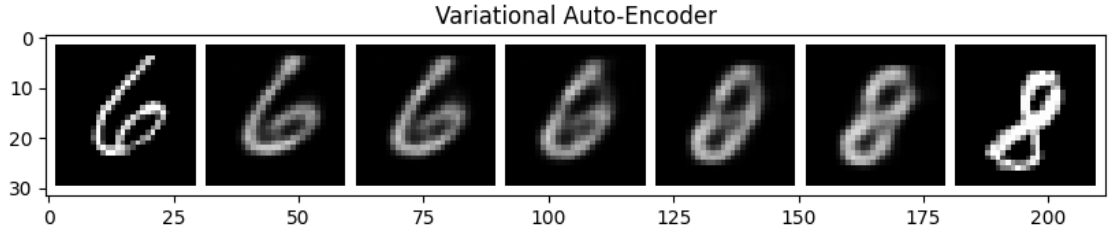
[361]:
```
# Prob1-7
nz=32

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()
```

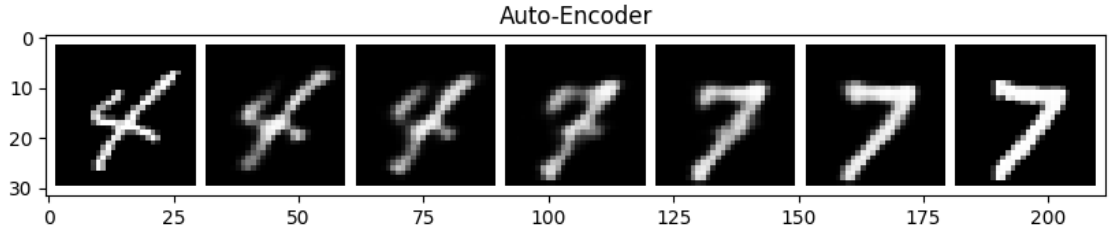
```
### Interpolation 1
START_LABEL = 0
END_LABEL = 1
# sample two training images with given labels
start_img = get_image_with_label(START_LABEL)
end_img = get_image_with_label(END_LABEL)
# visualize interpolations for AE and VAE models
interpolate_and_visualize(ae_model, "Auto-Encoder", start_img,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

### Interpolation 2
START_LABEL = 6
END_LABEL = 8
# 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 3
START_LABEL = 4
END_LABEL = 7
# 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)
```

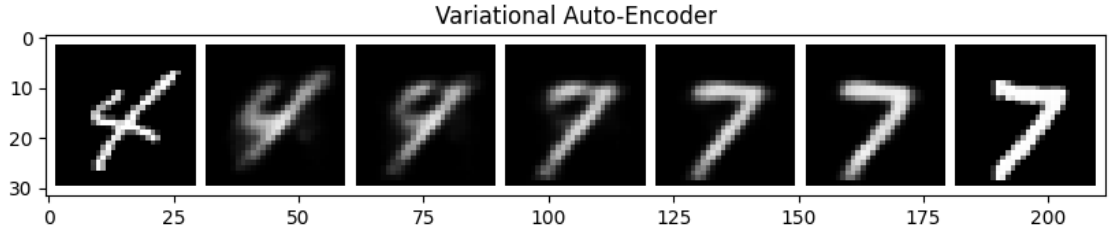



Variational Auto-Encoder
0
10
20
30
0
25
50
75
100
125
150
175
200


Auto-Encoder
0
10
20
30
0
25
50
75
100
125
150
175
200


Variational Auto-Encoder
0
10
20
30
0
25
50
75
100
125
150
175
200


Auto-Encoder
0
10
20
30
0
25
50
75
100
125
150
175
200


Variational Auto-Encoder
0
10
20
30
0
25
50
75
100
125
150
175
200

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**: During start to end label trasitions, AE tends to generate images that do not resemble any of the 10 target labels. VAE, on the other hand, generates images that somewhat still look like one of the target labels, even if it's not the end label desired. For example, when swtching from 0 to 1, VAE generates images that look like 8, while AE fails to generate any meaningful images. This suggests that VAE has a better performance than AE for downstream leanring as it has a more stable generation of target labels, which is crucial for model reliability and performance.

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.

[287]:
```
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(in_features=self.input_size, out_features=256),
        nn.ReLU(),
        nn.Linear(in_features=256, out_features=64),
        nn.ReLU(),
        nn.Linear(in_features=64, out_features=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(in_features=nz, out_features=64),
        nn.ReLU(),
        nn.Linear(in_features=64, out_features=256),
        nn.ReLU(),
        nn.Linear(in_features=256, out_features=self.output_size),
        nn.Sigmoid()
    )
    ################################## 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)                     ␣
↪           #
      ␣
↪#############################################################################################
        criterion = nn.BCELoss()
        outputs['rec']= outputs['rec'].reshape([outputs['rec'].shape[0],␣
↪outputs['rec'].shape[2]])
        rec_loss = criterion(outputs['rec'], x)

        q = outputs['q']
        q_mean, q_ls = q[:,:self.nz], q[:,self.nz:]
        p_mean, p_ls = torch.zeros(q_mean.shape).to(device), torch.zeros(q_ls.
↪shape).to(device)
        kl_loss = torch.mean(torch.sum(kl_divergence(q_mean, q_ls, p_mean,␣
↪p_ls), dim=1), dim=0)
        ######################################## 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)
        mean = q[:self.nz]
        image = self.decoder(mean)
        image = image.reshape(-1, 28, 28)
        ######################################## END TODO␣
 ↪#######################################
        return image
```

6.2 Setting up the CVAE Training loop

[420]:
```
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 = 0.01
####################################### 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␣
 ↪#######################################
```

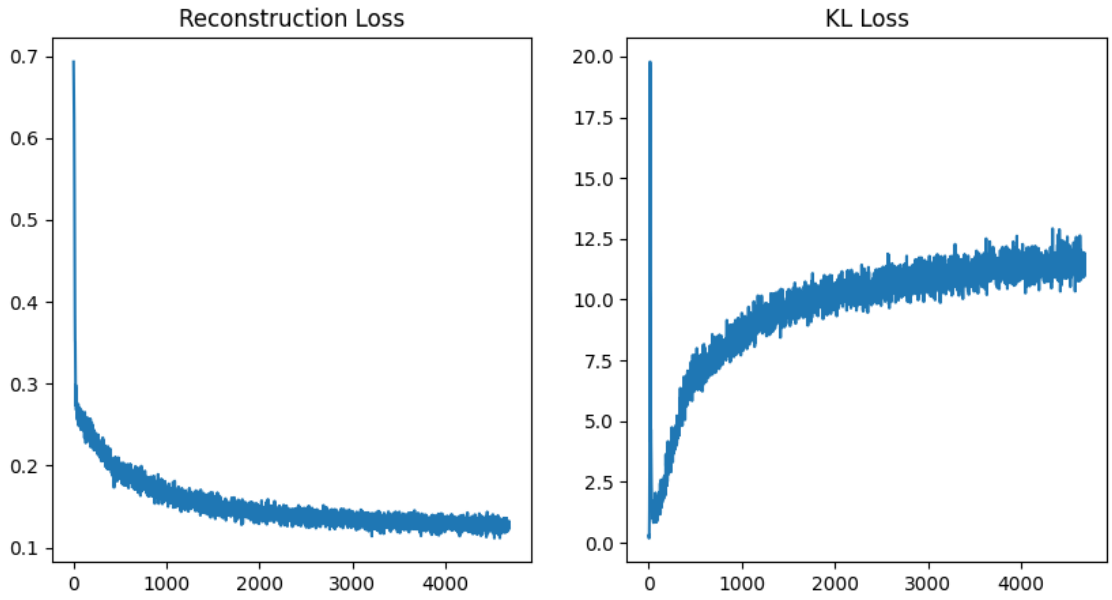
```
# Build the optimizer for the cvae_model. We will again use the Adam optimizer␣
 ↪with #
# the given learning rate and otherwise default parameters.                    ␣
 ↪    #
################################################################################
optimizer = torch.optim.Adam(cvae_model.parameters(), 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 sample_img, label in mnist_data_loader:
    sample_img = sample_img.reshape([batch_size, in_size])
    sample_img = sample_img.to(device)
    output = cvae_model(sample_img, c=label)
    optimizer.zero_grad()
    total_loss, losses = vae_model.loss(sample_img, output)
    total_loss.backward()
    optimizer.step()
```

```
    rec_loss.append(losses['rec_loss']); kl_loss.append(losses['kl_loss'])
    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=0.01
Run Epoch 0
It 0: Total Loss: 0.6935358047485352,    Rec Loss: 0.6929436326026917,   KL
Loss: 0.2960883677005768
It 100: Total Loss: 0.26474207639694214,         Rec Loss: 0.2619436979293823,
KL Loss: 1.3991867303848267
It 200: Total Loss: 0.2341773808002472,          Rec Loss: 0.2284797728061676,
KL Loss: 2.848804473876953
It 300: Total Loss: 0.2322886735200882,          Rec Loss: 0.22337360680103302,
KL Loss: 4.457530975341797
It 400: Total Loss: 0.21966111660003662,         Rec Loss: 0.2075946182012558,
KL Loss: 6.033248424530029
It 500: Total Loss: 0.21144942939281464,         Rec Loss: 0.19825495779514313,
KL Loss: 6.597233295440674
It 600: Total Loss: 0.20878243446350098,         Rec Loss: 0.19441857933998108,
KL Loss: 7.18192720413208
It 700: Total Loss: 0.1997614949941635,          Rec Loss: 0.18551339209079742,
KL Loss: 7.124050140380859
It 800: Total Loss: 0.19499661028385162,         Rec Loss: 0.17860202491283417,
KL Loss: 8.19729232788086
It 900: Total Loss: 0.1852623075246811,          Rec Loss: 0.16838829219341278,
KL Loss: 8.437007904052734
```

```
Run Epoch 1
It 1000: Total Loss: 0.19231921434402466,        Rec Loss: 0.17342300713062286,
KL Loss: 9.448099136352539
It 1100: Total Loss: 0.17396648228168488,        Rec Loss: 0.1570943295955658,
KL Loss: 8.436073303222656
It 1200: Total Loss: 0.1710575520992279,         Rec Loss: 0.1531972587108612,
KL Loss: 8.930142402648926
It 1300: Total Loss: 0.1834518313407898,         Rec Loss: 0.16429518163204193,
KL Loss: 9.578322410583496
It 1400: Total Loss: 0.17625729739665985,        Rec Loss: 0.15726348757743835,
KL Loss: 9.496904373168945
It 1500: Total Loss: 0.1713685393333435,         Rec Loss: 0.15138904750347137,
KL Loss: 9.989745140075684
It 1600: Total Loss: 0.16961167752742767,        Rec Loss: 0.15005694329738617,
KL Loss: 9.777363777160645
It 1700: Total Loss: 0.1814037412405014,         Rec Loss: 0.1608085036277771,
KL Loss: 10.29762077331543
It 1800: Total Loss: 0.1583620011806488,         Rec Loss: 0.13931375741958618,
KL Loss: 9.52411937713623
Run Epoch 2
It 1900: Total Loss: 0.17302896082401276,        Rec Loss: 0.1523207724094391,
KL Loss: 10.354093551635742
It 2000: Total Loss: 0.16893990337848663,        Rec Loss: 0.14842288196086884,
KL Loss: 10.258508682250977
It 2100: Total Loss: 0.1719074845314026,         Rec Loss: 0.1502954363822937,
KL Loss: 10.806022644042969
It 2200: Total Loss: 0.16278207302093506,        Rec Loss: 0.14268562197685242,
KL Loss: 10.048223495483398
It 2300: Total Loss: 0.1576378345489502,         Rec Loss: 0.13731876015663147,
KL Loss: 10.159538269042969
It 2400: Total Loss: 0.1582363247871399,         Rec Loss: 0.13805541396141052,
KL Loss: 10.0904541015625
It 2500: Total Loss: 0.15296058356761932,        Rec Loss: 0.13281936943531036,
KL Loss: 10.070605278015137
It 2600: Total Loss: 0.16662615537643433,        Rec Loss: 0.14462988078594208,
KL Loss: 10.998138427734375
It 2700: Total Loss: 0.15396837890148163,        Rec Loss: 0.13306836783885956,
KL Loss: 10.450004577636719
It 2800: Total Loss: 0.16139809787273407,        Rec Loss: 0.13920624554157257,
KL Loss: 11.095928192138672
Run Epoch 3
It 2900: Total Loss: 0.16162139177322388,        Rec Loss: 0.14061987400054932,
KL Loss: 10.50075912475586
It 3000: Total Loss: 0.15273991227149963,        Rec Loss: 0.13195757567882538,
KL Loss: 10.39116382598877
It 3100: Total Loss: 0.1594906449317932,         Rec Loss: 0.13730542361736298,
KL Loss: 11.09261417388916
It 3200: Total Loss: 0.14755786955356598,        Rec Loss: 0.12605498731136322,
```

```
KL Loss: 10.751441955566406
It 3300: Total Loss: 0.1527685970067978,        Rec Loss: 0.13053888082504272,
KL Loss: 11.114859580993652
It 3400: Total Loss: 0.15604491531848907,       Rec Loss: 0.13413920998573303,
KL Loss: 10.952849388122559
It 3500: Total Loss: 0.1459040343761444,        Rec Loss: 0.12429933995008469,
KL Loss: 10.802347183227539
It 3600: Total Loss: 0.14695483446121216,       Rec Loss: 0.12542381882667542,
KL Loss: 10.765509605407715
It 3700: Total Loss: 0.16594113409519196,       Rec Loss: 0.14161260426044464,
KL Loss: 12.164262771606445
Run Epoch 4
It 3800: Total Loss: 0.14608174562454224,       Rec Loss: 0.12366638332605362,
KL Loss: 11.207680702209473
It 3900: Total Loss: 0.14977988600730896,       Rec Loss: 0.1278030276298523,
KL Loss: 10.988431930541992
It 4000: Total Loss: 0.15285593271255493,       Rec Loss: 0.12907752394676208,
KL Loss: 11.889204978942871
It 4100: Total Loss: 0.14653339982032776,       Rec Loss: 0.12428390979766846,
KL Loss: 11.124746322631836
It 4200: Total Loss: 0.15139280259609222,       Rec Loss: 0.12872083485126495,
KL Loss: 11.335982322692871
It 4300: Total Loss: 0.1493145227432251,        Rec Loss: 0.12618306279182434,
KL Loss: 11.565731048583984
It 4400: Total Loss: 0.1579899787902832,        Rec Loss: 0.13474208116531372,
KL Loss: 11.62394905090332
It 4500: Total Loss: 0.15313488245010376,       Rec Loss: 0.1293616145849228,
KL Loss: 11.88663101196289
It 4600: Total Loss: 0.15490607917308807,       Rec Loss: 0.13195520639419556,
KL Loss: 11.475439071655273
Done!
```

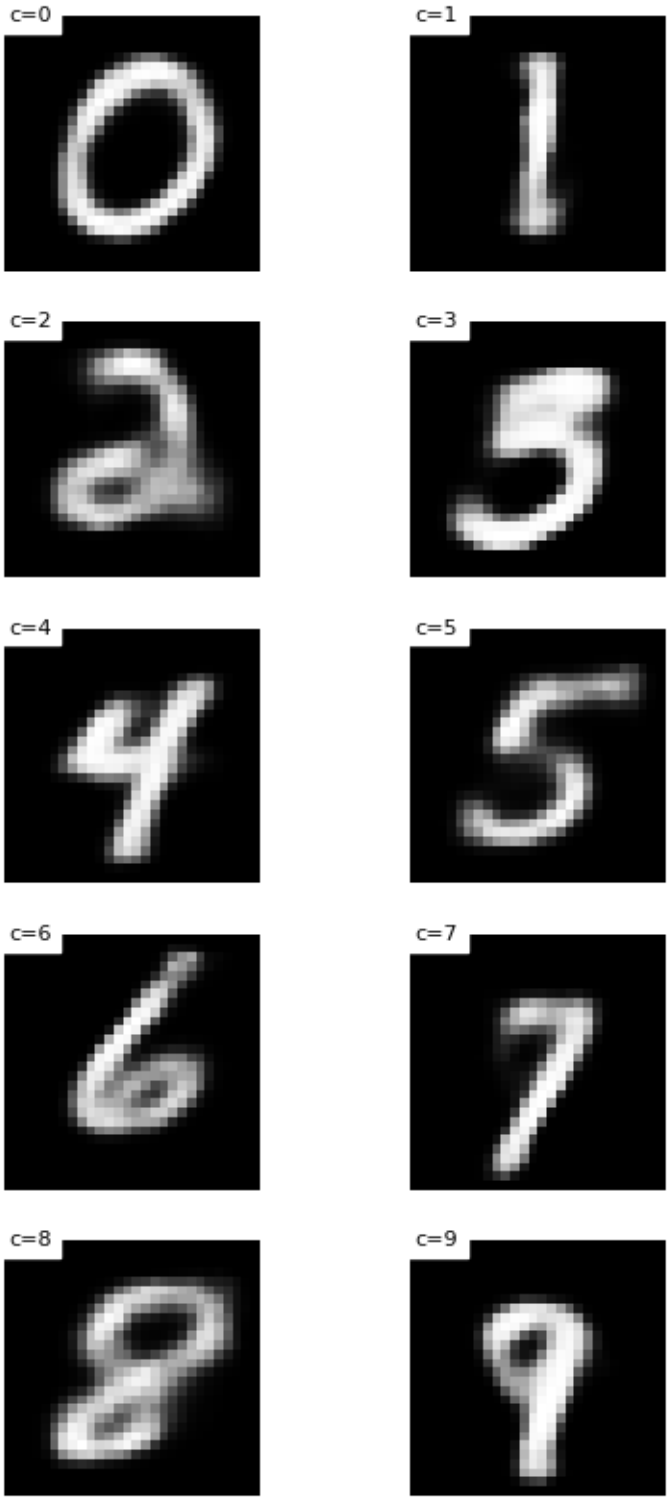


6.2.1 Verifying conditional samples from CVAE [6 pt]

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

[439]:
```
# 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

Problem2_GAN_v0

March 21, 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.

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

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

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`.

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

```
--2023-03-21 16:07:54--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)… 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 170498071 (163M) [application/x-gzip]
```

```
Saving to: 'data/cifar-10-python.tar.gz'

cifar-10-python.tar 100%[===================>] 162.60M  25.9MB/s    in 6.9s

2023-03-21 16:08:02 (23.6 MB/s) - 'data/cifar-10-python.tar.gz' saved
[170498071/170498071]

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
```

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

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

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

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

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

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

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

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

[4]:
```
class Generator(nn.Module):
    def __init__(self, starting_shape):
        super(Generator, self).__init__()
        self.fc = nn.Linear(starting_shape, 4 * 4 * 128)
        self.upsample_and_generate = nn.Sequential(
            nn.BatchNorm2d(128),
            nn.LeakyReLU(),
            nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=4,␣
 ↪stride=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
```

[5]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.downsample = nn.Sequential(
```

```
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=4, stride=2,␣
↪padding=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
```

[22]:
```
# Use this to put tensors on GPU/CPU automatically when defining tensors
device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')

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

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

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


        ␣
↪#########################################################################################
        # Prob 2-1: Define the generator and discriminator, and loss functions ␣
↪#
        # Also, apply the custom weight initialization (see link:                ␣
↪       #
        # https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html)      ␣
↪       #
```

```
        ␣
↪#########################################################################################
        # To-Do: Initialize generator and discriminator
        # use variable name "self._generator" and "self._discriminator",␣
↪respectively
        # (also move them to torch device for accelerating the training later)
        self._generator = Generator(self.code_size).to(device)
        self._discriminator = Discriminator().to(device)

        # 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_optimizer = torch.optim.Adam(self._generator.
↪parameters(), lr=self.learning_rate, betas=betas)
        self._discriminator_optimizer = torch.optim.Adam(self._discriminator.
↪parameters(), lr=self.learning_rate, betas=betas)


    # To-Do: Define weight initialization function
    # see link: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
    def _weight_initialization(self, 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, inputs, labels):
        criterion = nn.BCEWithLogitsLoss()
```

```
        return criterion(inputs, labels)

    ␣
↪#############################################################################
        #                                 END OF YOUR CODE                      ␣
↪       #
    ␣
↪#############################################################################

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

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

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

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

            ␣
↪#############################################################################
                # Prob 2-1: Train the discriminator on all real images first  ␣
↪       #
            ␣
↪#############################################################################
                # To-Do: HINT: Remember to eliminate all discriminator␣
↪gradients first! (.zero_grad())
                # To-Do: feed real samples to the discriminator
```

```
                # To-Do: calculate the discriminator loss for real samples
                # use the variable name "real_dis_loss"
                self._discriminator_optimizer.zero_grad()
                real_dis_output = self._discriminator(batch_samples)
                real_dis_loss = self._classification_loss(real_dis_output,␣
↪real_label)
                real_dis_loss.backward()

            ␣
↪#############################################################################
                # 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
                # To-Do: generate fake samples from the noise using the␣
↪generator
                # 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

                noise = torch.normal(0,1,(self.batch_size,self.code_size)).
↪to(device)
                fake_gen_output = self._generator(noise)
                fake_dis_output = self._discriminator(fake_gen_output.detach())

                # To-Do: calculate the discriminator loss for fake samples
                # use the variable name "fake_dis_loss"
                # To-Do: calculate the total discriminator loss (real loss +␣
↪fake loss)
                # To-Do: calculate the gradients for the total discriminator␣
↪loss
                # To-Do: update the discriminator weights
                fake_dis_loss = self._classification_loss(fake_dis_output,␣
↪fake_label)
                fake_dis_loss.backward()
                self._discriminator_optimizer.step()

            ␣
↪#############################################################################
```

```
                # Prob 2-1: Train the generator                                 ␣
↪                #
              ␣
↪#########################################################################################␣
↪
                # To-Do: Remember to eliminate all generator gradients first! (.
↪zero_grad())
                # To-Do: sample noises from IID Normal(0, 1)^d on the torch␣
↪device
                # To-Do: generate fake samples from the noise using the␣
↪generator
                # 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.
                self._generator_optimizer.zero_grad()
                fake_gen_output = self._generator(noise)
                fake_dis_output = self._discriminator(fake_gen_output)

                # To-Do: calculate the generator loss
                # hint: the goal of the generator is to make the discriminator
                # consider the fake samples as real
                # To-Do: Calculate the generator loss gradients
                # To-Do: Update the generator weights
                gen_loss = self._classification_loss(fake_dis_output,␣
↪real_label)
                gen_loss.backward()
                self._generator_optimizer.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, inputs, labels):
      l2_loss = nn.MSELoss()
      return l2_loss(inputs, labels)



  def reconstruct(self, samples):
      recon_code = torch.zeros([samples.shape[0], self.code_size],␣
↪device=device, requires_grad=True)
```

```
        samples = torch.tensor(samples, device=device, dtype=torch.float32)

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

      ␣
↪#############################################################################
        # Prob 2-4: complete the definition of the optimizer.                 ␣
↪       #
        # **skip this part when working on problem 2-1 and come back for␣
↪problem 2-4   #
      ␣
↪#############################################################################␣
↪
        # To-Do: define the optimizer
        # Hinit: Use self.vis_learning_rate as one of the parameters for Adam␣
↪optimizer
        recon_optimizer = torch.optim.Adam([recon_code], lr=self.
↪vis_learning_rate)


        for i in range(500):
          ␣
↪#############################################################################
          # Prob 2-4: Fill in the training loop for reconstruciton            ␣
↪          #
          # **skip this part when working on problem 2-1 and come back for␣
↪problem 2-4   #
          ␣
↪#############################################################################
          # To-Do: eliminate the gradients
          recon_optimizer.zero_grad()

          # To-Do: feed the reconstruction codes to the generator for␣
↪generating reconstructed samples
          # use the variable name "recon_samples"

          recon_samples = self._generator(recon_code)


          # To-Do: calculate reconstruction loss
          # use the variable name "recon_loss"
          recon_loss = self._reconstruction_loss(recon_samples, samples)
```

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

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


            ␣
↪#############################################################################
            #                                END OF YOUR CODE                             ␣
↪            #
            ␣
↪#############################################################################

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


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

Now let's do the training!

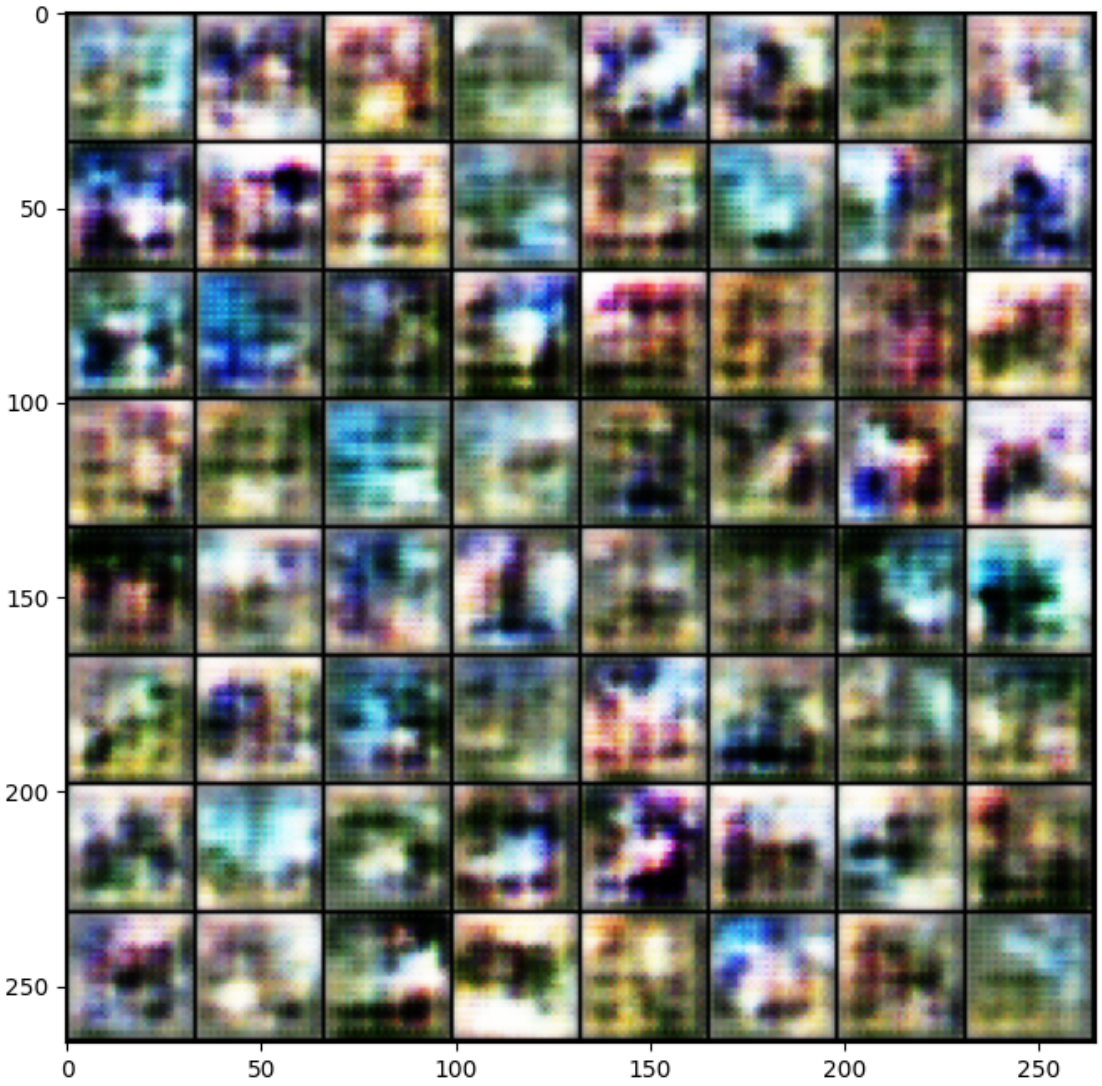
Don't panic if the loss curve goes wild. The two networks are competing for the loss curve to go different directions, so virtually anything can happen. If your code is correct, the generated samples

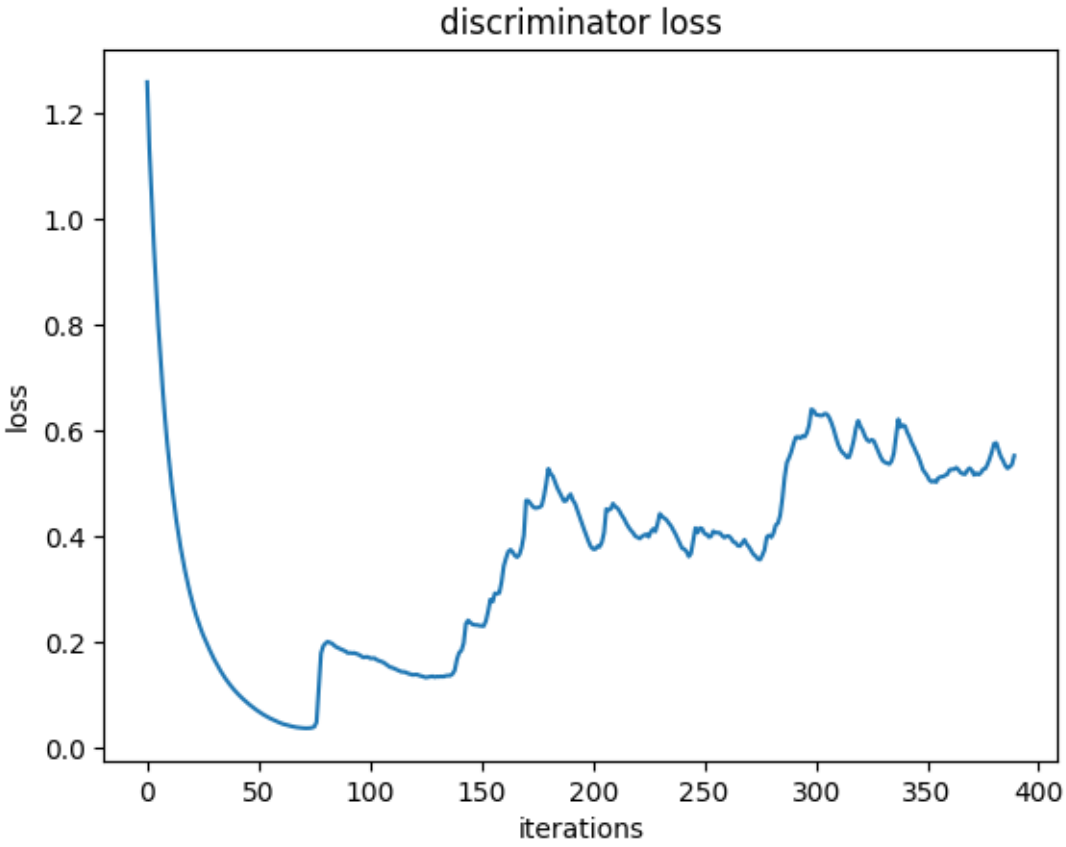
should have a high variety.

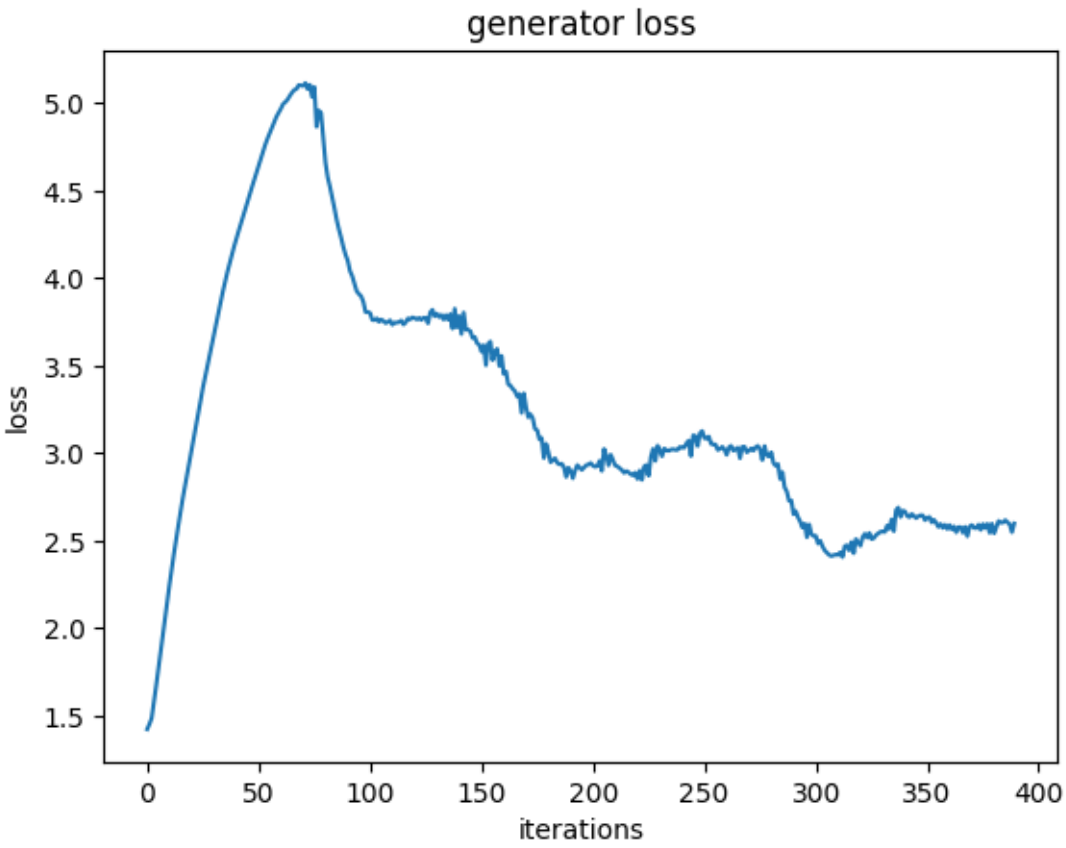
Do NOT change the number of epochs, learning rate, or batch size. If you're using Google Colab, the batch size will not be an issue during training.

[13]:
```
set_seed(42)

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

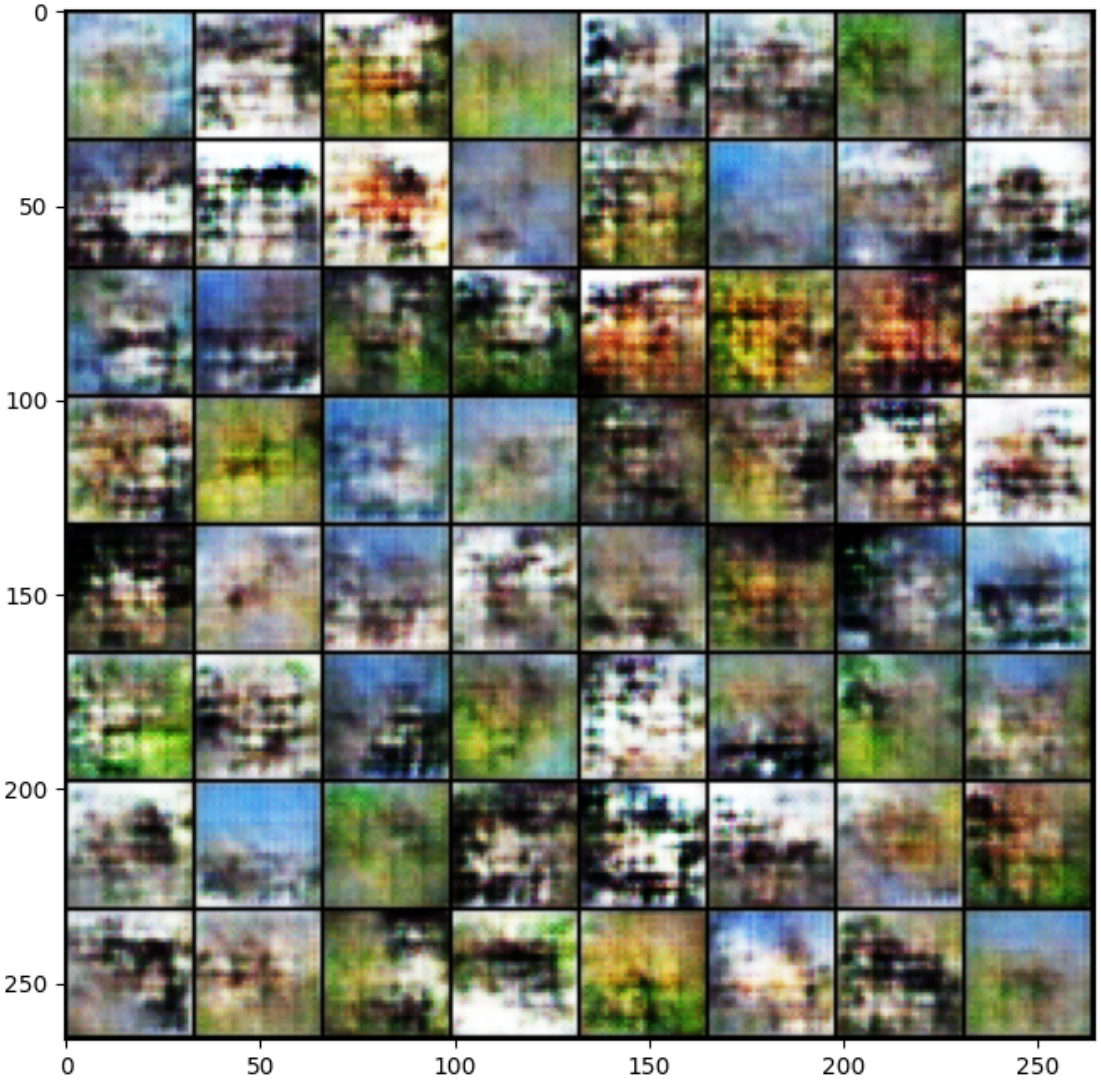
```
Start training …
Iteration 100/9750: dis loss = 0.1715, gen loss = 3.8024
Iteration 200/9750: dis loss = 0.1975, gen loss = 3.1074
Iteration 300/9750: dis loss = 0.5975, gen loss = 2.3079
```

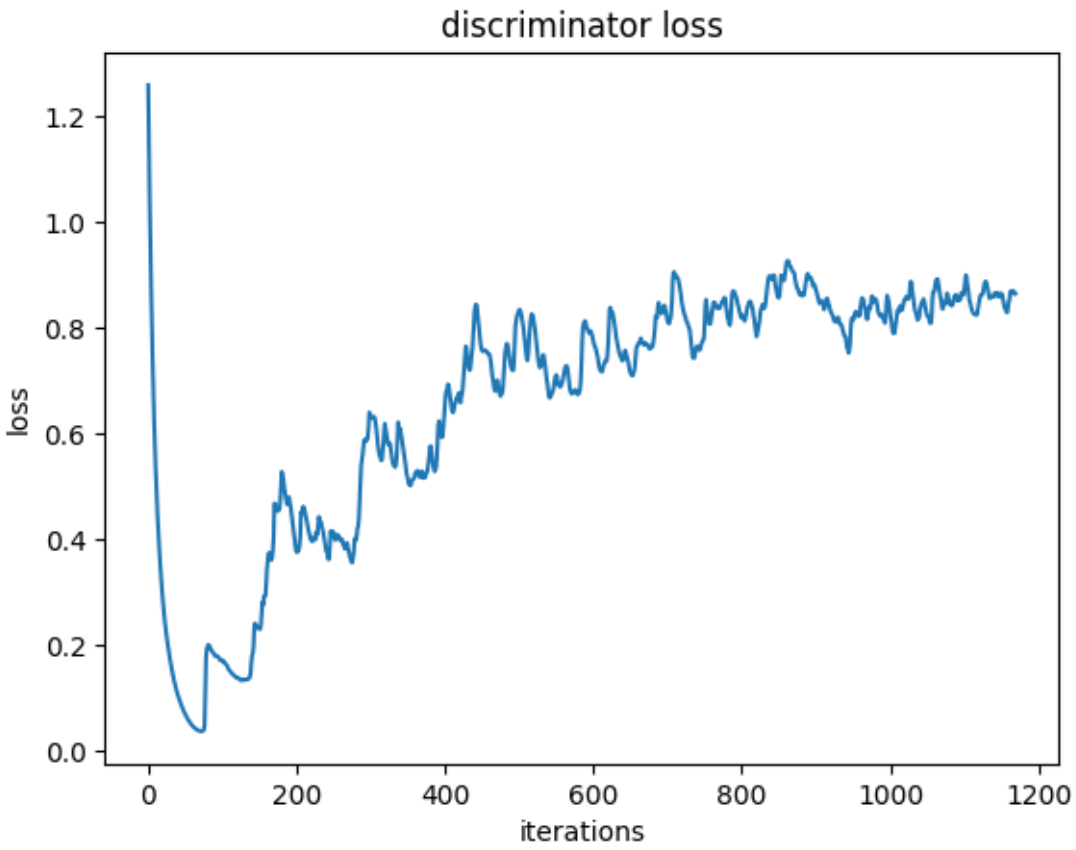



discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
50
100
150
200
250
300
350
400

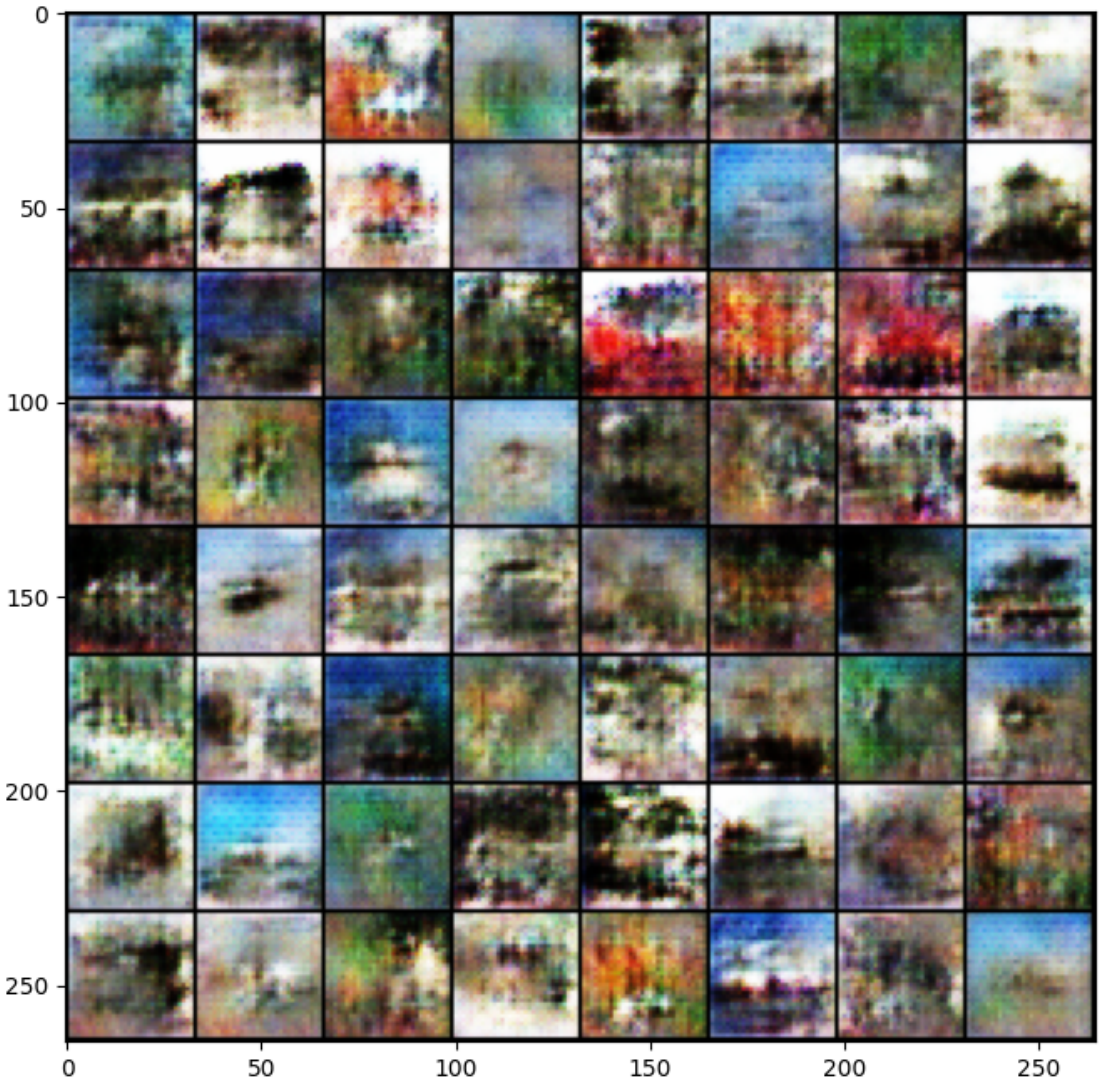


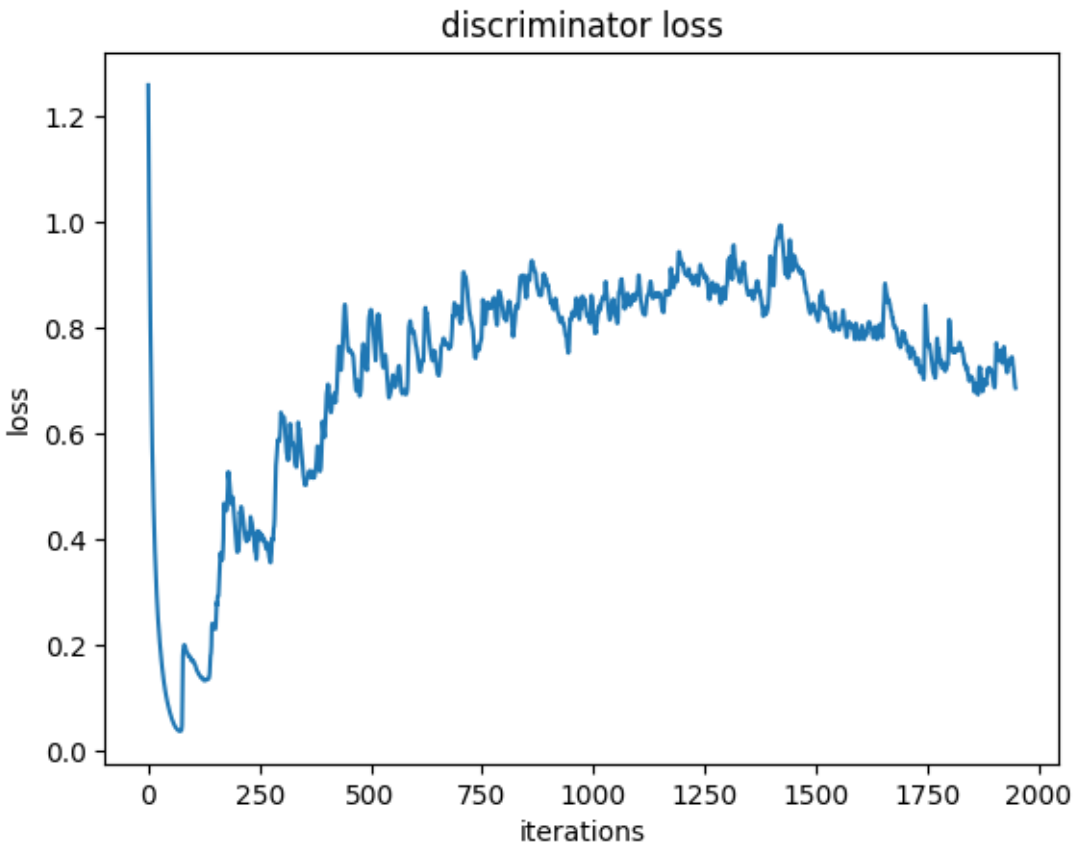

```
Iteration 400/9750: dis loss = 1.0320, gen loss = 3.3922
Iteration 500/9750: dis loss = 0.9234, gen loss = 1.6478
Iteration 600/9750: dis loss = 0.6451, gen loss = 1.8004
Iteration 700/9750: dis loss = 0.6732, gen loss = 1.9062
Iteration 800/9750: dis loss = 0.6949, gen loss = 2.4928
Iteration 900/9750: dis loss = 0.7751, gen loss = 1.3028
Iteration 1000/9750: dis loss = 0.6556, gen loss = 1.8348
Iteration 1100/9750: dis loss = 0.8228, gen loss = 2.0917
```

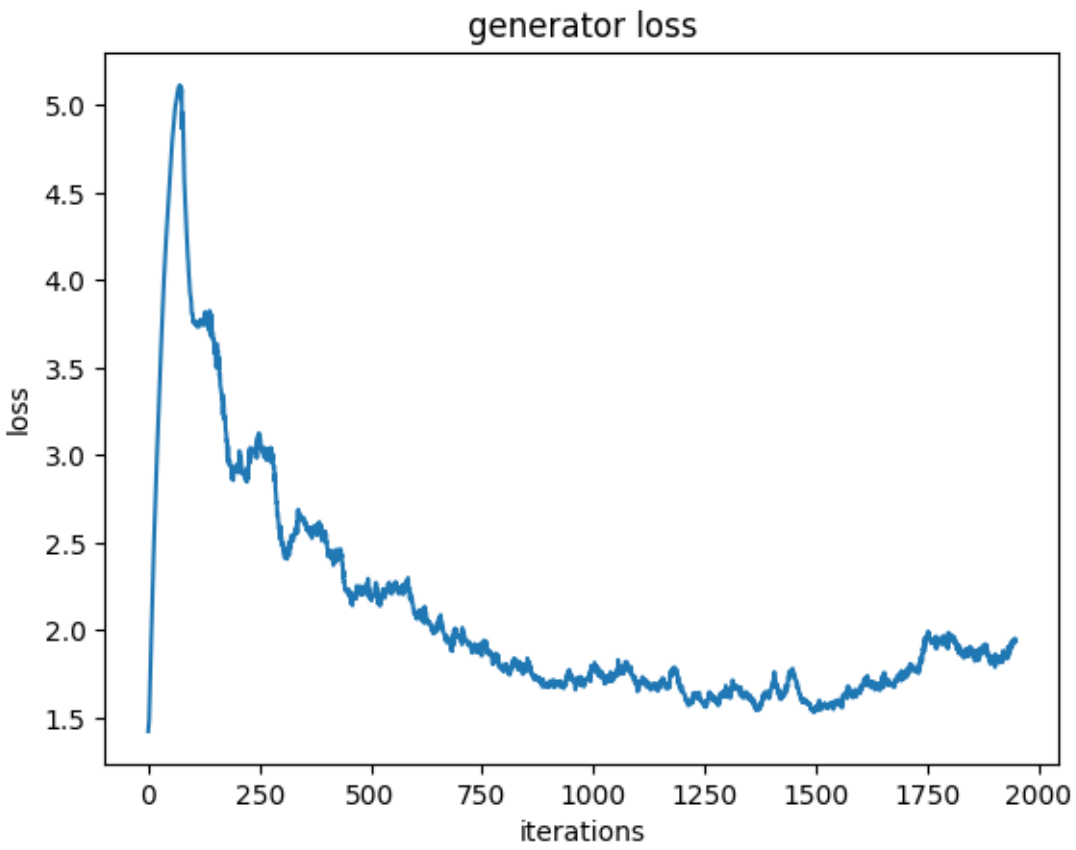

0
50
100
150
200
250
0
50
100
150
200
250


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

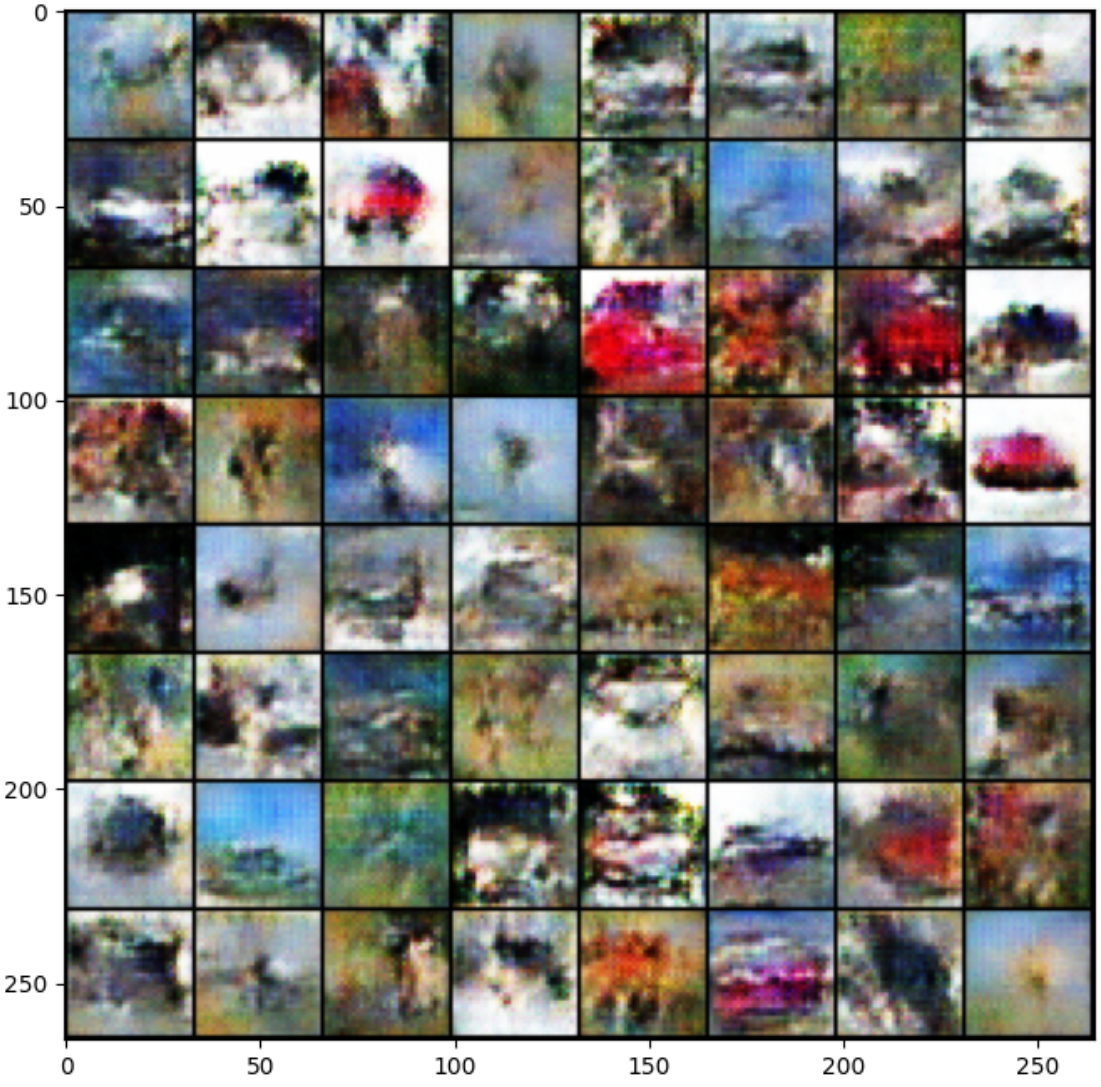

```
Iteration 1200/9750: dis loss = 0.8615, gen loss = 2.2946
Iteration 1300/9750: dis loss = 1.0936, gen loss = 1.1880
Iteration 1400/9750: dis loss = 0.7637, gen loss = 1.7356
Iteration 1500/9750: dis loss = 0.8194, gen loss = 1.9065
Iteration 1600/9750: dis loss = 0.7268, gen loss = 1.8283
Iteration 1700/9750: dis loss = 0.8615, gen loss = 1.2928
Iteration 1800/9750: dis loss = 1.4959, gen loss = 0.7431
Iteration 1900/9750: dis loss = 0.5630, gen loss = 2.0664
```

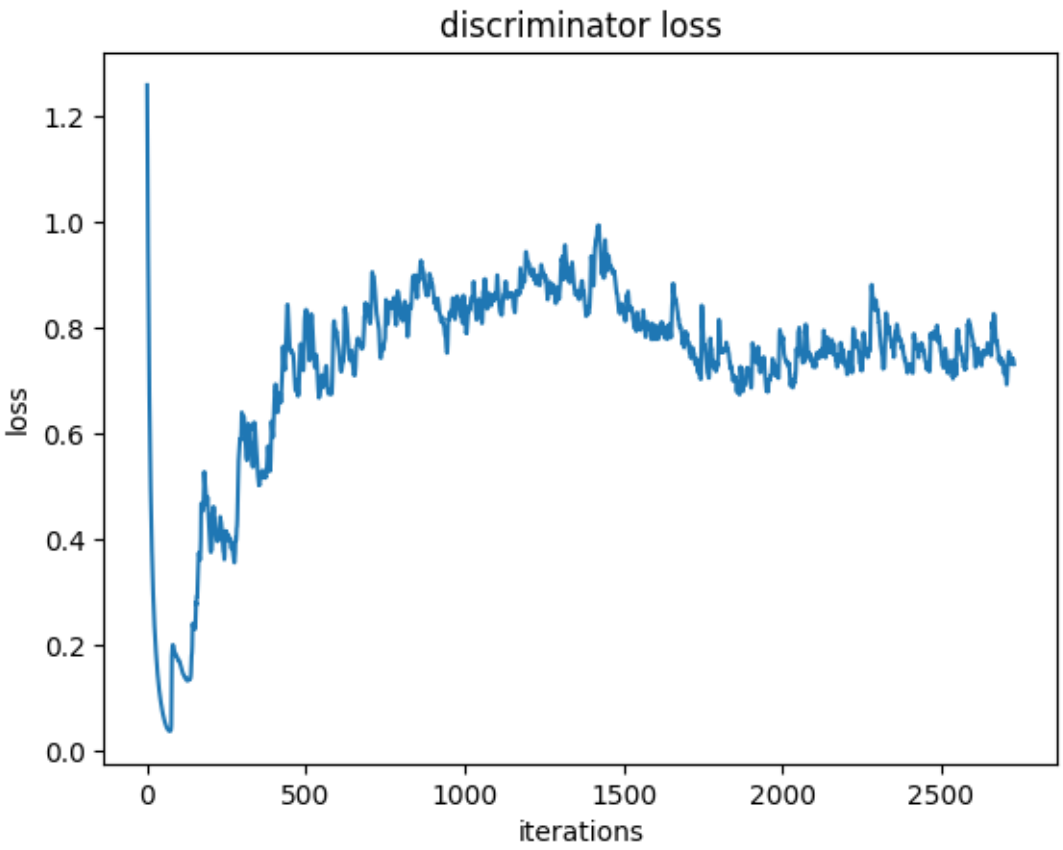

0
50
100
150
200
250
0
50
100
150
200
250

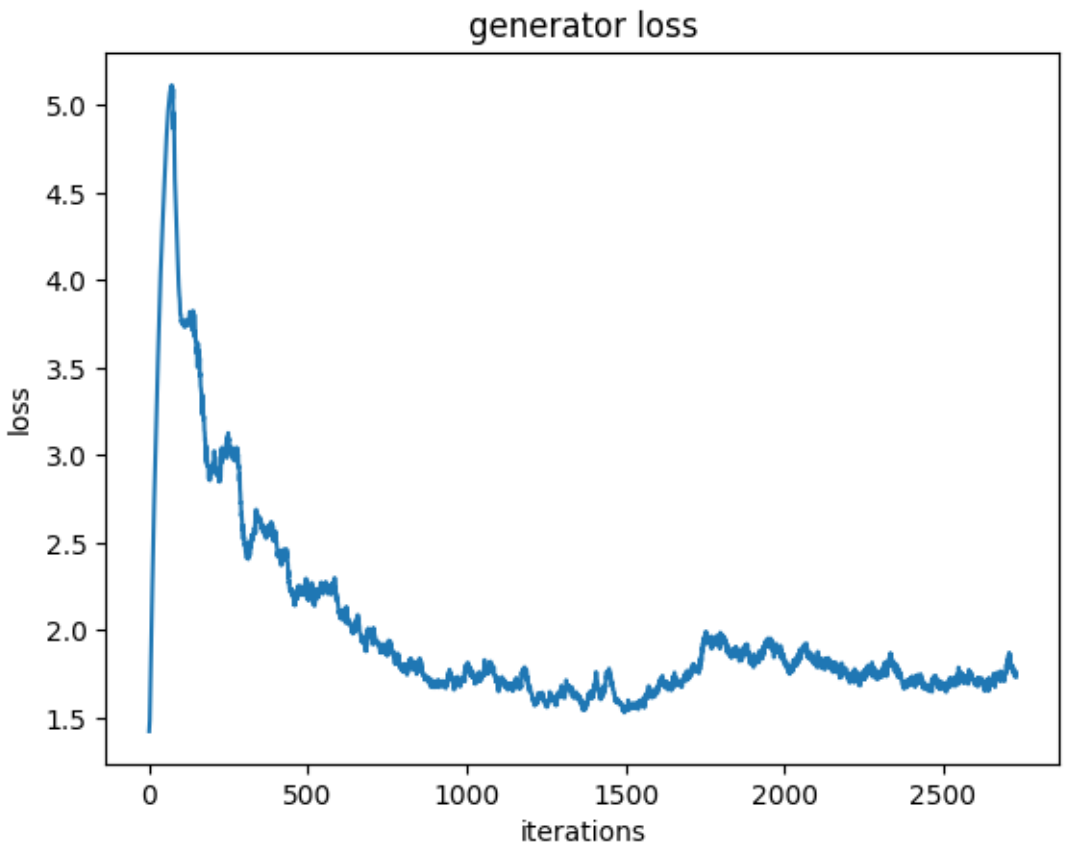

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
250
500
750
1000
1250
1500
1750
2000



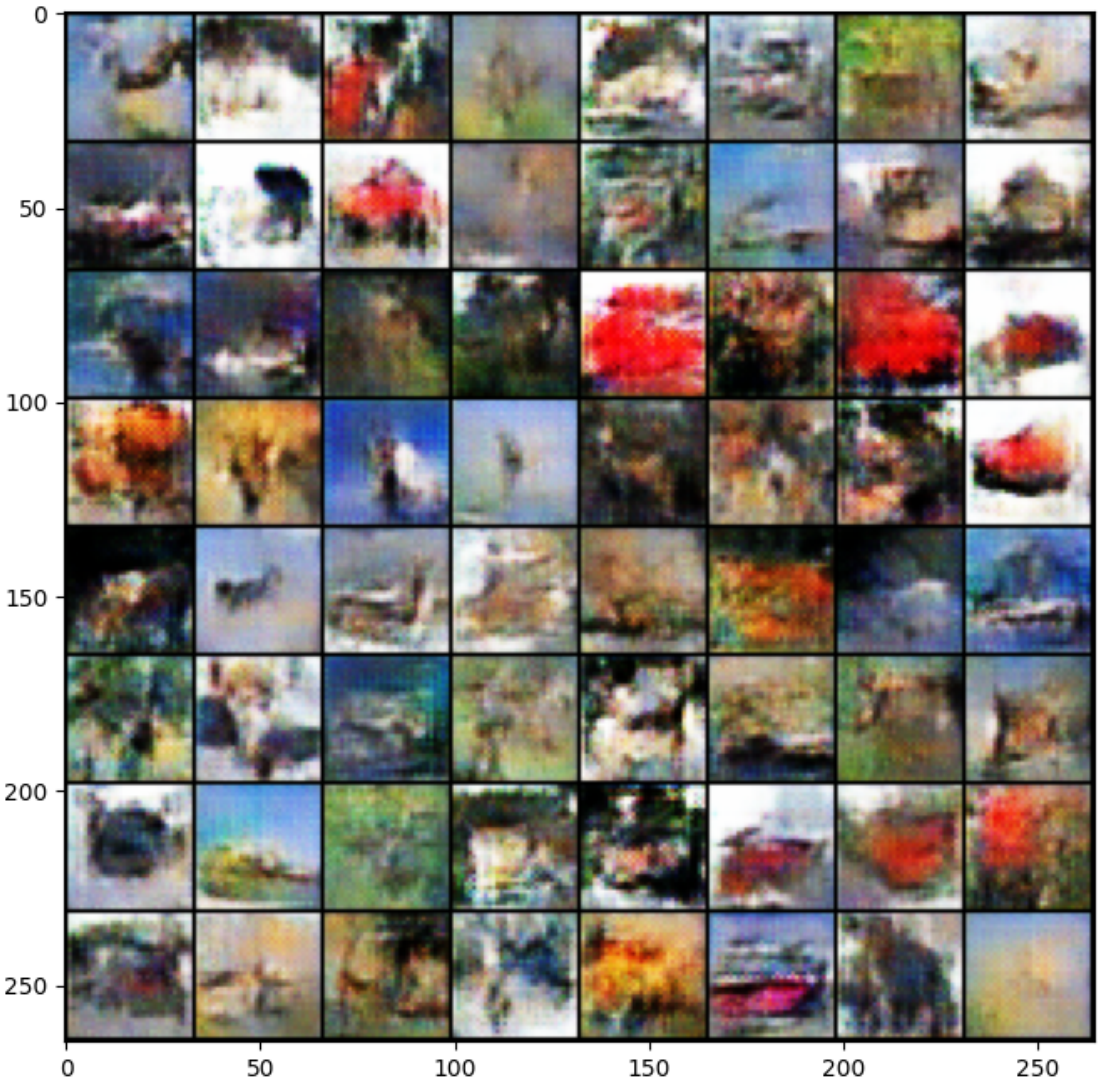

```
Iteration 2000/9750: dis loss = 0.8362, gen loss = 1.2202
Iteration 2100/9750: dis loss = 0.7789, gen loss = 1.5455
Iteration 2200/9750: dis loss = 0.6993, gen loss = 1.3049
Iteration 2300/9750: dis loss = 0.8345, gen loss = 1.9034
Iteration 2400/9750: dis loss = 0.7424, gen loss = 1.9809
Iteration 2500/9750: dis loss = 0.7230, gen loss = 1.1669
Iteration 2600/9750: dis loss = 0.7646, gen loss = 1.2956
Iteration 2700/9750: dis loss = 0.7001, gen loss = 2.7234
```

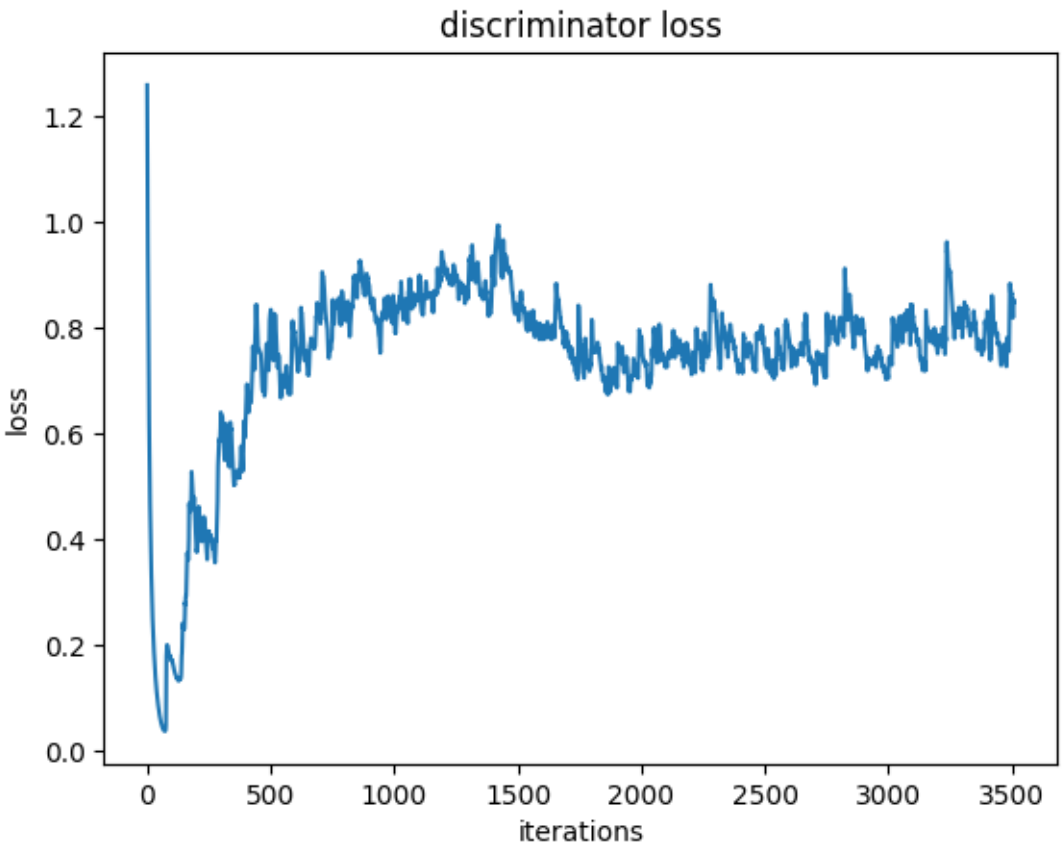

0
50
100
150
200
250
0
50
100
150
200
250

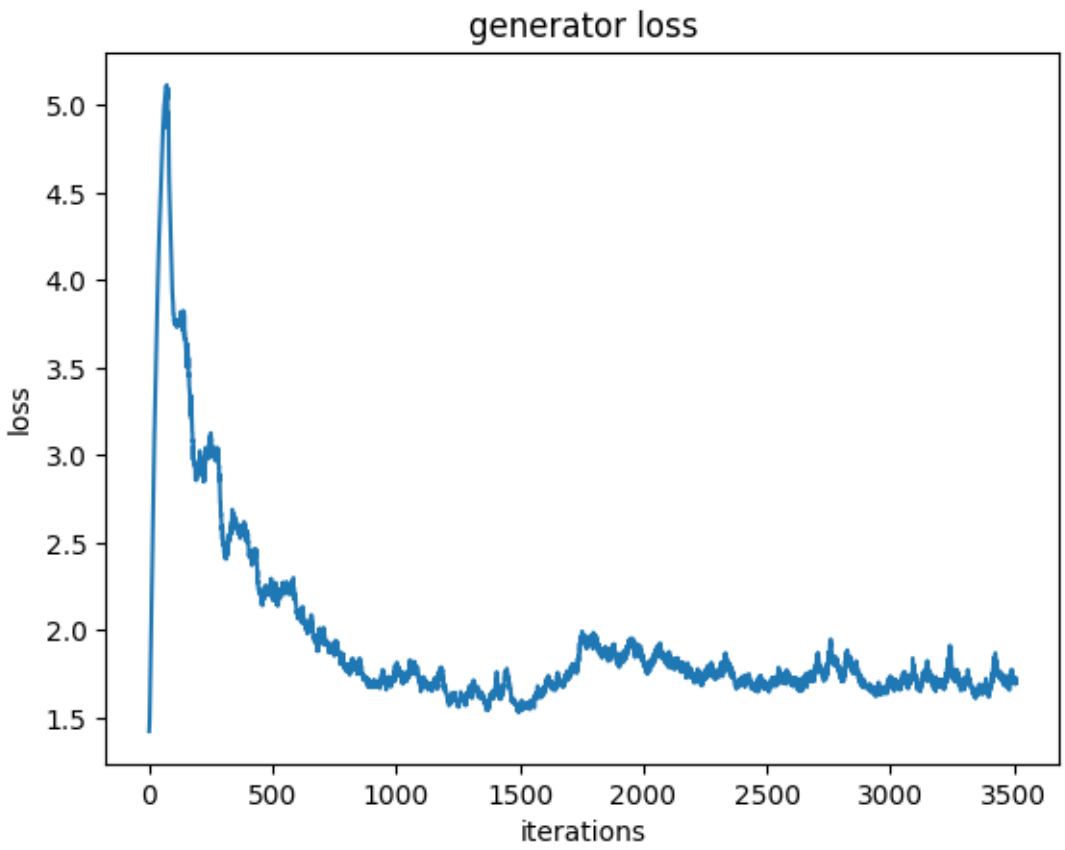

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
500
1000
1500
2000
2500




```
Iteration 2800/9750: dis loss = 1.0692, gen loss = 0.7866
Iteration 2900/9750: dis loss = 0.5644, gen loss = 1.8295
Iteration 3000/9750: dis loss = 0.5984, gen loss = 1.2715
Iteration 3100/9750: dis loss = 0.7622, gen loss = 1.4246
Iteration 3200/9750: dis loss = 0.7400, gen loss = 1.3327
Iteration 3300/9750: dis loss = 0.5669, gen loss = 2.0666
Iteration 3400/9750: dis loss = 0.9010, gen loss = 0.9449
Iteration 3500/9750: dis loss = 0.8023, gen loss = 1.1018
```

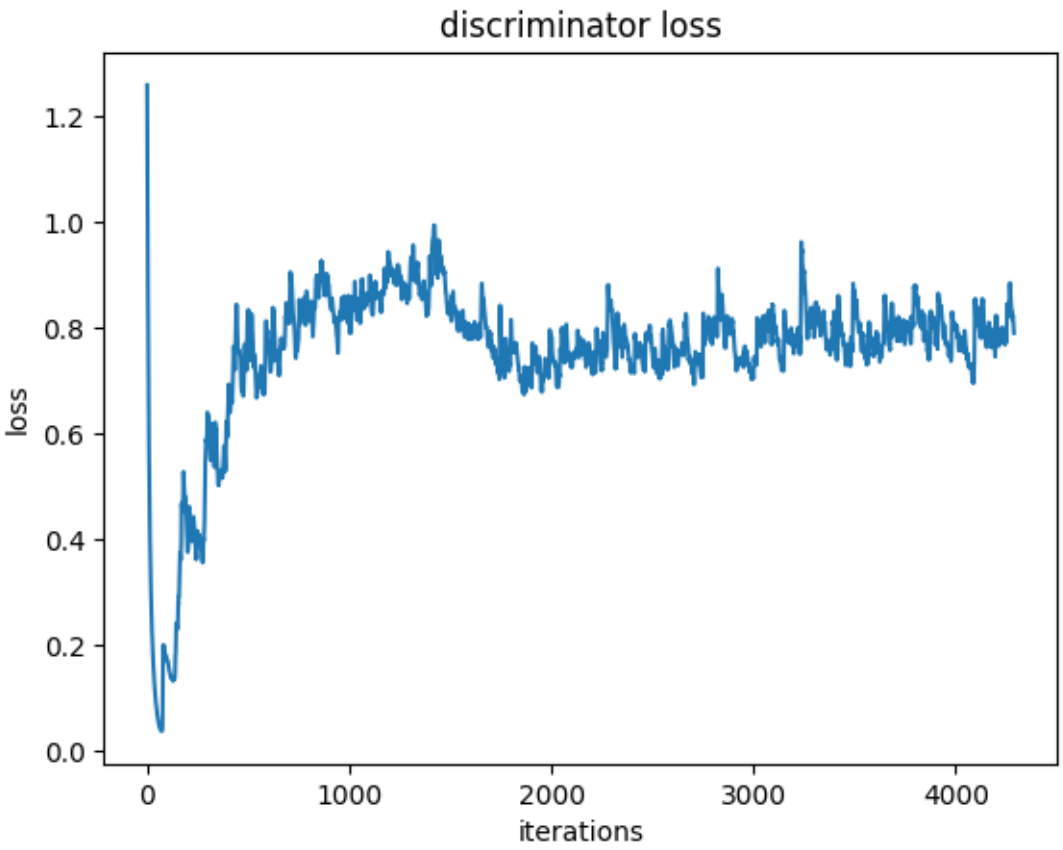

0
50
100
150
200
250
0
50
100
150
200
250

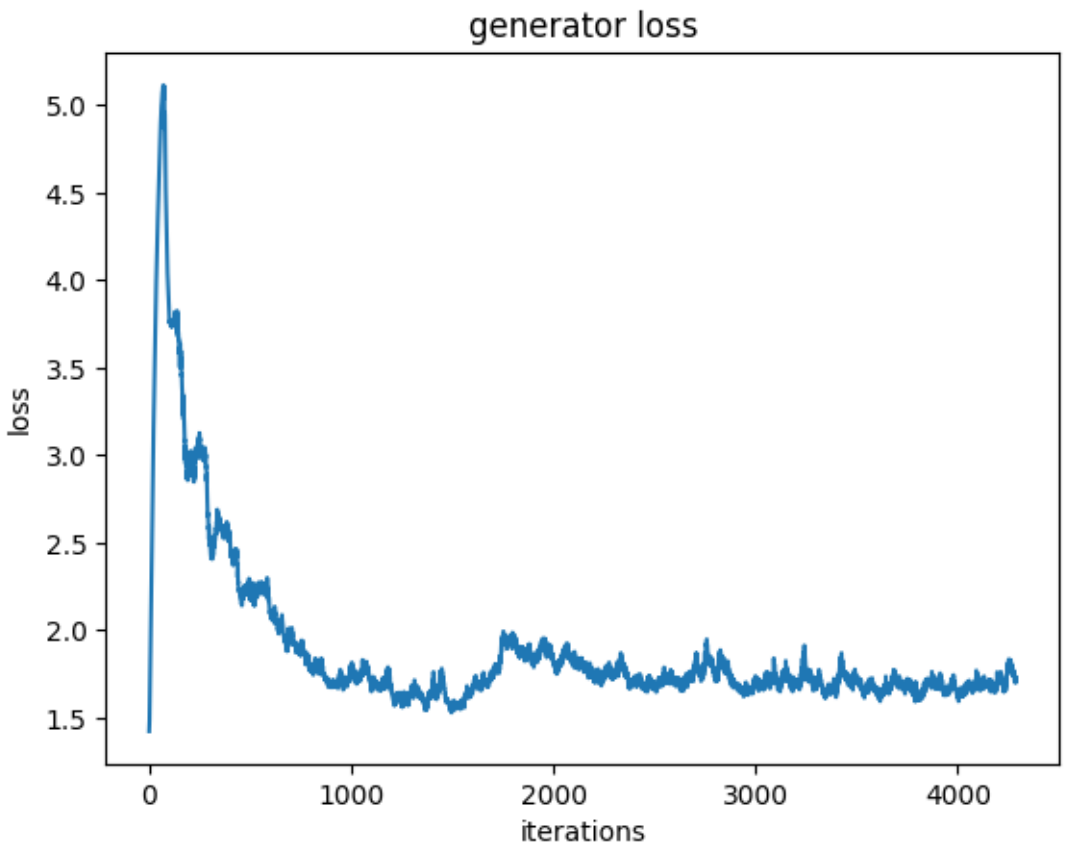

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
500
1000
1500
2000
2500
3000
3500



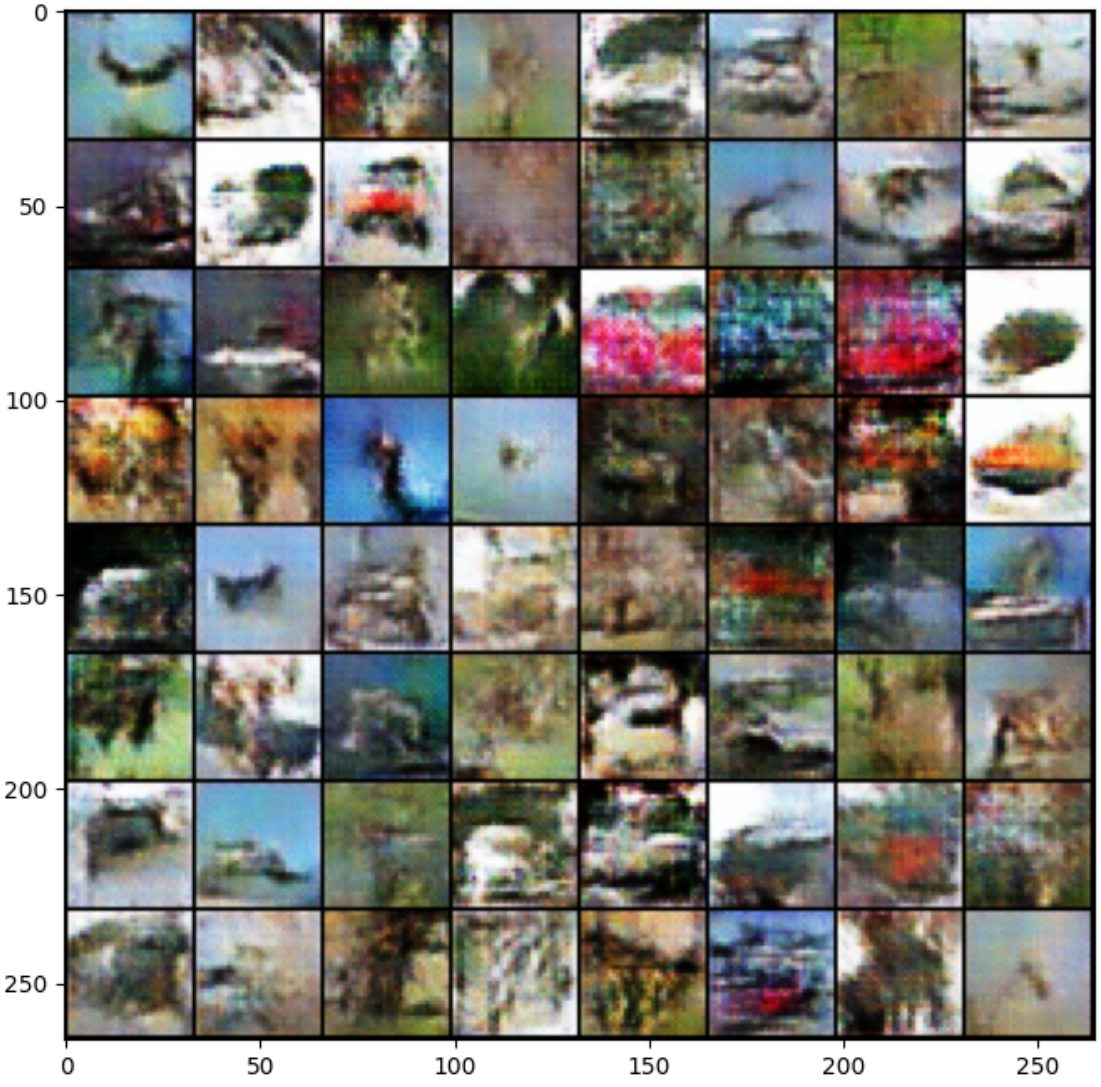

```
Iteration 3600/9750: dis loss = 0.8973, gen loss = 1.1151
Iteration 3700/9750: dis loss = 0.6862, gen loss = 1.8275
Iteration 3800/9750: dis loss = 0.9367, gen loss = 0.8085
Iteration 3900/9750: dis loss = 0.4030, gen loss = 2.4684
Iteration 4000/9750: dis loss = 0.7597, gen loss = 1.7252
Iteration 4100/9750: dis loss = 0.5507, gen loss = 2.4119
Iteration 4200/9750: dis loss = 0.9700, gen loss = 1.5594
```

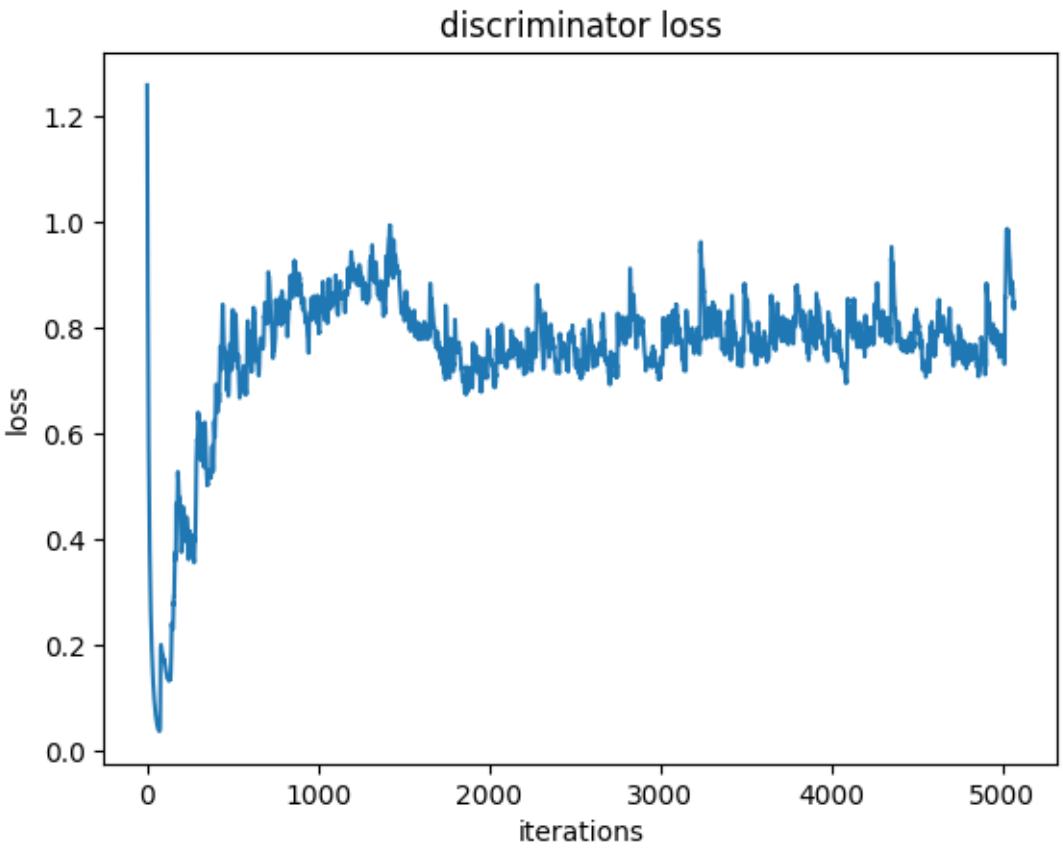

0
50
100
150
200
250
0
50
100
150
200
250

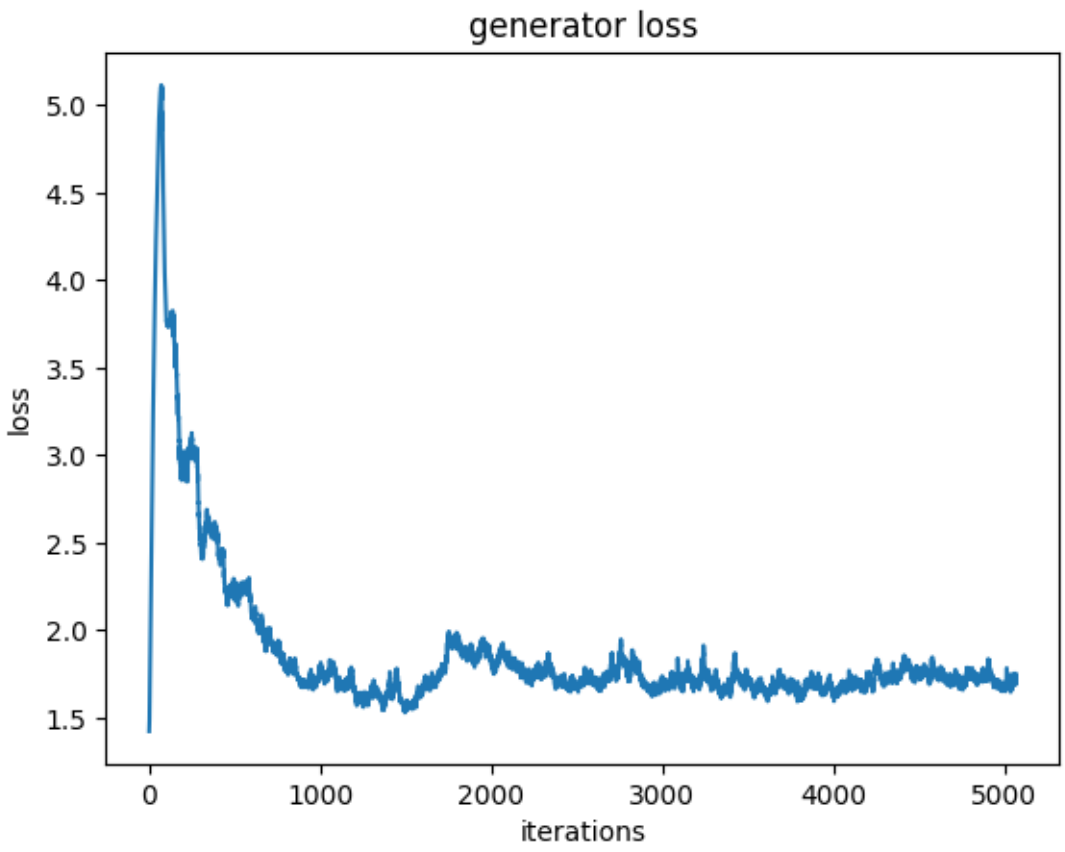

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
1000
2000
3000
4000



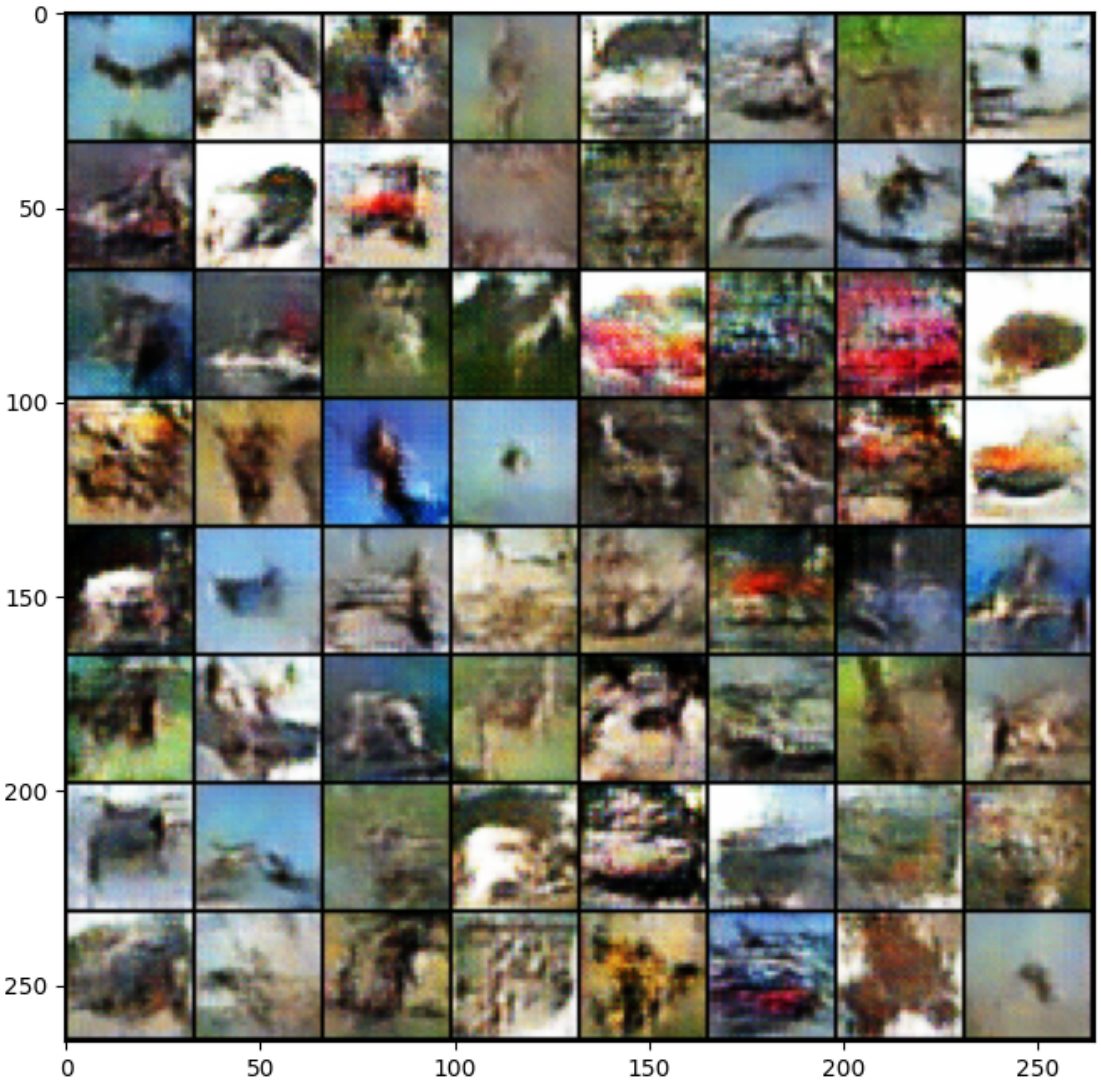

```
Iteration 4300/9750: dis loss = 0.9303, gen loss = 1.6798
Iteration 4400/9750: dis loss = 0.6704, gen loss = 1.2976
Iteration 4500/9750: dis loss = 0.8882, gen loss = 1.7568
Iteration 4600/9750: dis loss = 0.6455, gen loss = 1.9738
Iteration 4700/9750: dis loss = 0.7385, gen loss = 1.2936
Iteration 4800/9750: dis loss = 0.5776, gen loss = 1.7853
Iteration 4900/9750: dis loss = 0.5523, gen loss = 1.7323
Iteration 5000/9750: dis loss = 0.6727, gen loss = 1.9711
```

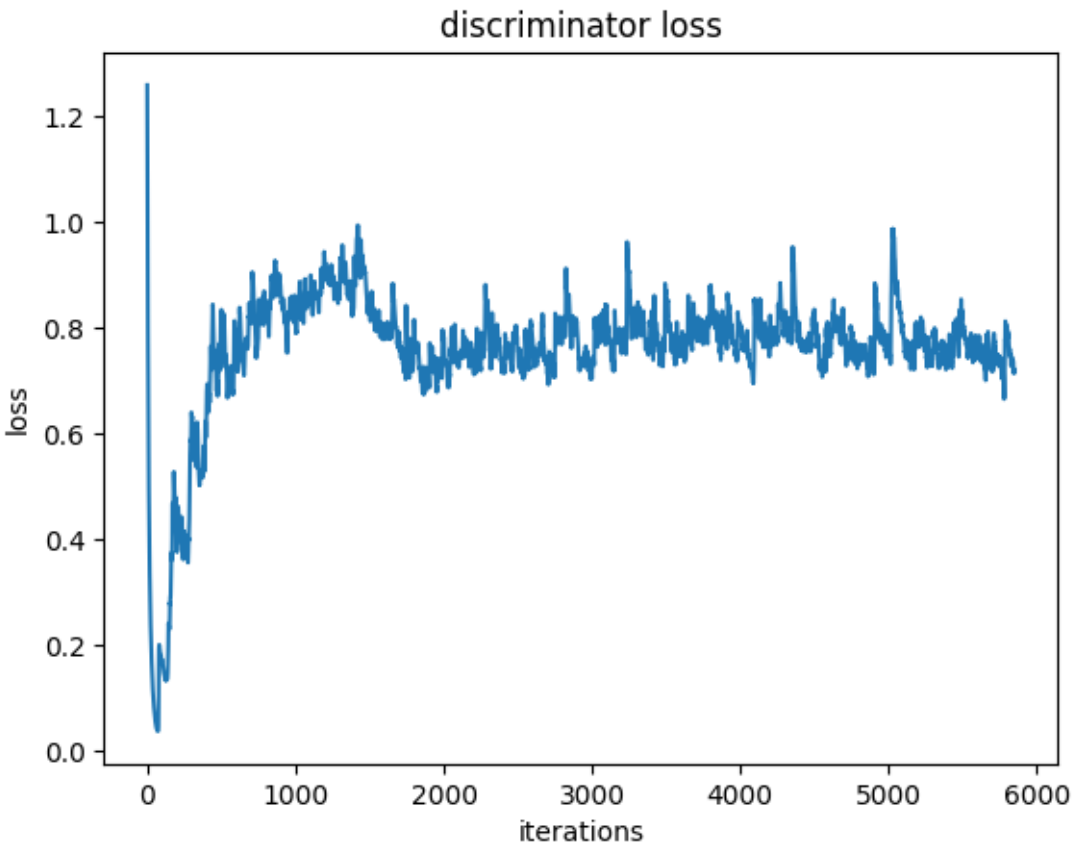

0
50
100
150
200
250
0
50
100
150
200
250

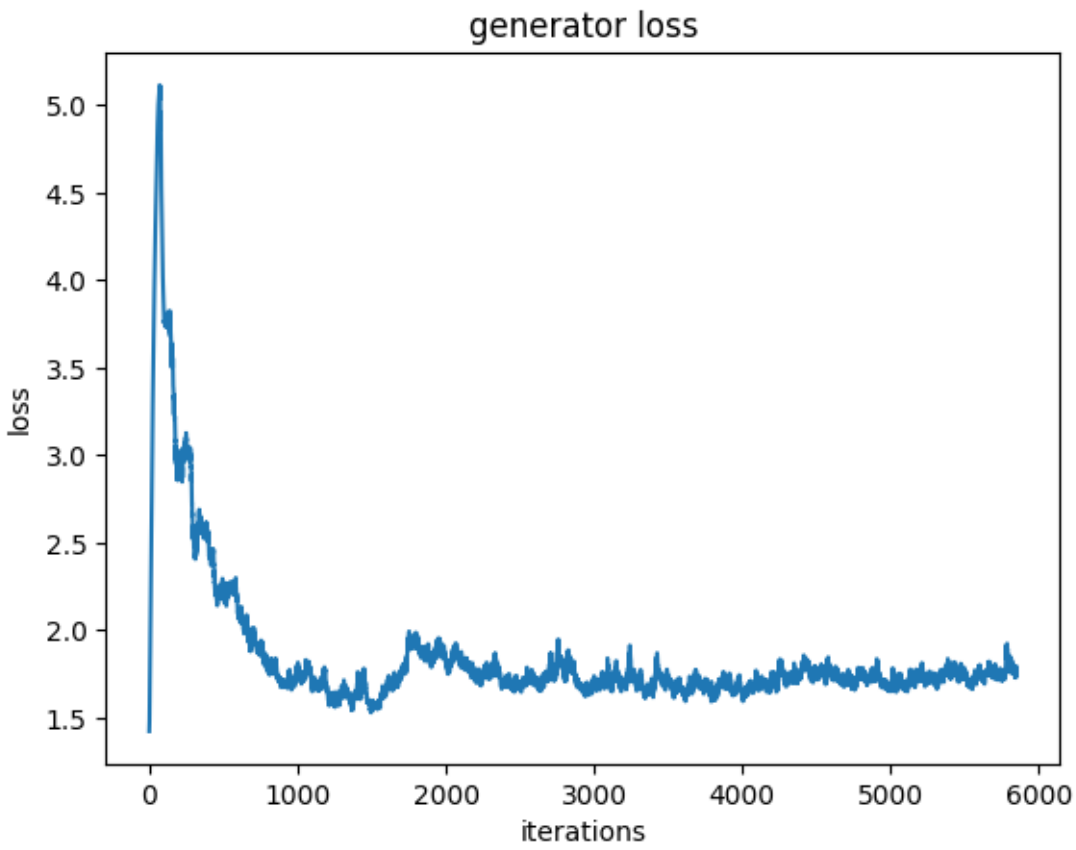

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
1000
2000
3000
4000
5000




```
Iteration 5100/9750: dis loss = 0.6875, gen loss = 1.8162
Iteration 5200/9750: dis loss = 0.8957, gen loss = 2.0733
Iteration 5300/9750: dis loss = 0.8808, gen loss = 1.1092
Iteration 5400/9750: dis loss = 0.6228, gen loss = 2.2439
Iteration 5500/9750: dis loss = 0.7840, gen loss = 1.7309
Iteration 5600/9750: dis loss = 0.9104, gen loss = 2.9168
Iteration 5700/9750: dis loss = 0.6917, gen loss = 1.4137
Iteration 5800/9750: dis loss = 0.7266, gen loss = 1.4895
```

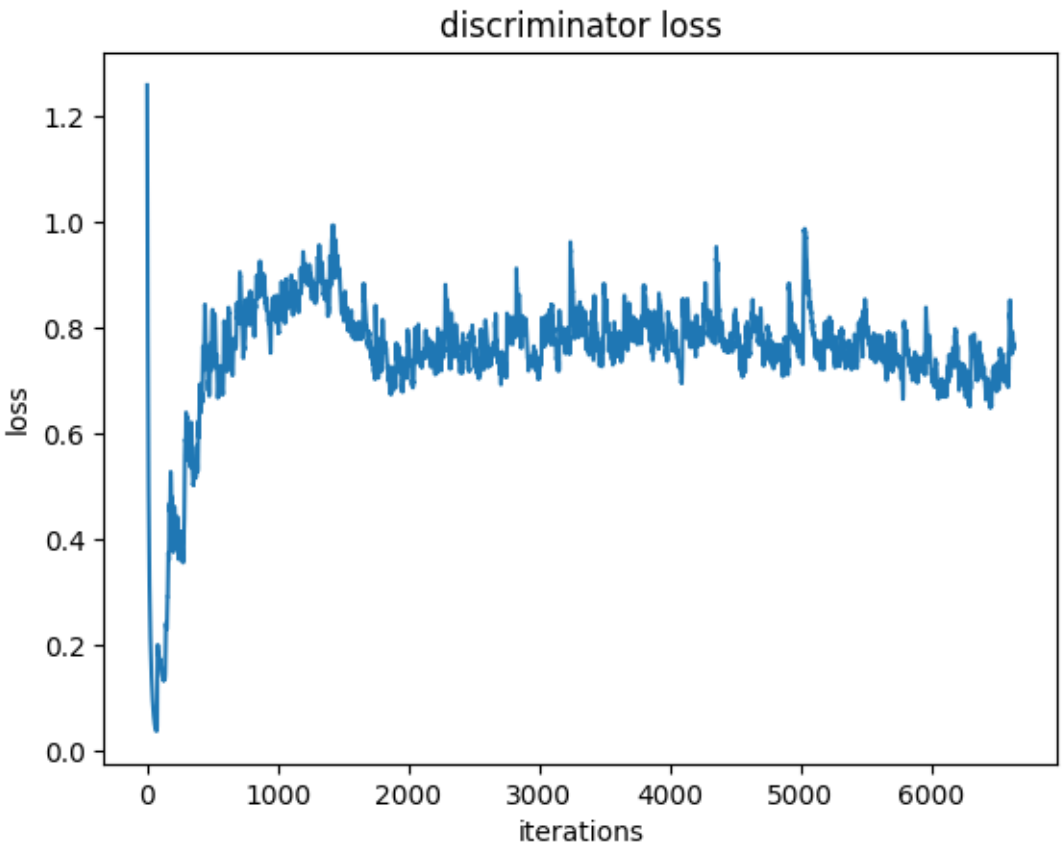

0
50
100
150
200
250
0
50
100
150
200
250

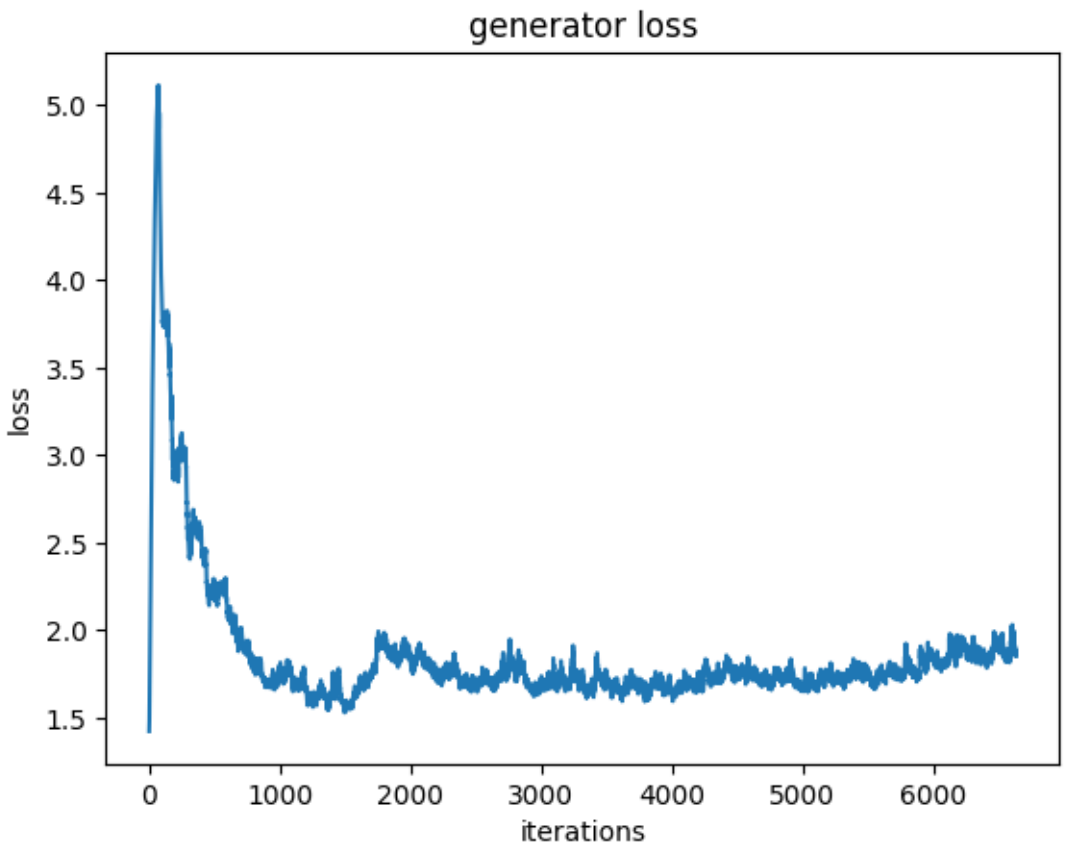

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
1000
2000
3000
4000
5000
6000



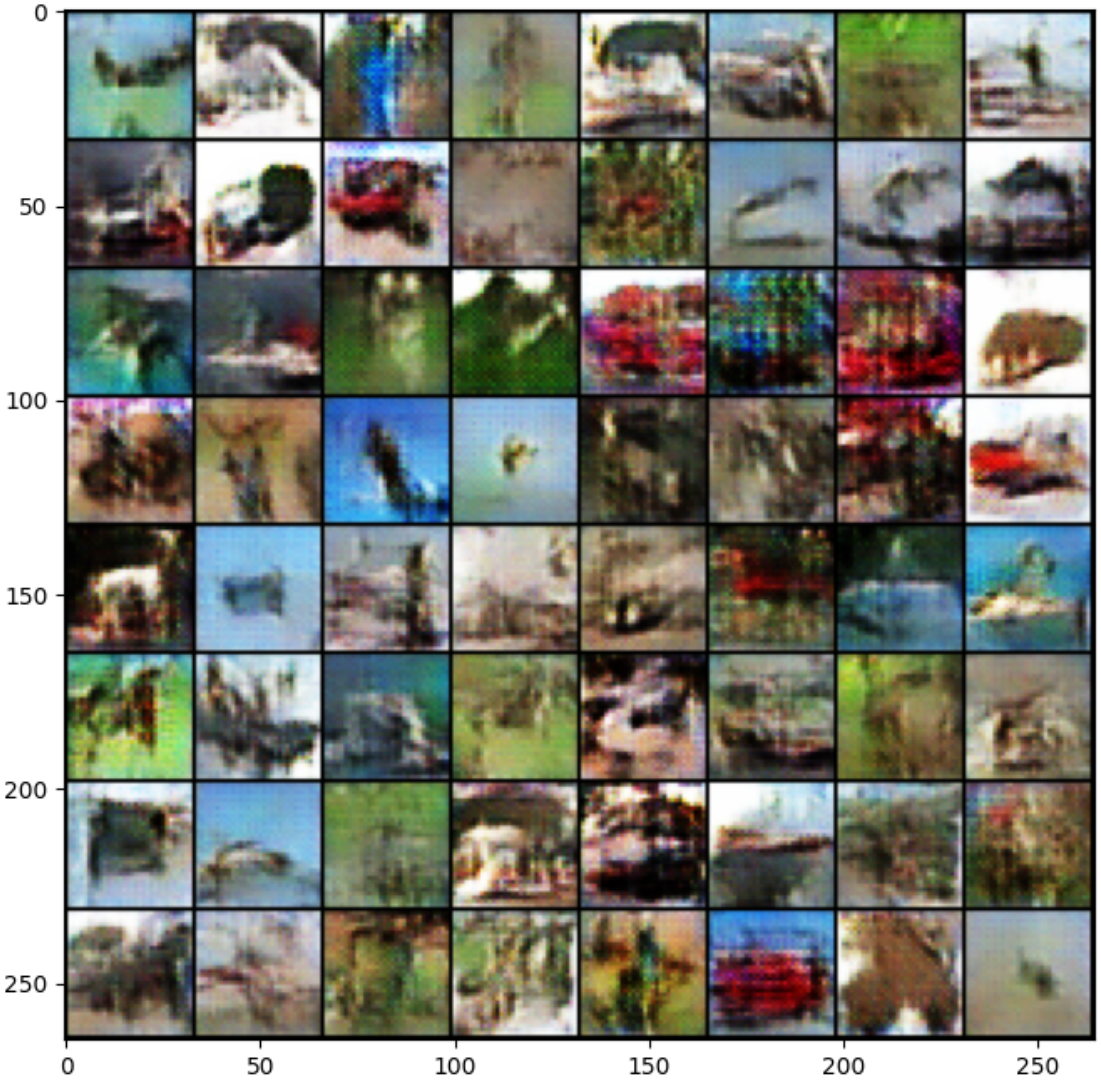

```
Iteration 5900/9750: dis loss = 0.6684, gen loss = 1.2872
Iteration 6000/9750: dis loss = 0.7906, gen loss = 0.9904
Iteration 6100/9750: dis loss = 0.6598, gen loss = 2.0952
Iteration 6200/9750: dis loss = 0.6678, gen loss = 1.8417
Iteration 6300/9750: dis loss = 1.0931, gen loss = 3.8661
Iteration 6400/9750: dis loss = 0.6833, gen loss = 2.3845
Iteration 6500/9750: dis loss = 0.6777, gen loss = 1.9726
Iteration 6600/9750: dis loss = 0.8606, gen loss = 2.2749
```

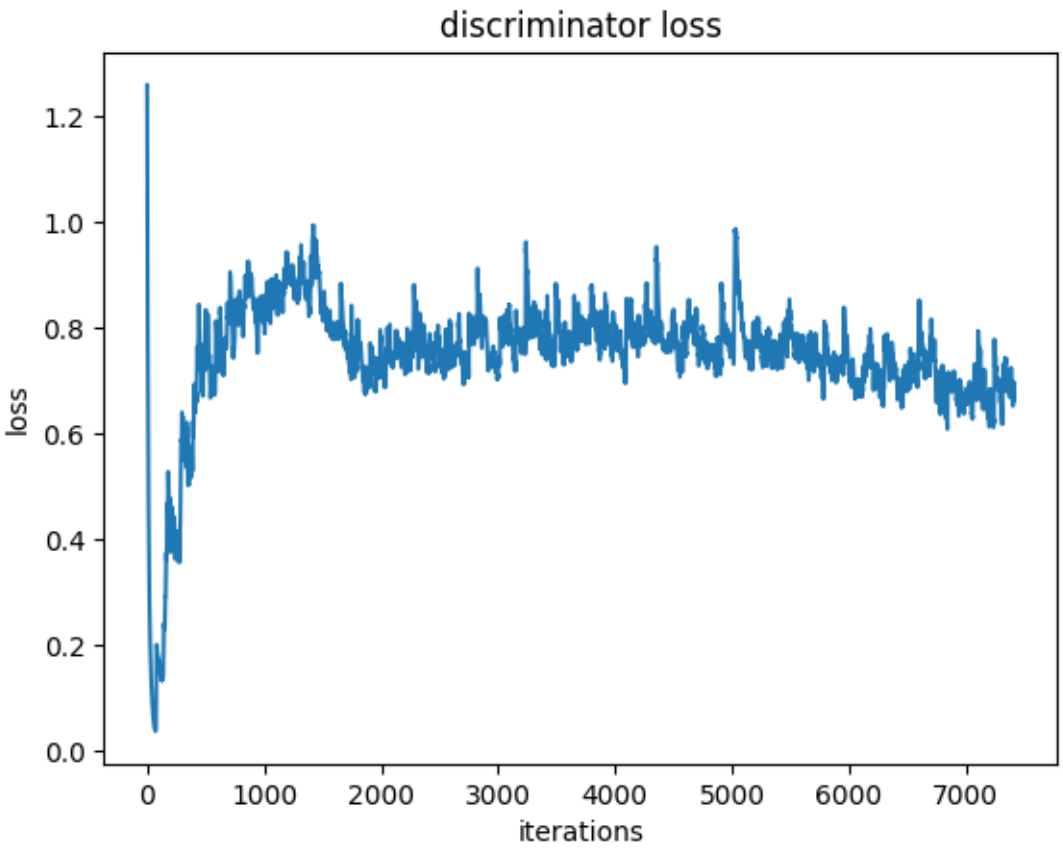

0
50
100
150
200
250
0
50
100
150
200
250

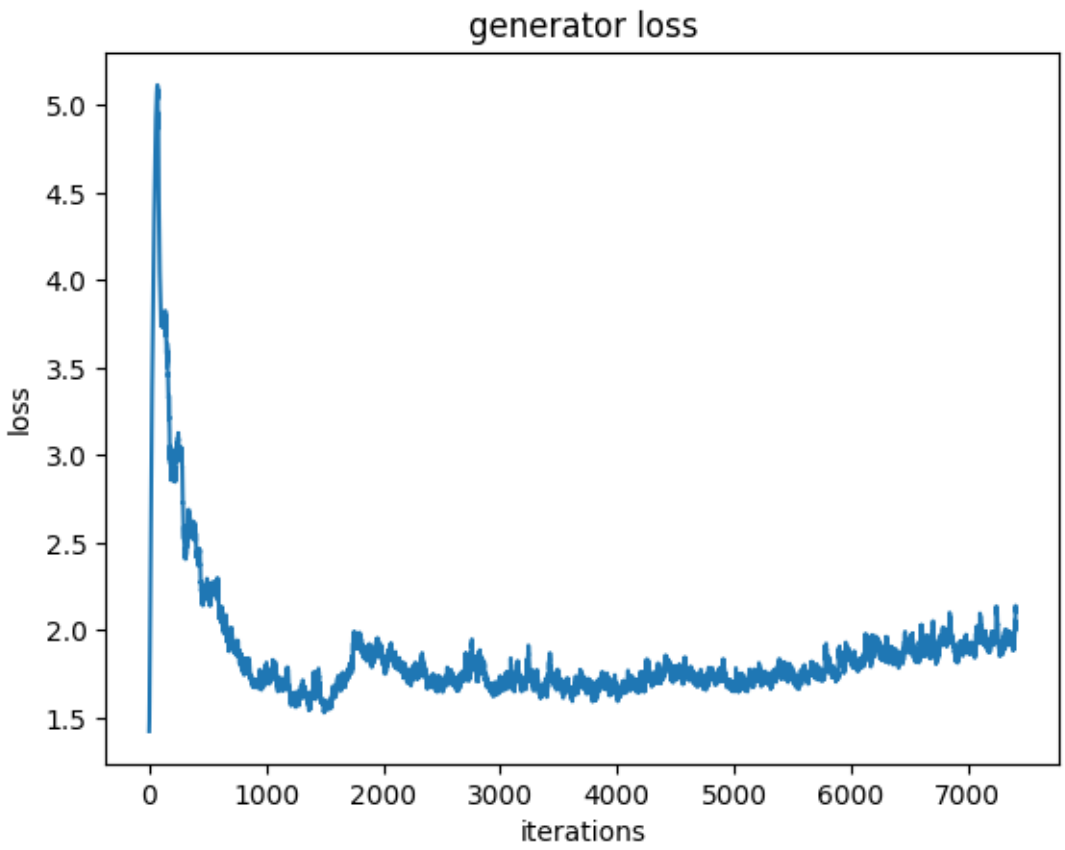

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
1000
2000
3000
4000
5000
6000



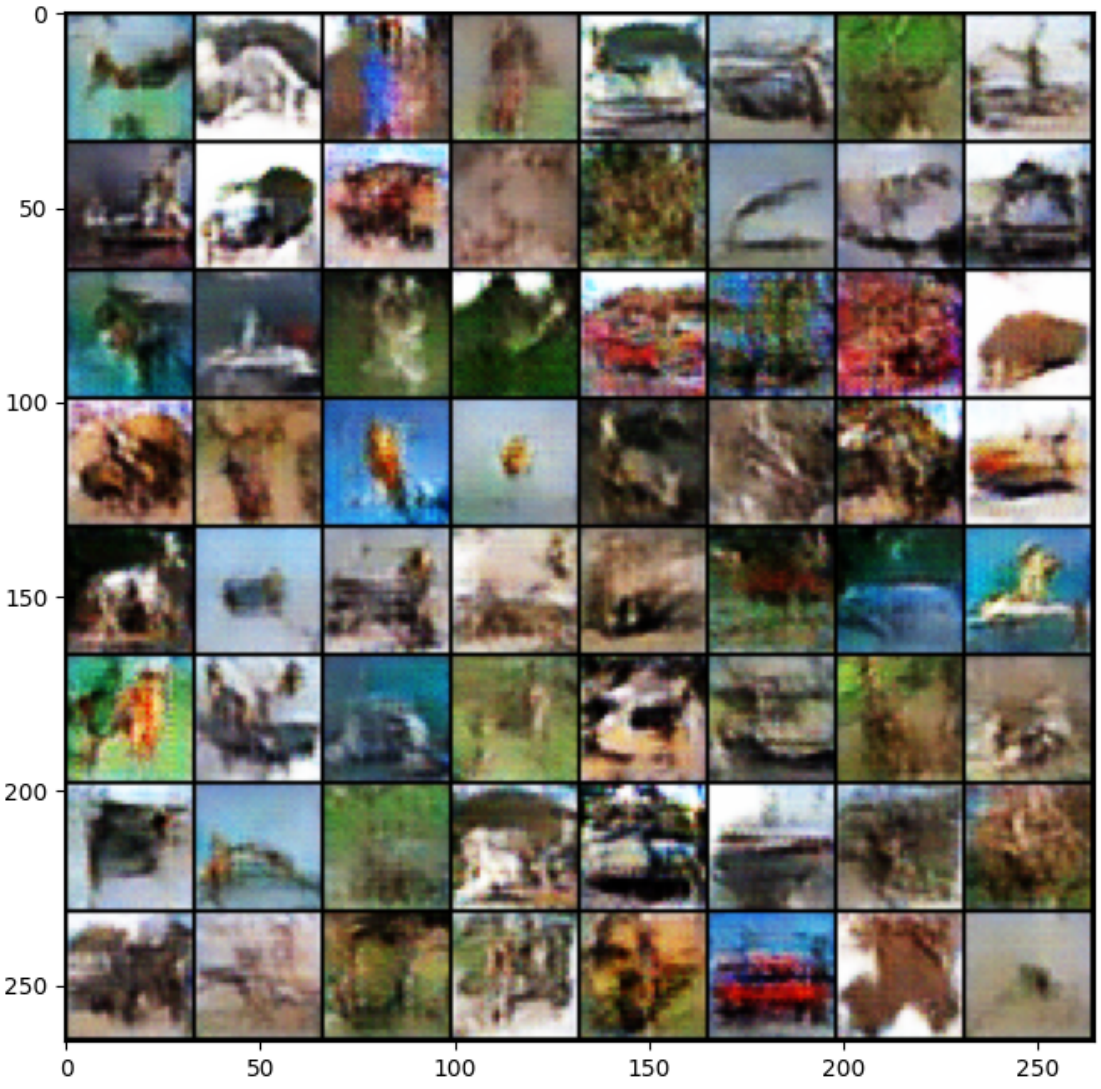

```
Iteration 6700/9750: dis loss = 0.6024, gen loss = 1.6173
Iteration 6800/9750: dis loss = 0.5097, gen loss = 2.8316
Iteration 6900/9750: dis loss = 0.5809, gen loss = 1.9387
Iteration 7000/9750: dis loss = 0.7697, gen loss = 2.5412
Iteration 7100/9750: dis loss = 1.0708, gen loss = 1.6810
Iteration 7200/9750: dis loss = 1.0099, gen loss = 3.1245
Iteration 7300/9750: dis loss = 0.4574, gen loss = 2.2388
Iteration 7400/9750: dis loss = 0.6766, gen loss = 2.3349
```

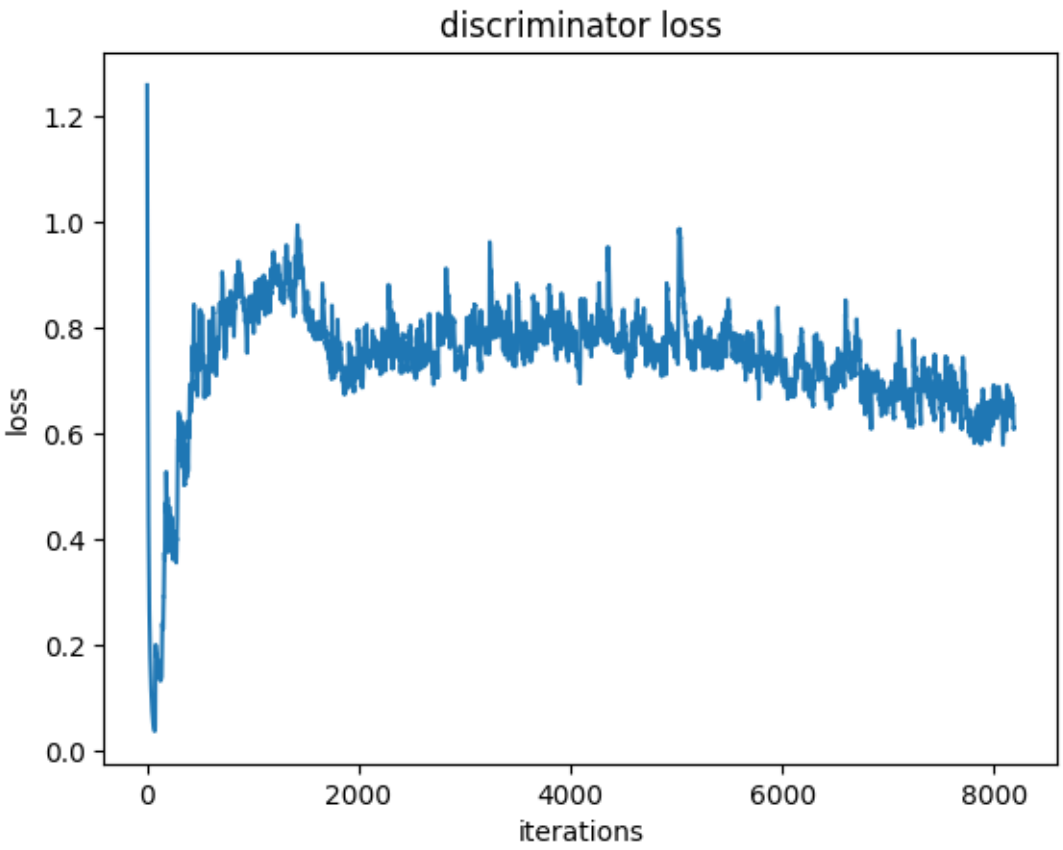

0
50
100
150
200
250
0
50
100
150
200
250

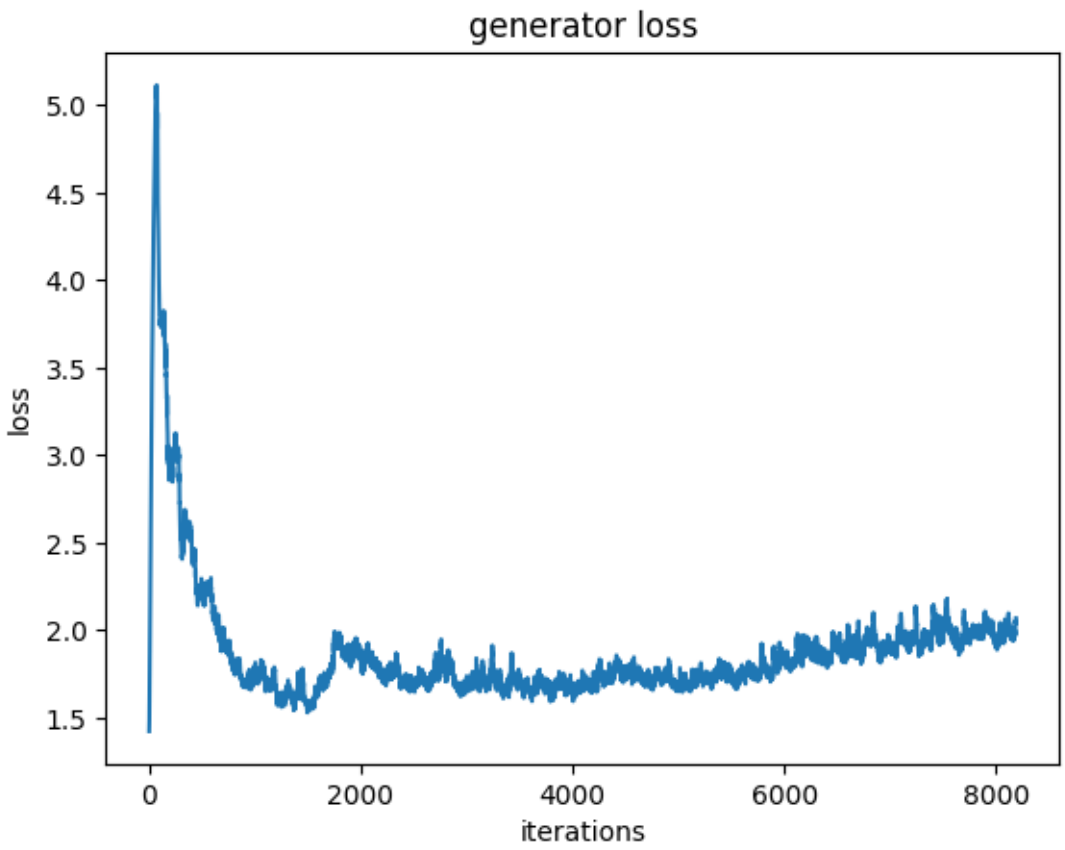

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
1000
2000
3000
4000
5000
6000
7000



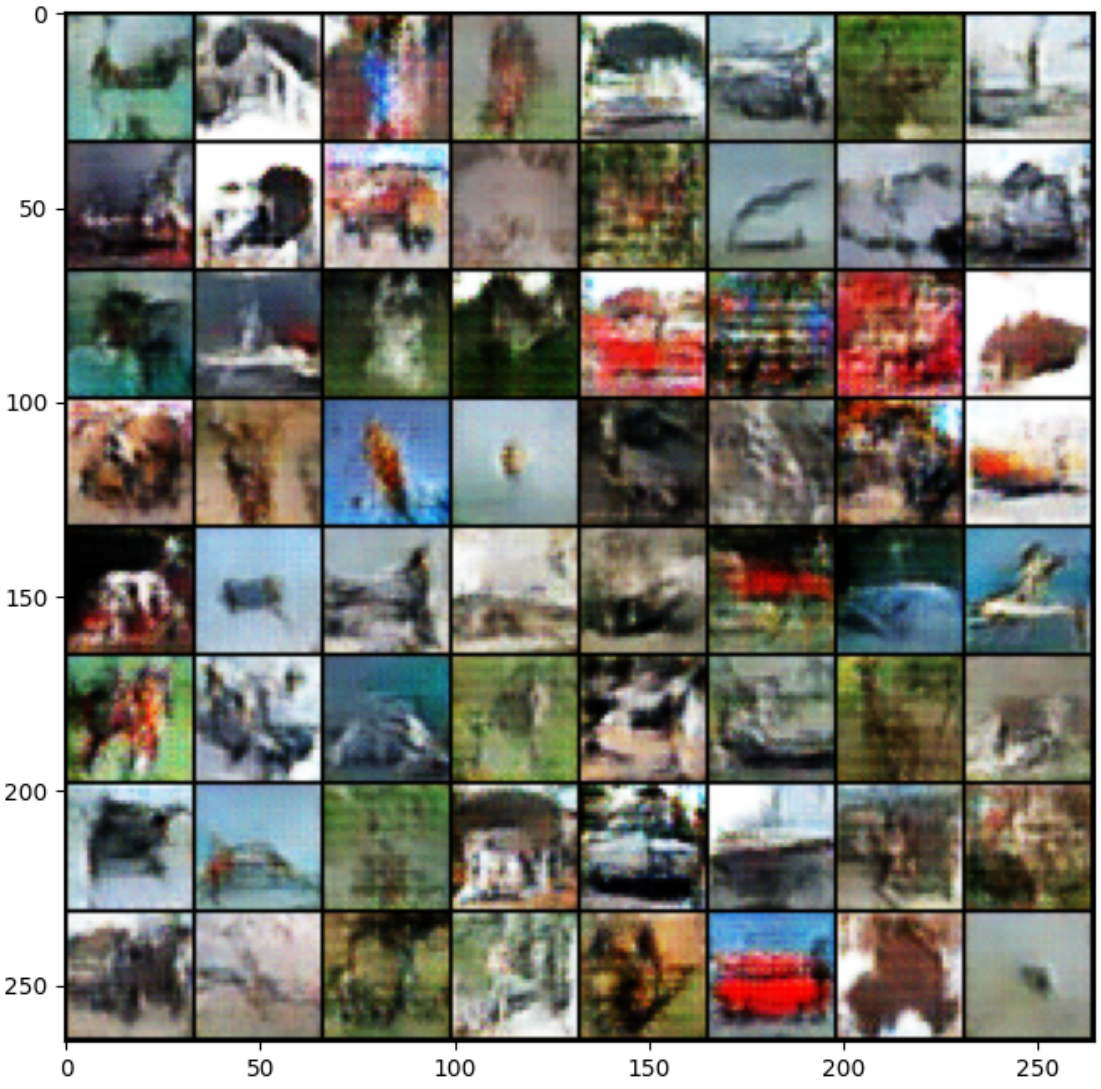

```
Iteration 7500/9750: dis loss = 0.6165, gen loss = 1.8778
Iteration 7600/9750: dis loss = 0.5171, gen loss = 2.8258
Iteration 7700/9750: dis loss = 2.2573, gen loss = 1.6881
Iteration 7800/9750: dis loss = 0.7315, gen loss = 1.1714
Iteration 7900/9750: dis loss = 0.6870, gen loss = 2.1258
Iteration 8000/9750: dis loss = 0.7996, gen loss = 2.8092
Iteration 8100/9750: dis loss = 0.6085, gen loss = 1.8672
```

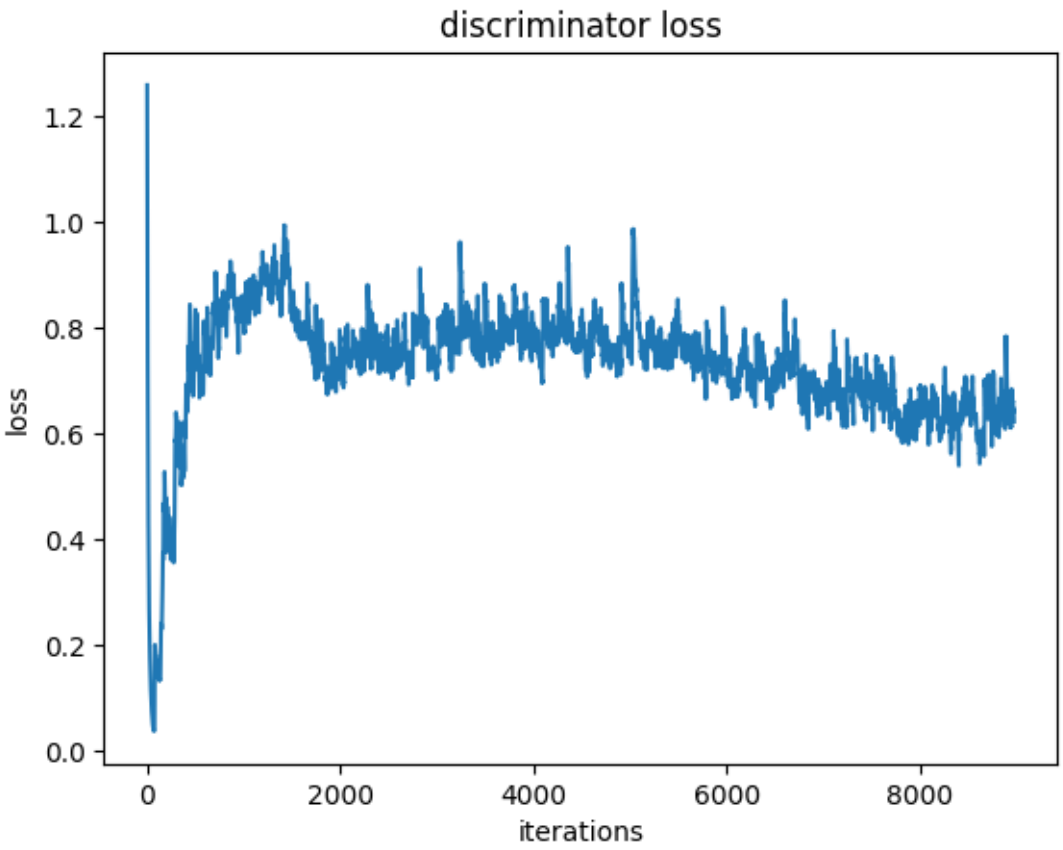

0
50
100
150
200
250
0
50
100
150
200
250

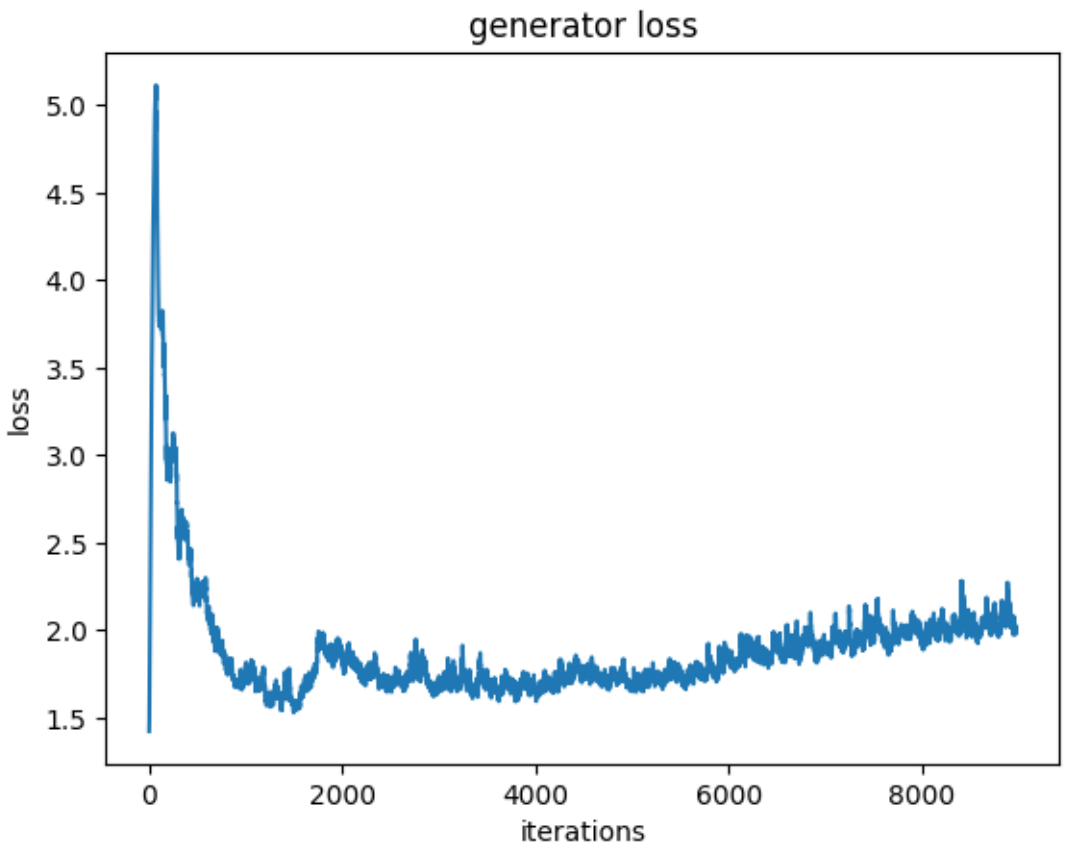

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
2000
4000
6000
8000



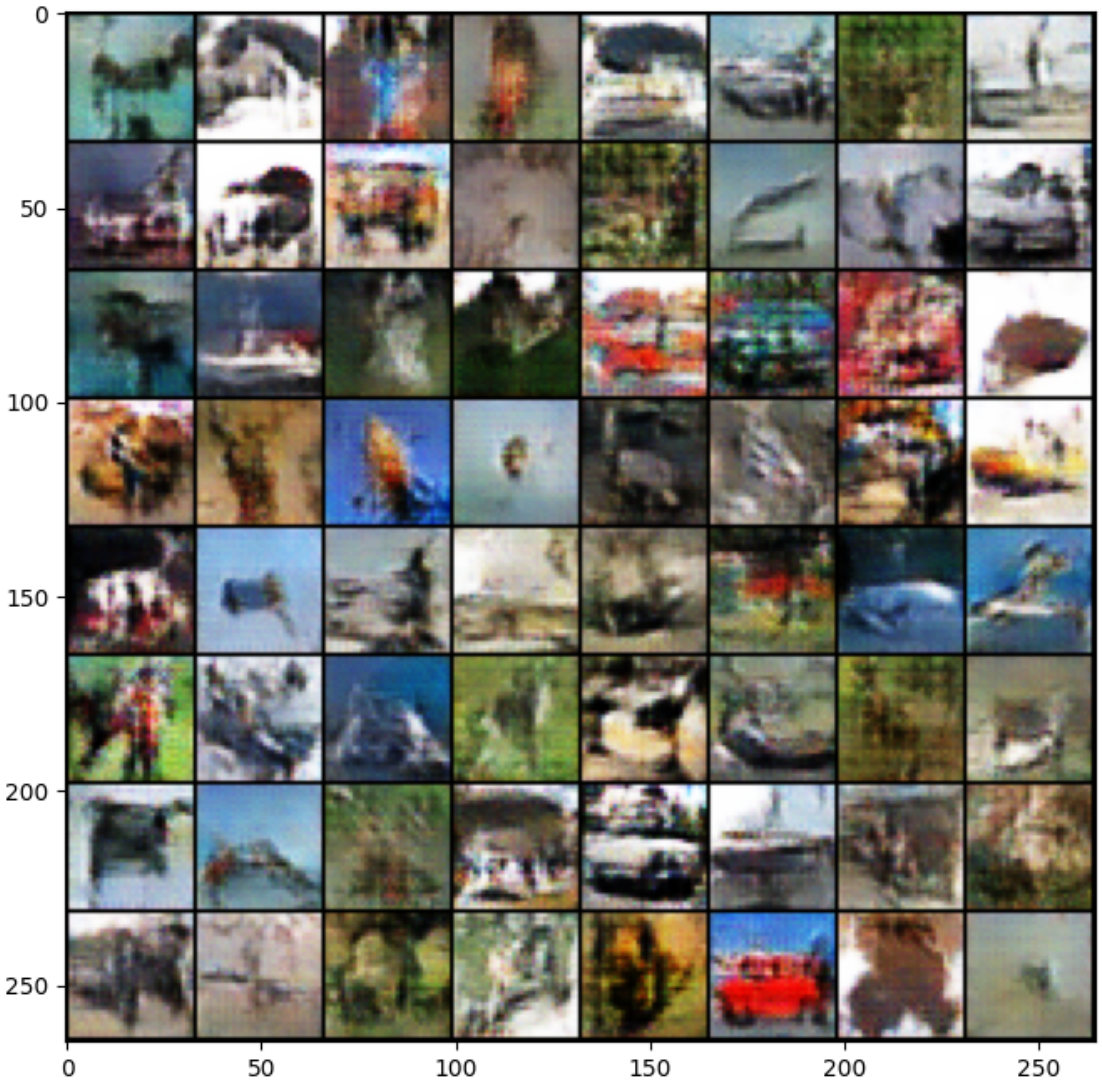

```
Iteration 8200/9750: dis loss = 0.5301, gen loss = 2.3957
Iteration 8300/9750: dis loss = 0.6703, gen loss = 1.0981
Iteration 8400/9750: dis loss = 1.7711, gen loss = 3.3536
Iteration 8500/9750: dis loss = 0.5589, gen loss = 1.8404
Iteration 8600/9750: dis loss = 0.4608, gen loss = 2.3146
Iteration 8700/9750: dis loss = 0.5318, gen loss = 1.8552
Iteration 8800/9750: dis loss = 0.5819, gen loss = 2.4937
Iteration 8900/9750: dis loss = 0.5663, gen loss = 1.7293
```

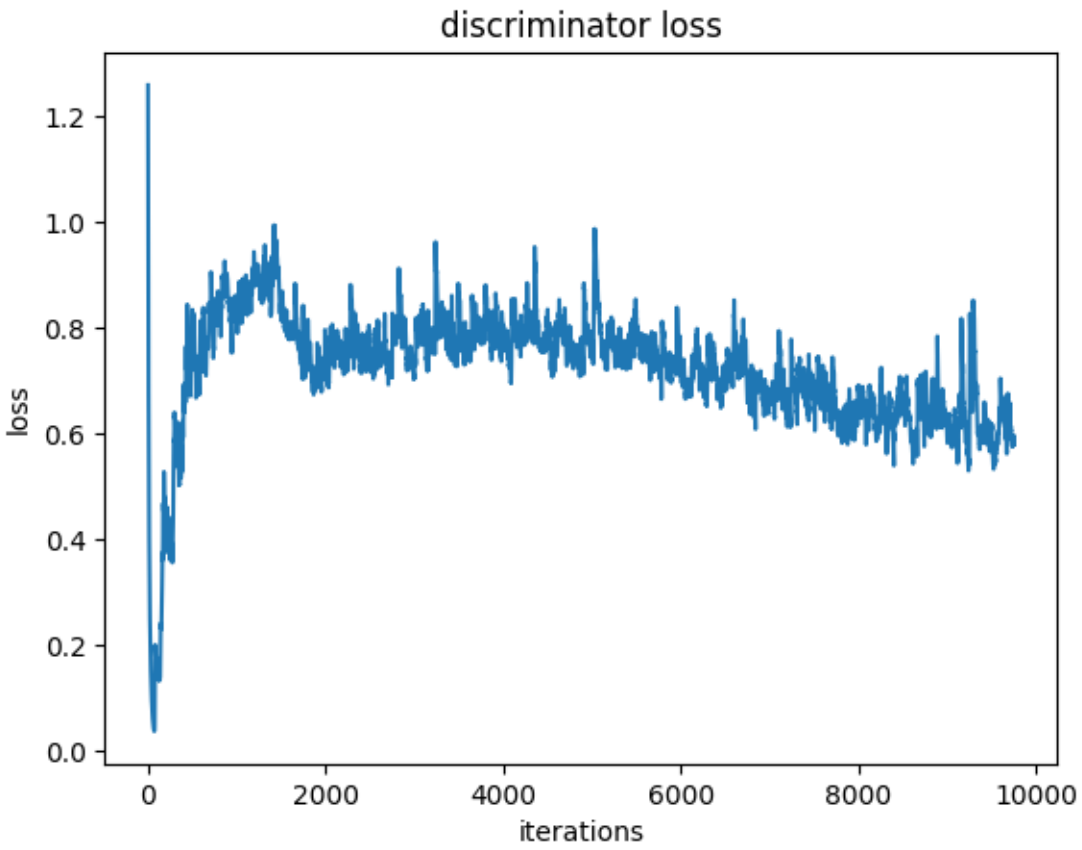

0
50
100
150
200
250
0
50
100
150
200
250

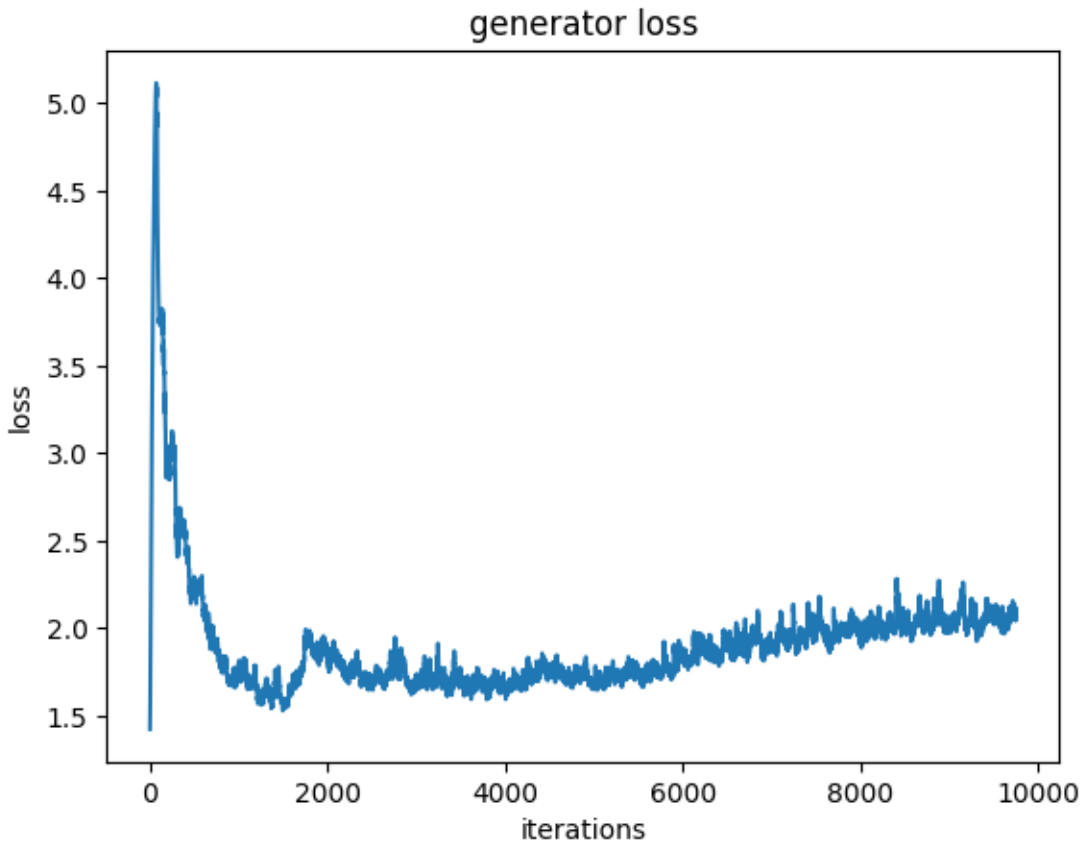

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
2000
4000
6000
8000




```
Iteration 9000/9750: dis loss = 0.4863, gen loss = 1.8045
Iteration 9100/9750: dis loss = 0.5259, gen loss = 2.2634
Iteration 9200/9750: dis loss = 0.5837, gen loss = 2.3736
Iteration 9300/9750: dis loss = 0.5492, gen loss = 2.0796
Iteration 9400/9750: dis loss = 0.6802, gen loss = 2.3887
Iteration 9500/9750: dis loss = 0.6260, gen loss = 1.6848
Iteration 9600/9750: dis loss = 0.5878, gen loss = 1.7177
Iteration 9700/9750: dis loss = 0.9385, gen loss = 1.9762
```


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


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



… 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: The batch normalization layer would normalize the fake and real batches in a way that the discriminator would not be able to distinguish the two distritbutions, hence failing to learn the proper distribution of the real batches.

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)

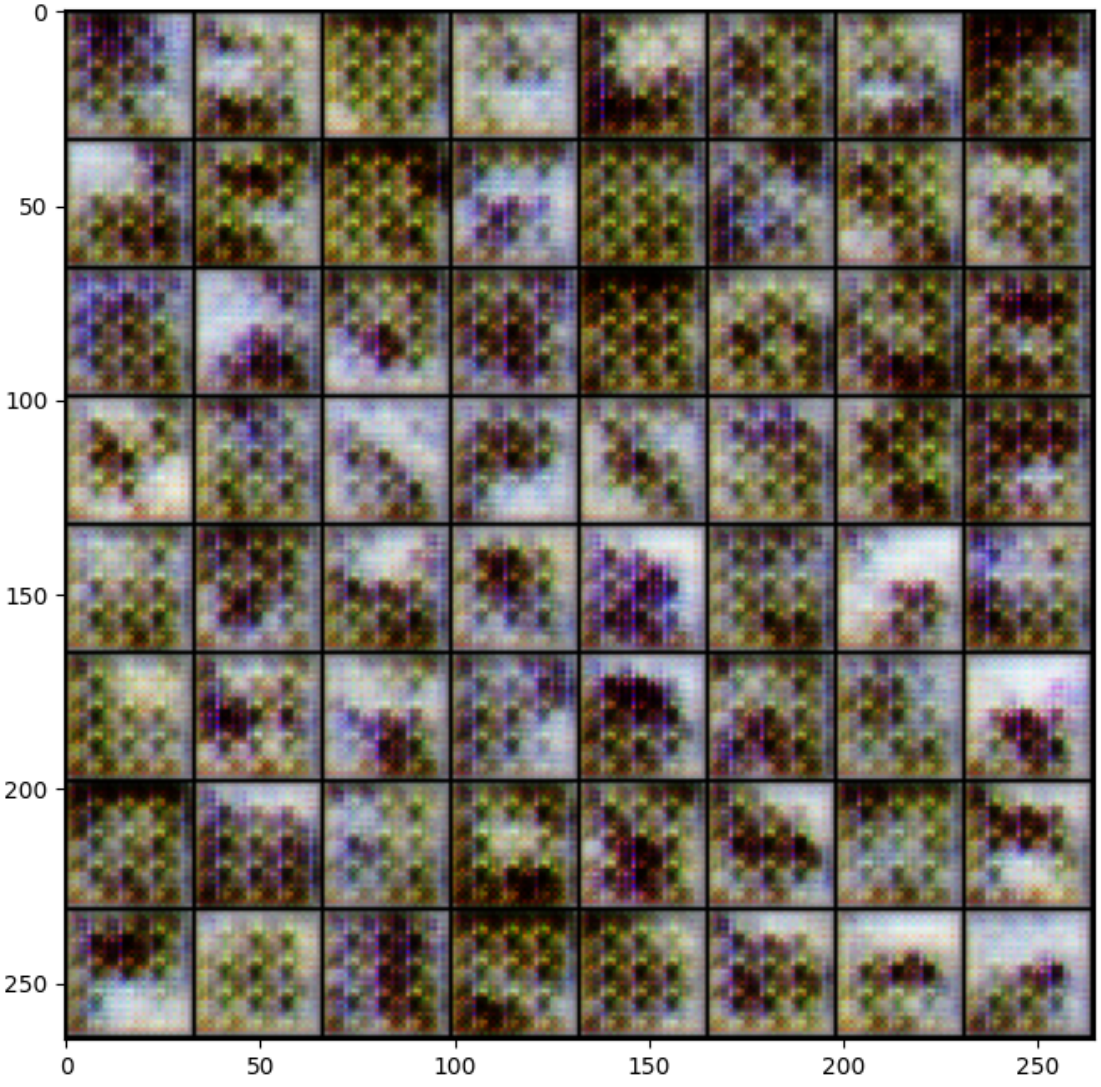
[16]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        ␣
↪###############################################################################
        # Prob 2-3:                                                             ␣
↪       #
        # adding spectral norm to the discriminator                             ␣
↪       #
        ␣
↪###############################################################################
        self.downsample = nn.Sequential(
            nn.utils.spectral_norm(nn.Conv2d(in_channels=3, out_channels=32,␣
↪kernel_size=4, stride=2, padding=1, bias=True)),
            nn.LeakyReLU(),
            nn.utils.spectral_norm(nn.Conv2d(in_channels=32, out_channels=64,␣
↪kernel_size=4, stride=2, padding=1, bias=True)),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.utils.spectral_norm(nn.Conv2d(in_channels=64, out_channels=128,␣
↪kernel_size=4, stride=2, padding=1, bias=True)),
            nn.BatchNorm2d(128),
```

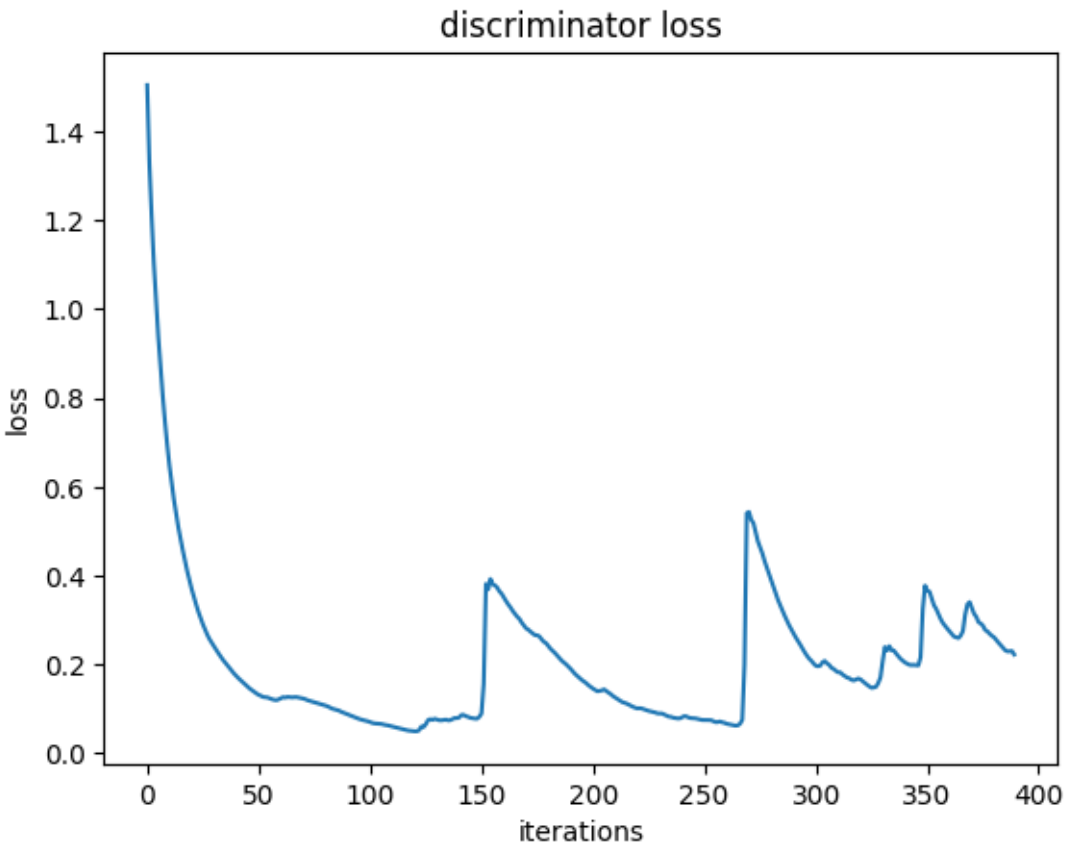
```
            nn.LeakyReLU(),
        )
        ␣
↪#############################################################################
        # END OF YOUR CODE                                                    ␣
↪       #
        ␣
↪#############################################################################
        self.fc = nn.Linear(4 * 4 * 128, 1)
    def forward(self, input):
        downsampled_image = self.downsample(input)
        reshaped_for_fc = downsampled_image.reshape((-1, 4 * 4 * 128))
        classification_probs = self.fc(reshaped_for_fc)
        return classification_probs
```

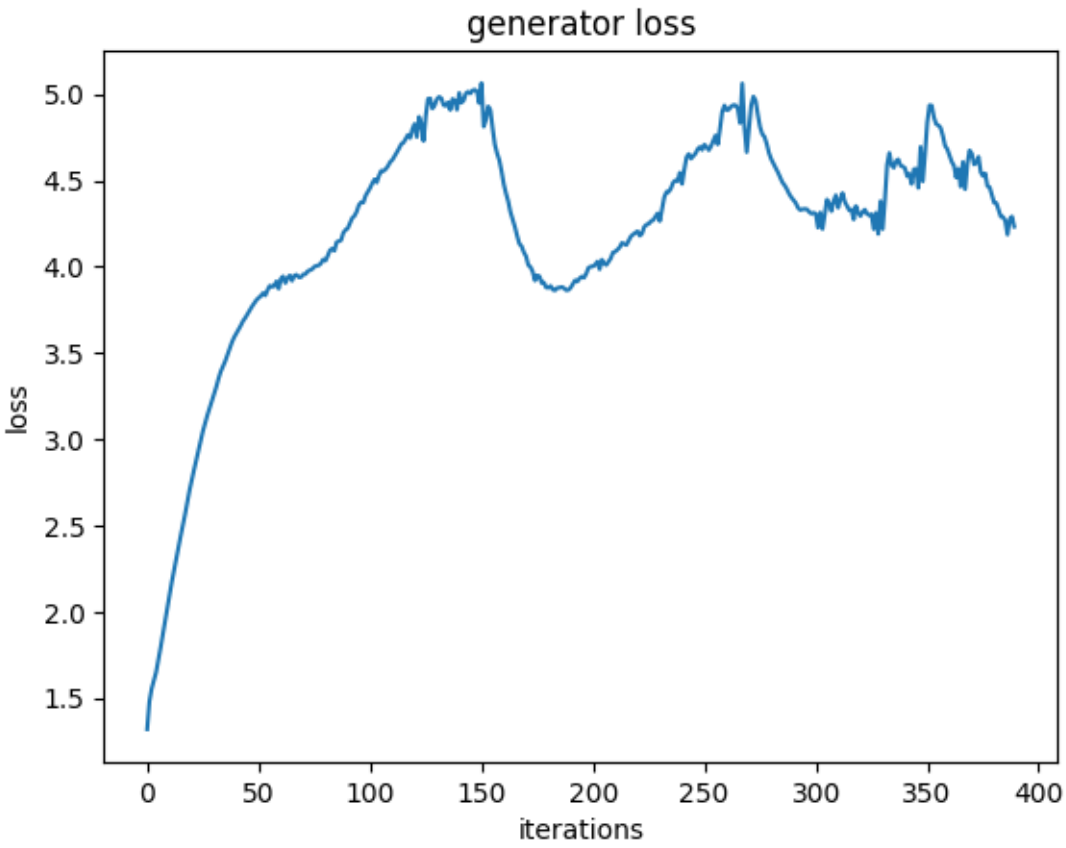
After adding the spectral norm to the discriminator, redo the training block below to see the effects.

[17]:
```
set_seed(42)

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

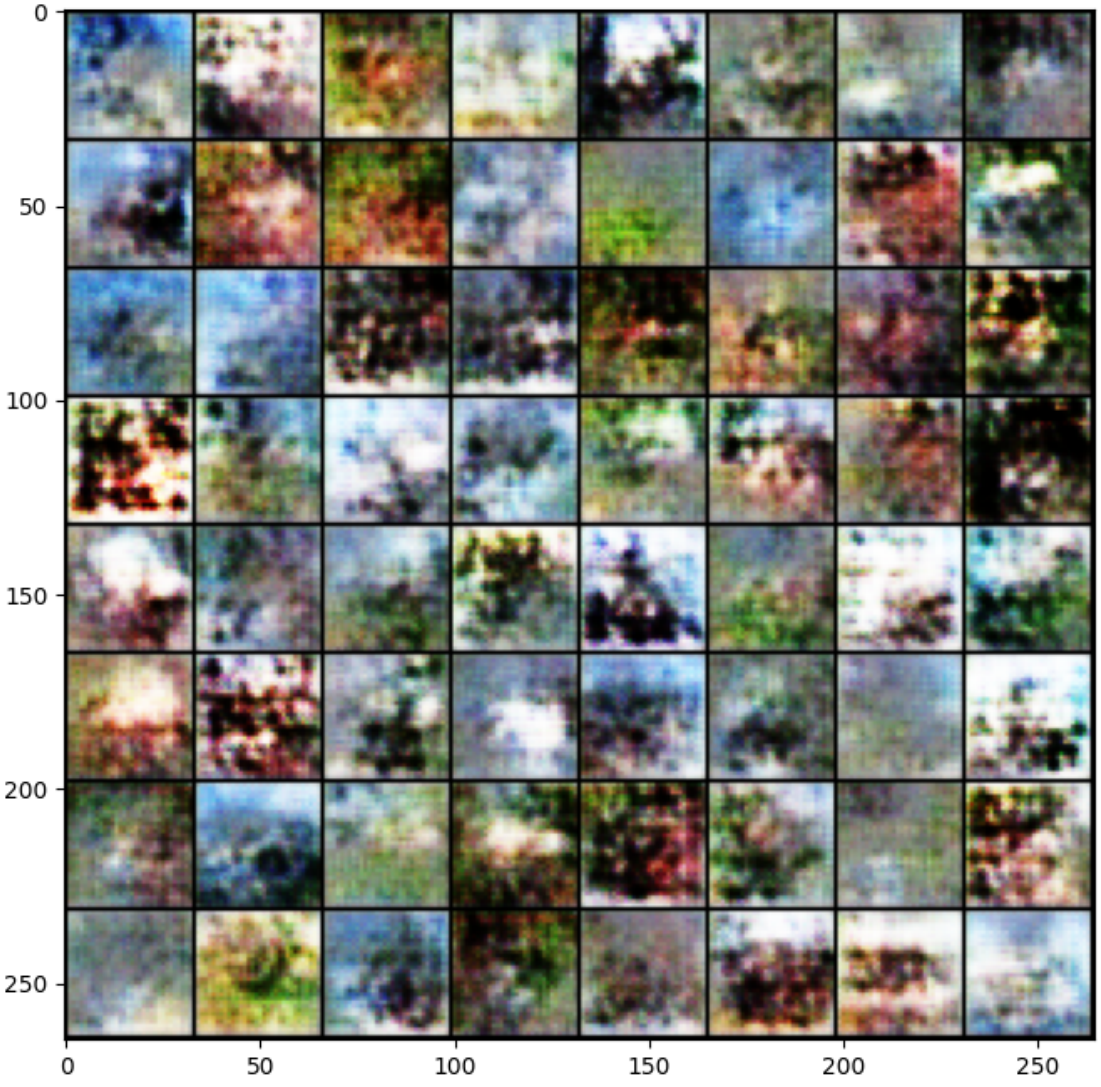
```
Start training …
Iteration 100/9750: dis loss = 0.0458, gen loss = 4.8489
Iteration 200/9750: dis loss = 0.0778, gen loss = 4.1109
Iteration 300/9750: dis loss = 0.0840, gen loss = 4.3930
```

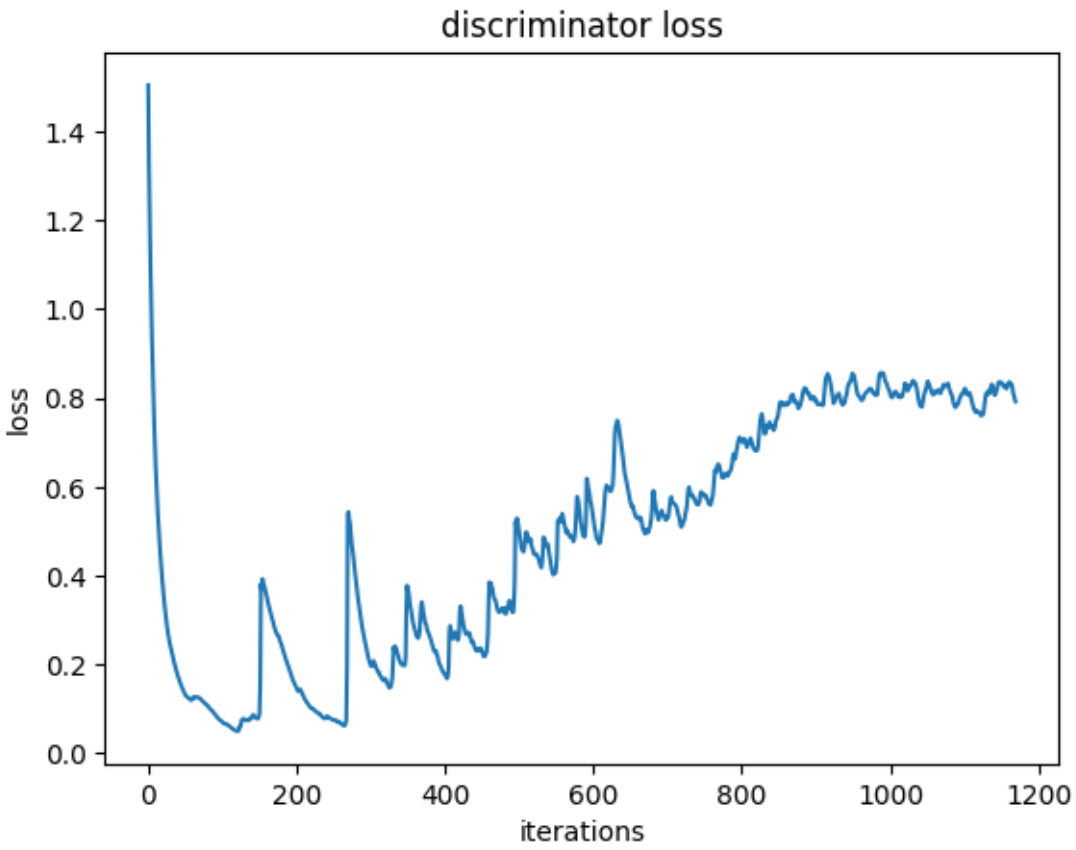

0
50
100
150
200
250
0
50
100
150
200
250

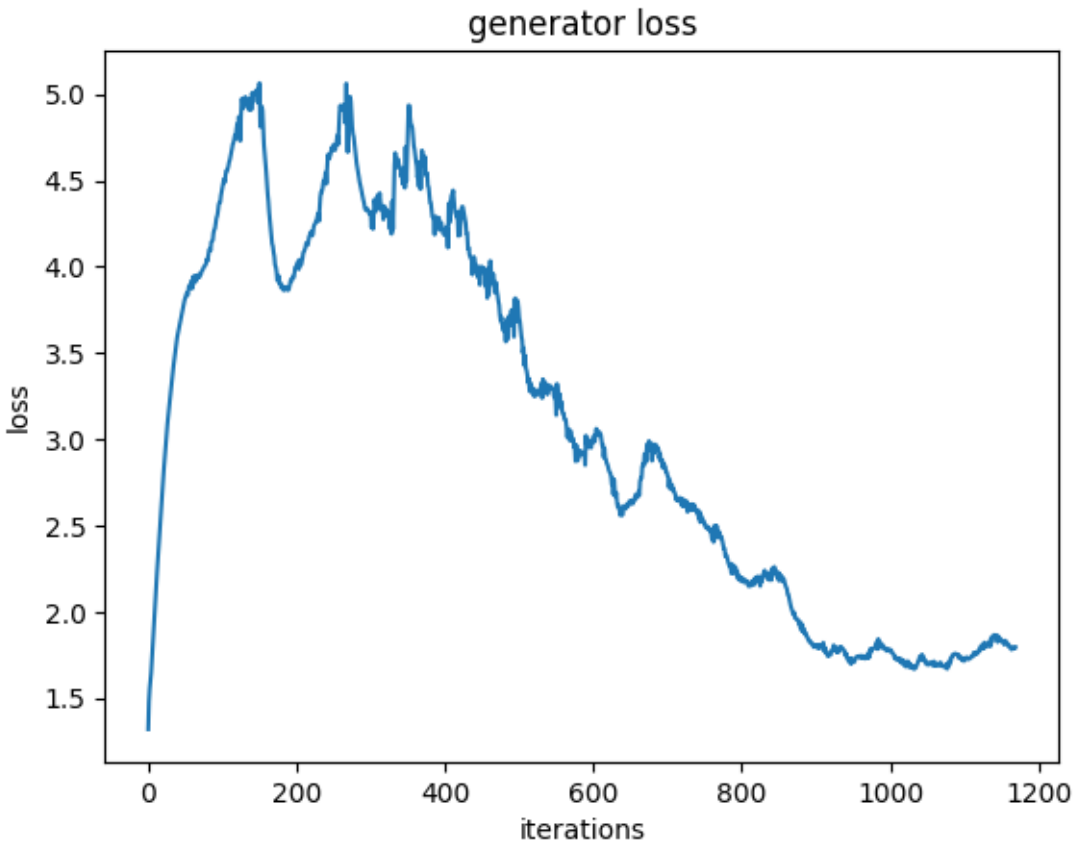

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
50
100
150
200
250
300
350
400



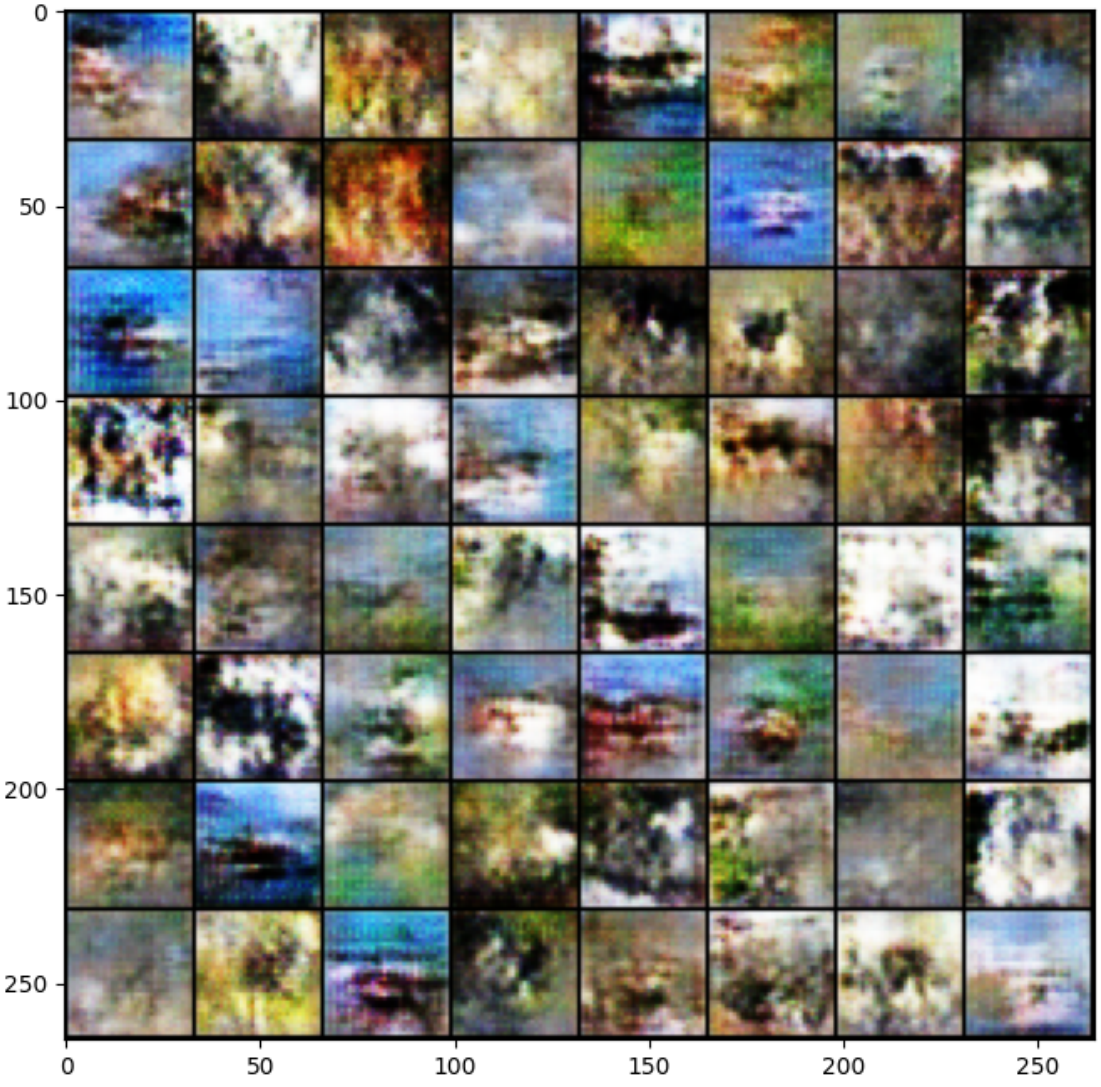

```
Iteration 400/9750: dis loss = 0.1386, gen loss = 3.8525
Iteration 500/9750: dis loss = 0.2707, gen loss = 2.9477
Iteration 600/9750: dis loss = 0.3321, gen loss = 3.0173
Iteration 700/9750: dis loss = 0.6665, gen loss = 2.8068
Iteration 800/9750: dis loss = 0.6666, gen loss = 2.4712
Iteration 900/9750: dis loss = 0.7357, gen loss = 1.8052
Iteration 1000/9750: dis loss = 0.7758, gen loss = 1.9062
Iteration 1100/9750: dis loss = 0.8681, gen loss = 2.0308
```

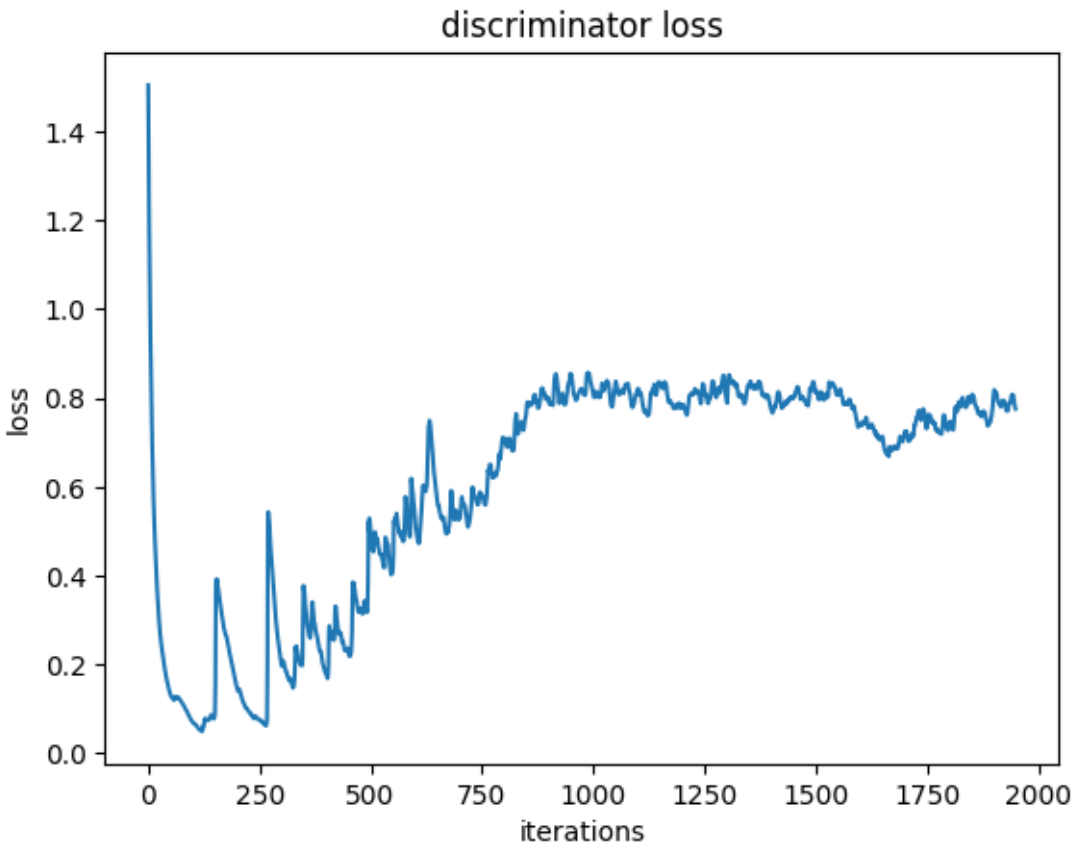

0
50
100
150
200
250
0
50
100
150
200
250

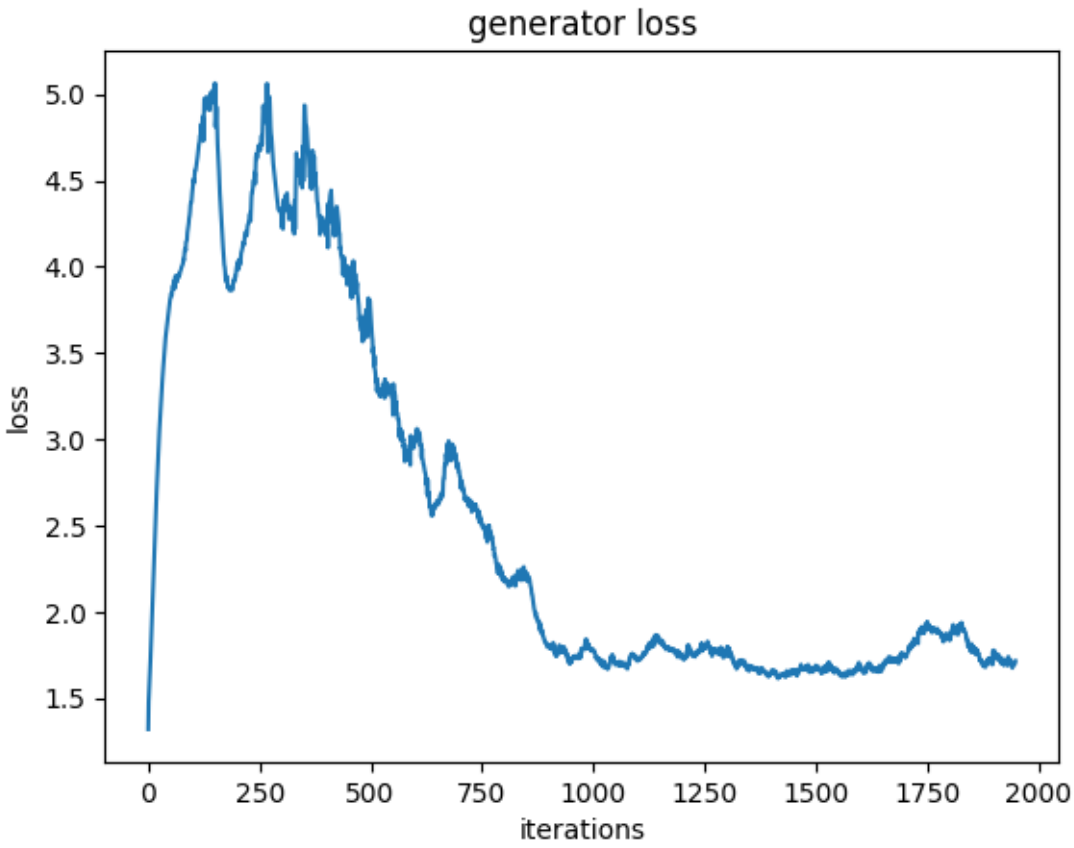

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
200
400
600
800
1000
1200




```
Iteration 1200/9750: dis loss = 0.8210, gen loss = 1.3849
Iteration 1300/9750: dis loss = 0.6909, gen loss = 1.6911
Iteration 1400/9750: dis loss = 0.6906, gen loss = 1.8569
Iteration 1500/9750: dis loss = 0.7242, gen loss = 1.7774
Iteration 1600/9750: dis loss = 0.6848, gen loss = 1.6278
Iteration 1700/9750: dis loss = 0.7858, gen loss = 1.4336
Iteration 1800/9750: dis loss = 0.7834, gen loss = 2.0044
Iteration 1900/9750: dis loss = 1.1328, gen loss = 0.8730
```

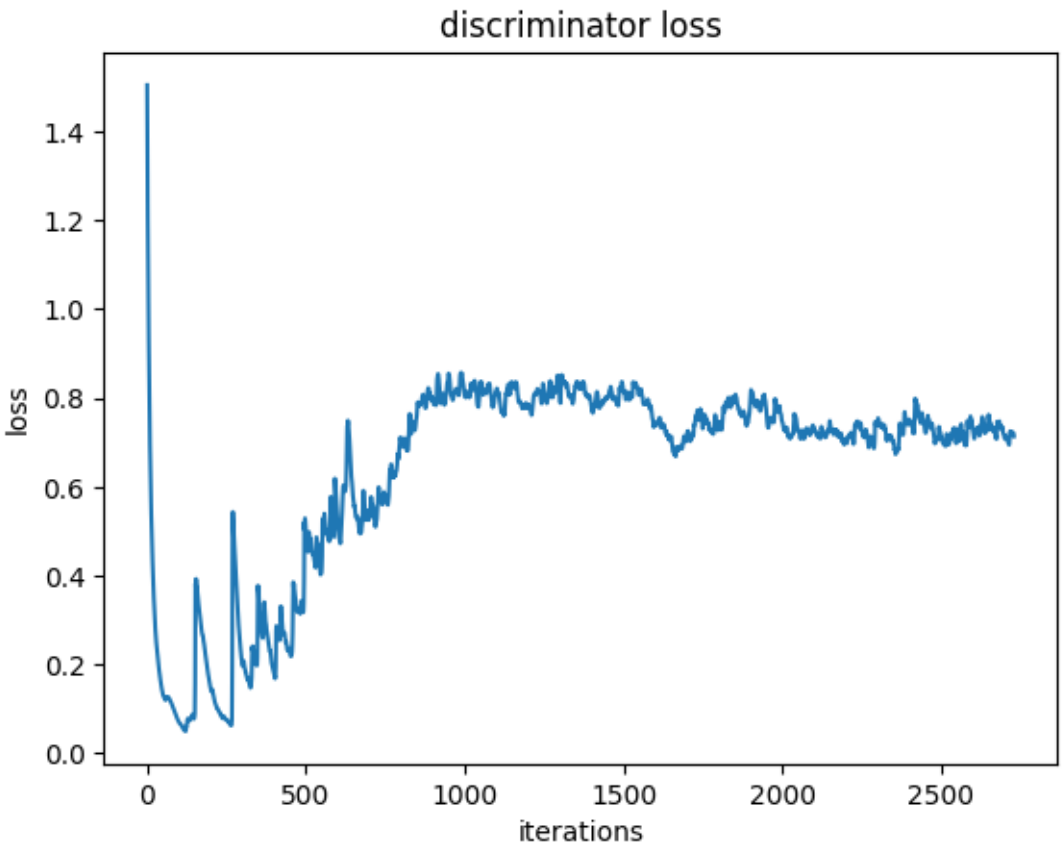

0
50
100
150
200
250
0
50
100
150
200
250

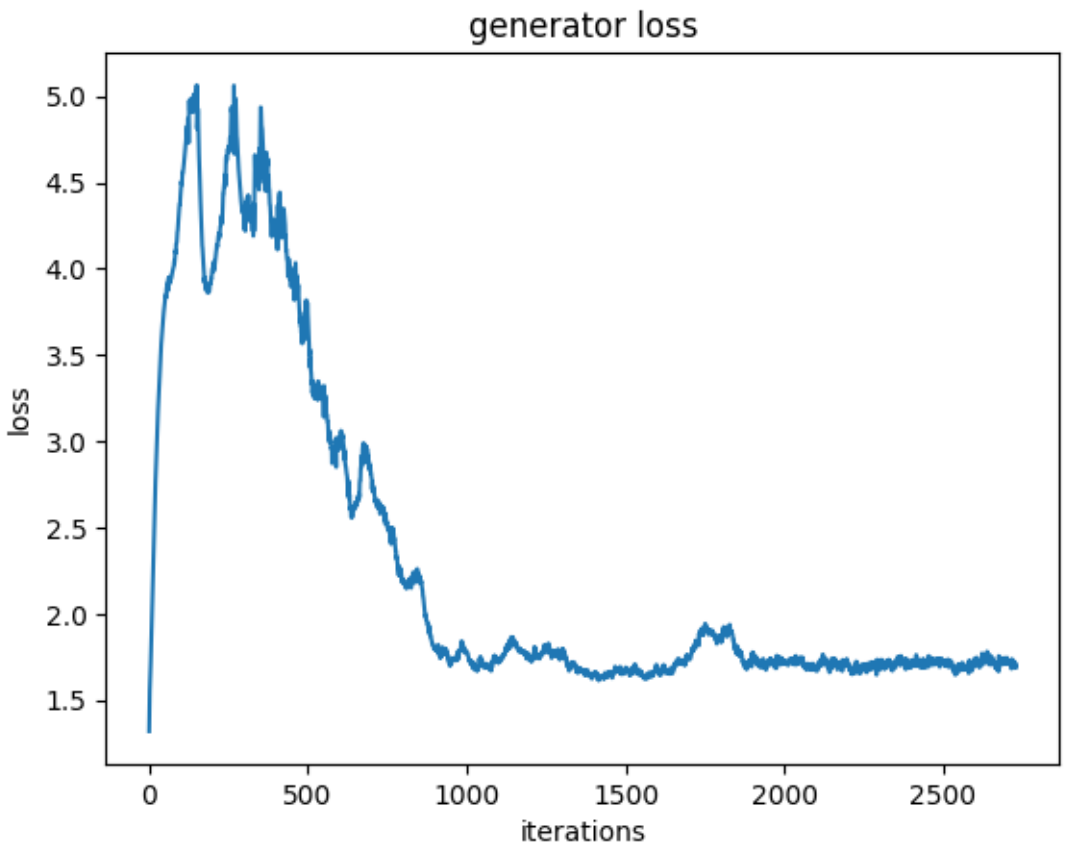

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
250
500
750
1000
1250
1500
1750
2000




```
Iteration 2000/9750: dis loss = 0.7076, gen loss = 1.9708
Iteration 2100/9750: dis loss = 0.7083, gen loss = 1.5735
Iteration 2200/9750: dis loss = 0.6701, gen loss = 1.7246
Iteration 2300/9750: dis loss = 0.7343, gen loss = 2.3249
Iteration 2400/9750: dis loss = 0.6627, gen loss = 1.6278
Iteration 2500/9750: dis loss = 0.7478, gen loss = 1.8297
Iteration 2600/9750: dis loss = 0.6652, gen loss = 2.2780
Iteration 2700/9750: dis loss = 0.6731, gen loss = 1.2722
```

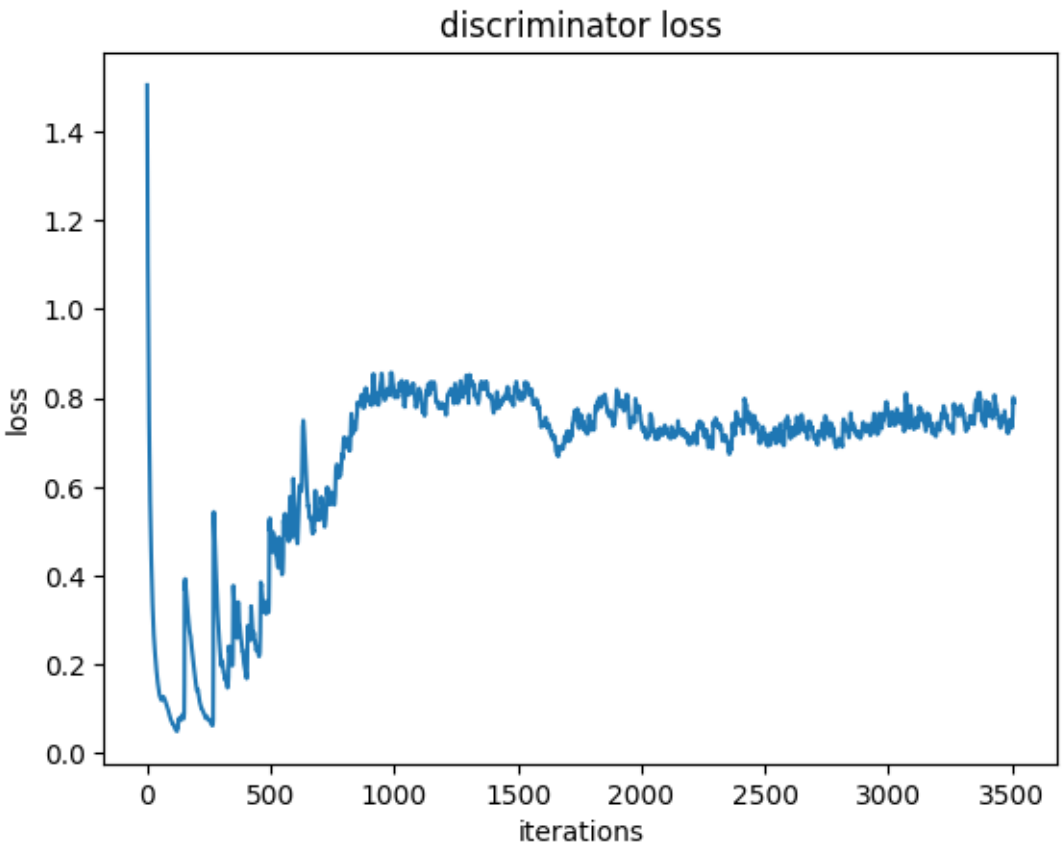

0
50
100
150
200
250
0
50
100
150
200
250

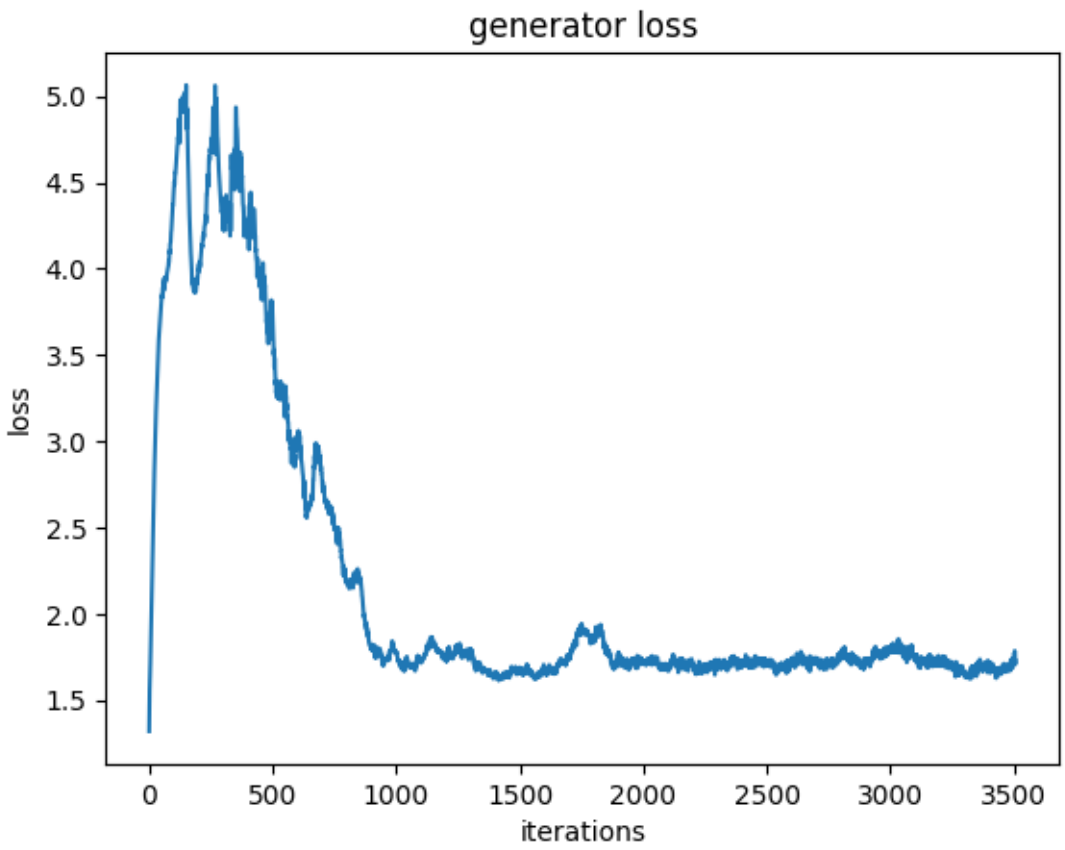

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
500
1000
1500
2000
2500




```
Iteration 2800/9750: dis loss = 0.7852, gen loss = 2.4948
Iteration 2900/9750: dis loss = 0.9161, gen loss = 2.2778
Iteration 3000/9750: dis loss = 0.7021, gen loss = 1.4866
Iteration 3100/9750: dis loss = 0.6041, gen loss = 1.6839
Iteration 3200/9750: dis loss = 0.6915, gen loss = 1.6592
Iteration 3300/9750: dis loss = 0.6417, gen loss = 1.4895
Iteration 3400/9750: dis loss = 0.7188, gen loss = 1.6998
Iteration 3500/9750: dis loss = 0.6386, gen loss = 1.7698
```

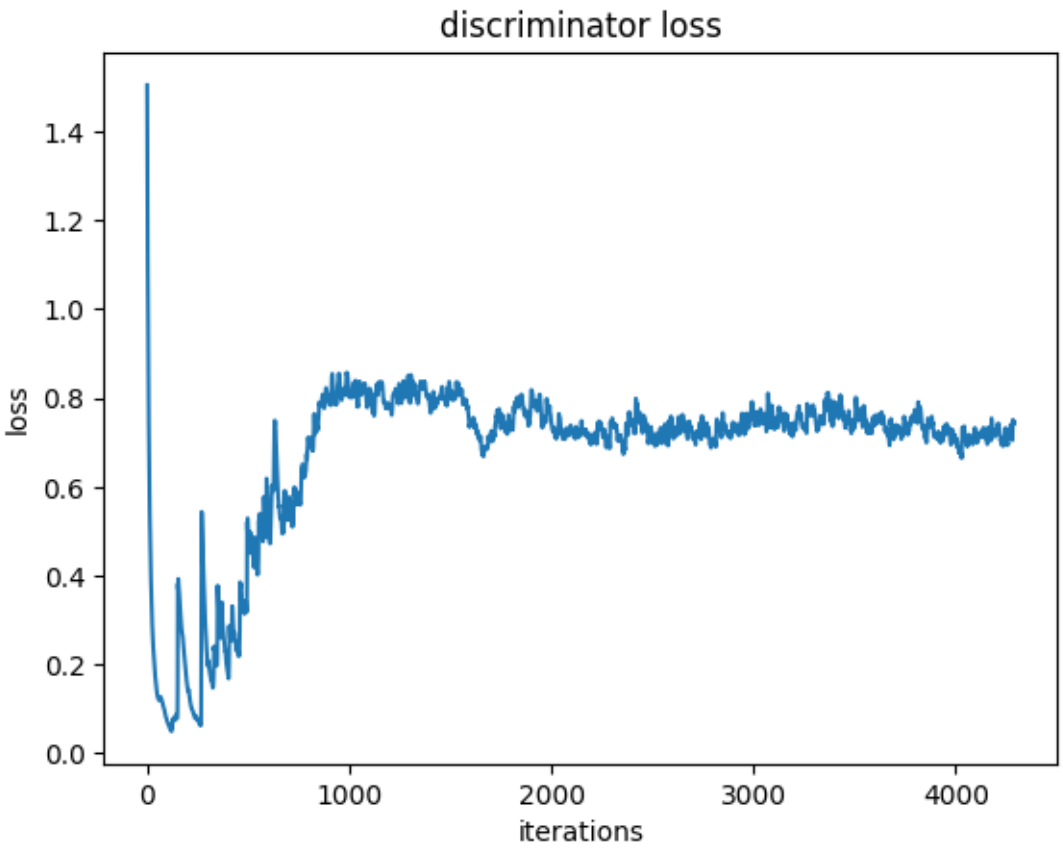

0
50
100
150
200
250
0
50
100
150
200
250

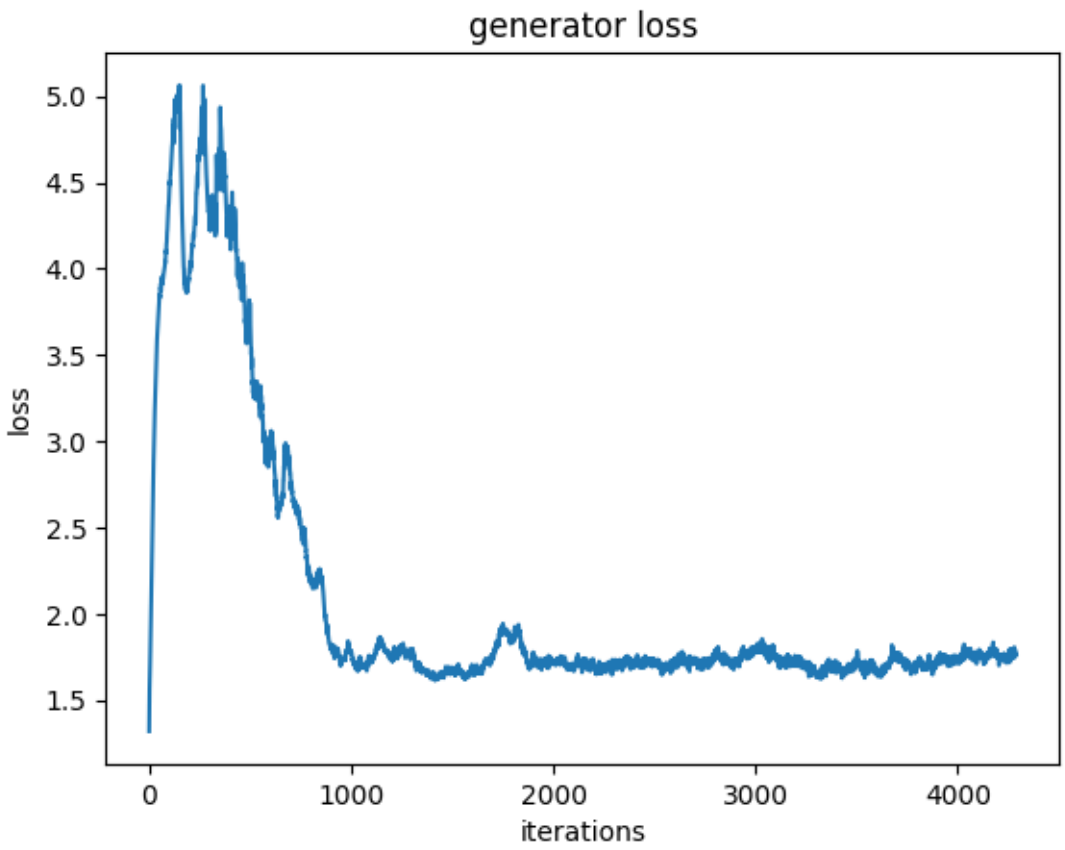

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
500
1000
1500
2000
2500
3000
3500




```
Iteration 3600/9750: dis loss = 0.7033, gen loss = 1.9161
Iteration 3700/9750: dis loss = 0.7140, gen loss = 1.4774
Iteration 3800/9750: dis loss = 0.7665, gen loss = 1.6551
Iteration 3900/9750: dis loss = 0.6374, gen loss = 1.7484
Iteration 4000/9750: dis loss = 0.5870, gen loss = 1.7360
Iteration 4100/9750: dis loss = 0.6994, gen loss = 1.1508
Iteration 4200/9750: dis loss = 0.6513, gen loss = 1.5698
```

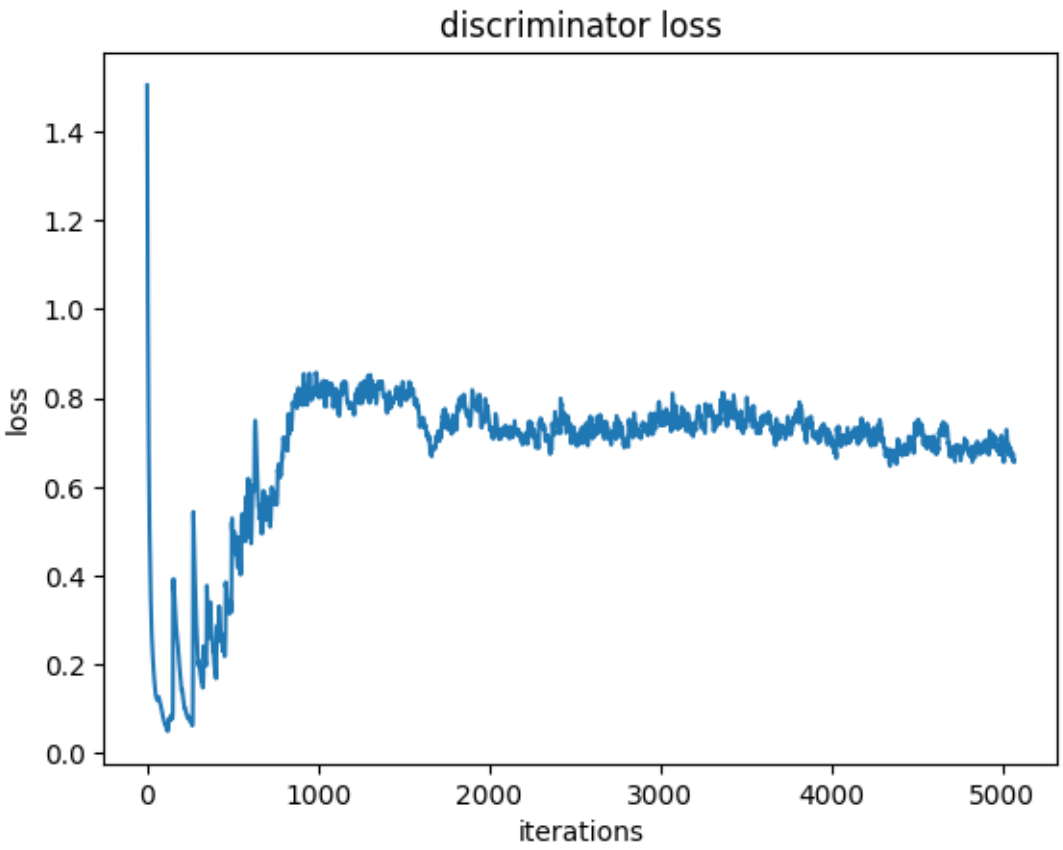

0
50
100
150
200
250
0
50
100
150
200
250

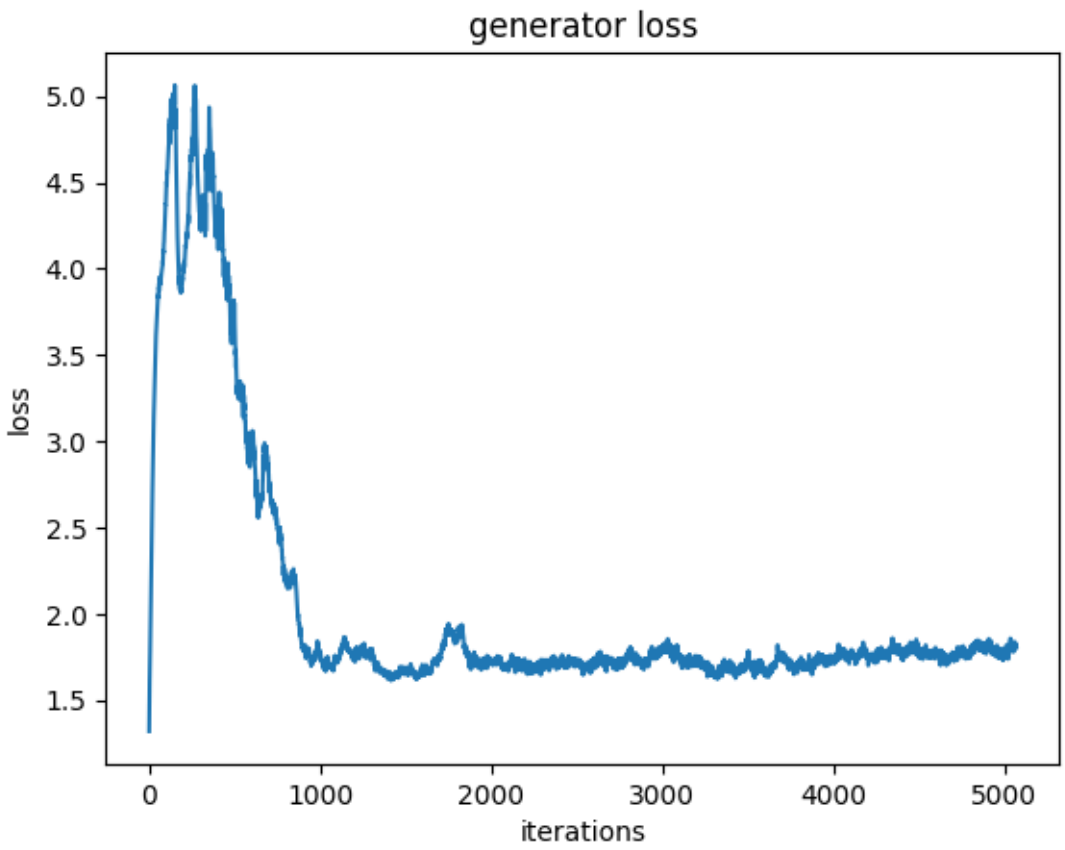

discriminator loss
loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
1000
2000
3000
4000
iterations



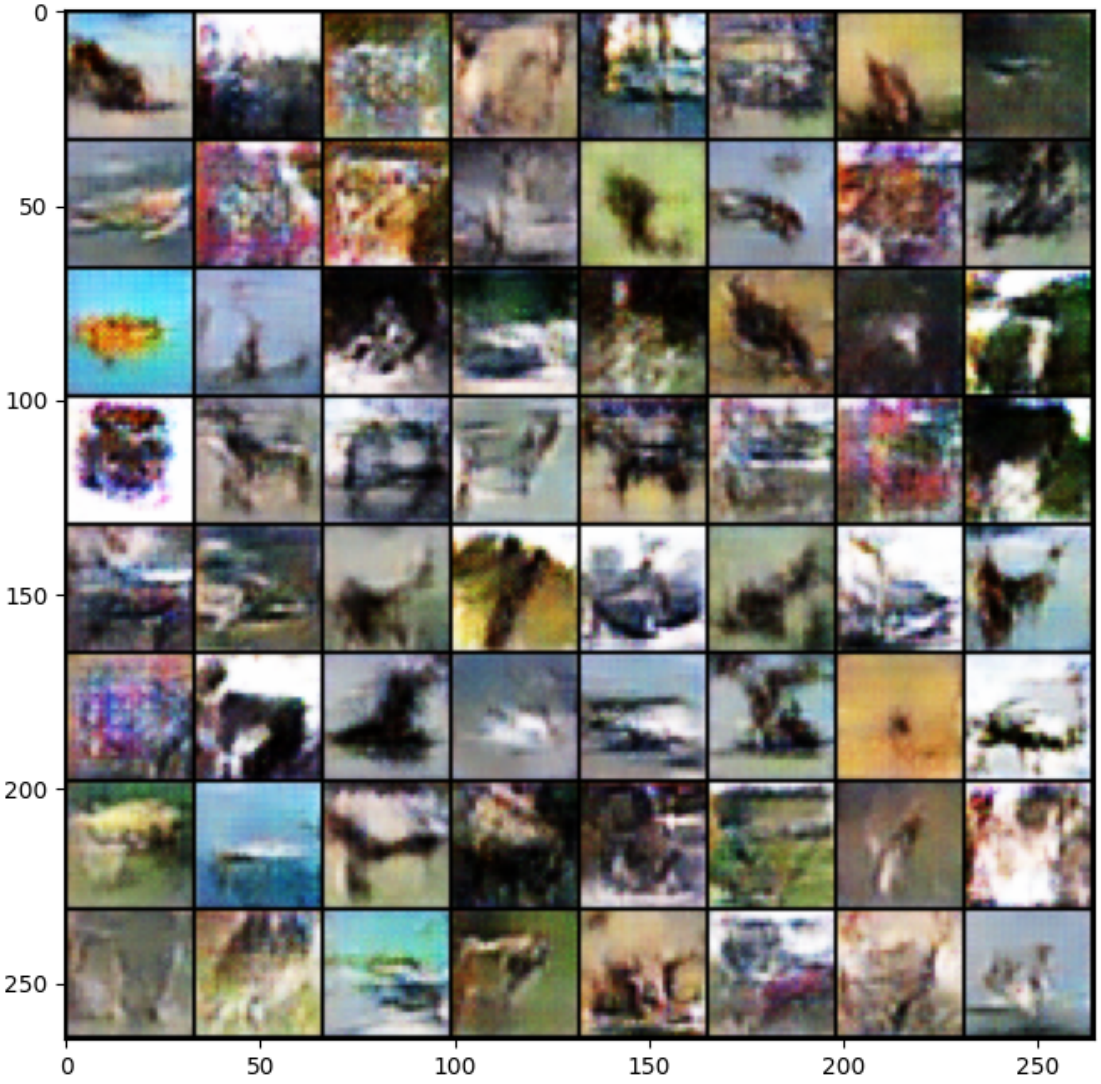

```
Iteration 4300/9750: dis loss = 0.6525, gen loss = 1.4837
Iteration 4400/9750: dis loss = 0.6256, gen loss = 1.8150
Iteration 4500/9750: dis loss = 0.7739, gen loss = 1.3214
Iteration 4600/9750: dis loss = 0.8081, gen loss = 2.1423
Iteration 4700/9750: dis loss = 0.6624, gen loss = 1.0490
Iteration 4800/9750: dis loss = 0.5978, gen loss = 1.8644
Iteration 4900/9750: dis loss = 0.6030, gen loss = 1.4212
Iteration 5000/9750: dis loss = 0.6135, gen loss = 1.6070
```

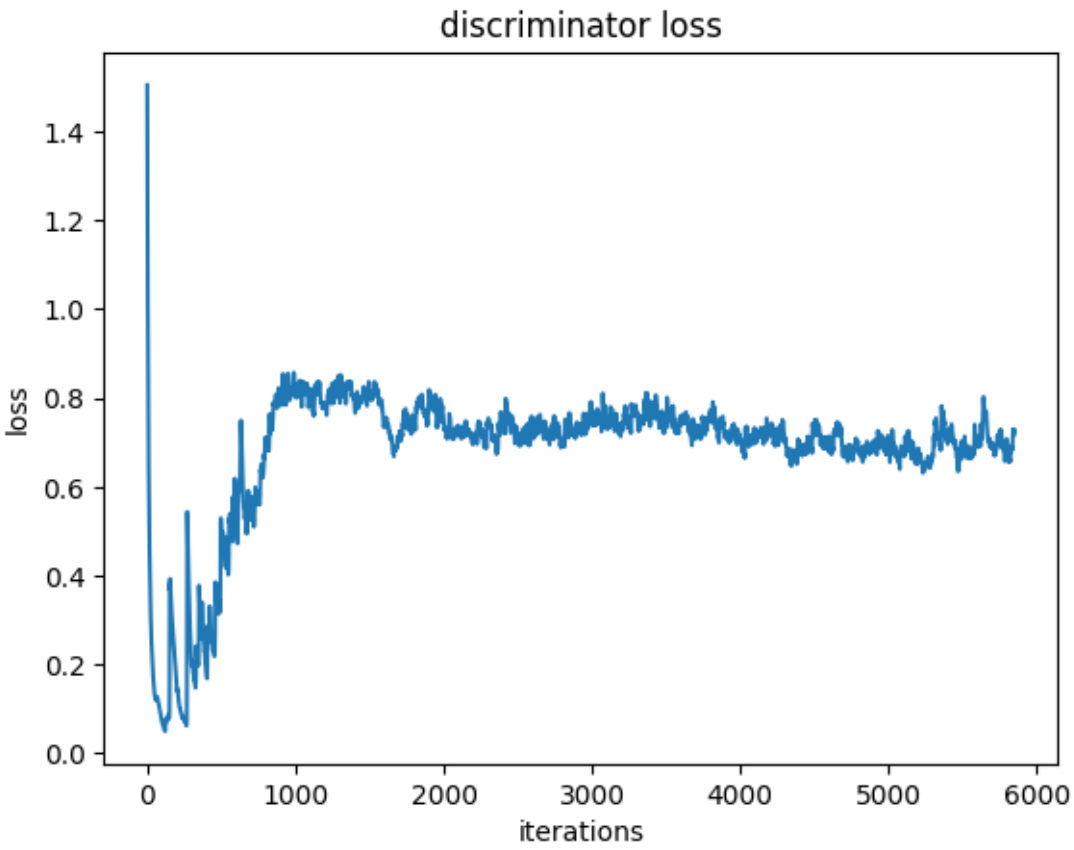

0
50
100
150
200
250
0
50
100
150
200
250

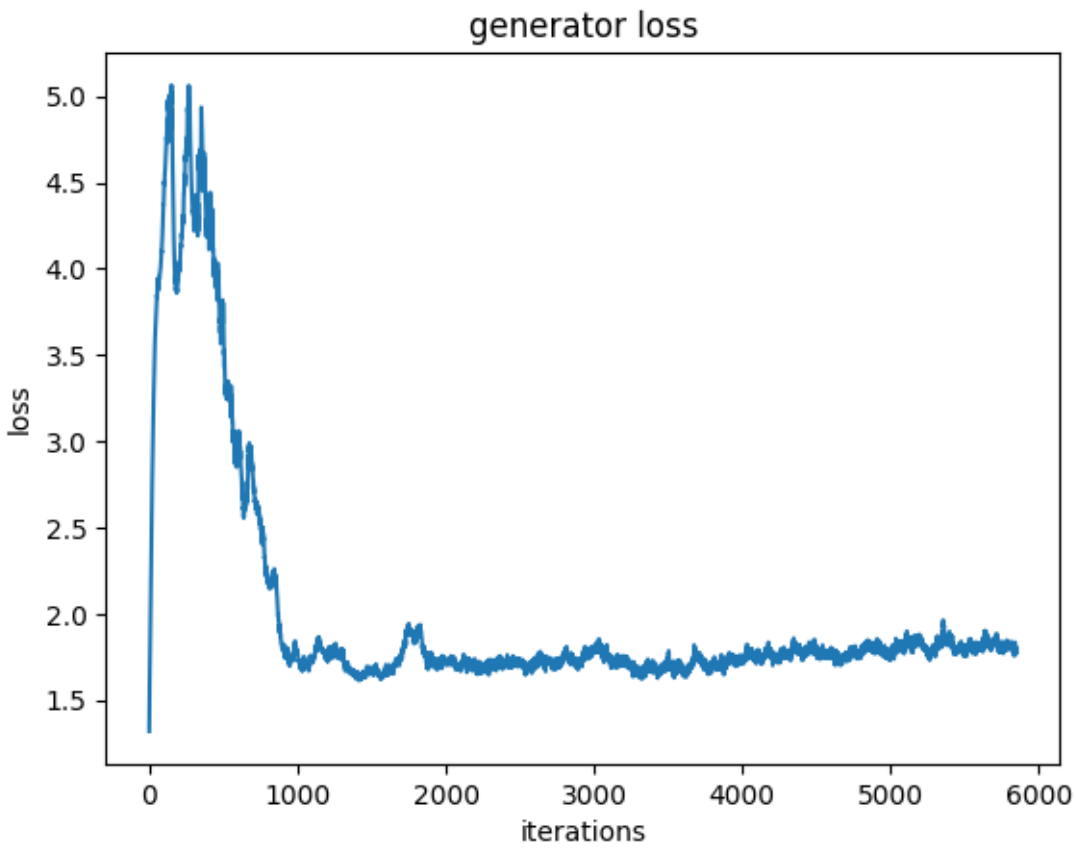

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
1000
2000
3000
4000
5000




```
Iteration 5100/9750: dis loss = 0.6182, gen loss = 2.5284
Iteration 5200/9750: dis loss = 0.6153, gen loss = 2.0696
Iteration 5300/9750: dis loss = 0.7061, gen loss = 1.6241
Iteration 5400/9750: dis loss = 0.6374, gen loss = 1.2614
Iteration 5500/9750: dis loss = 0.6989, gen loss = 1.3215
Iteration 5600/9750: dis loss = 0.6495, gen loss = 1.5497
Iteration 5700/9750: dis loss = 0.7082, gen loss = 2.0463
Iteration 5800/9750: dis loss = 0.6061, gen loss = 1.3321
```

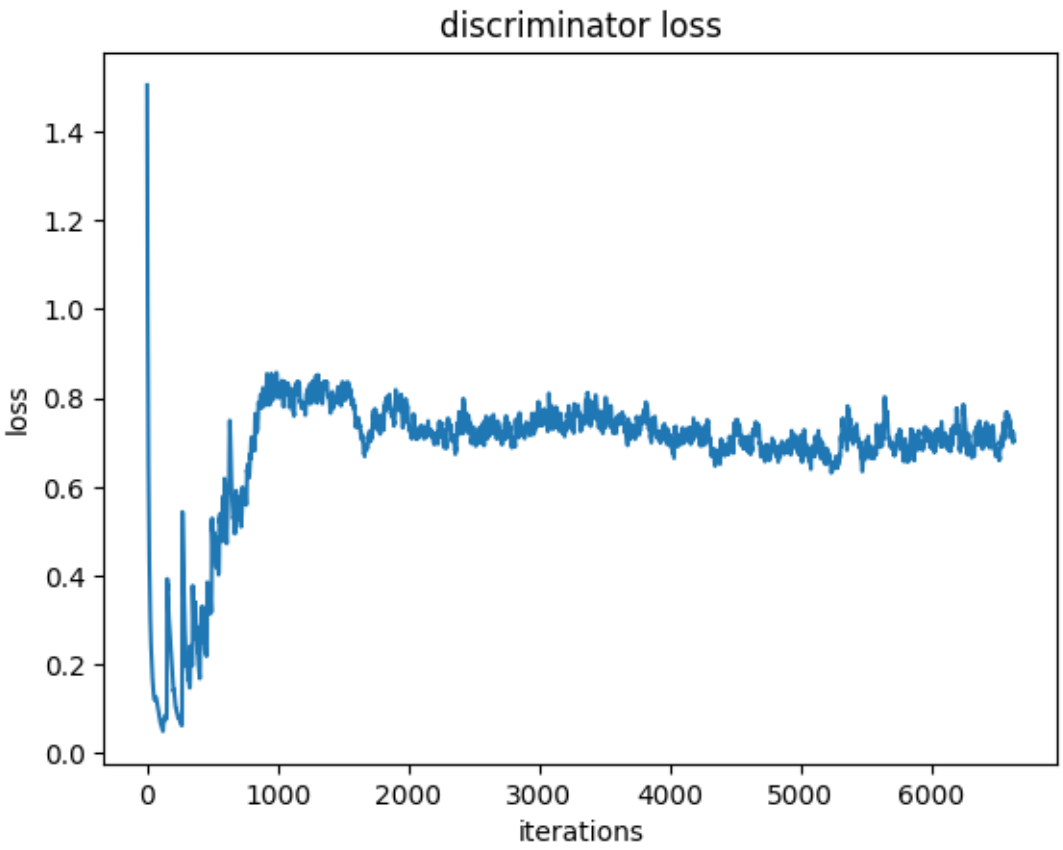

0
50
100
150
200
250
0
50
100
150
200
250

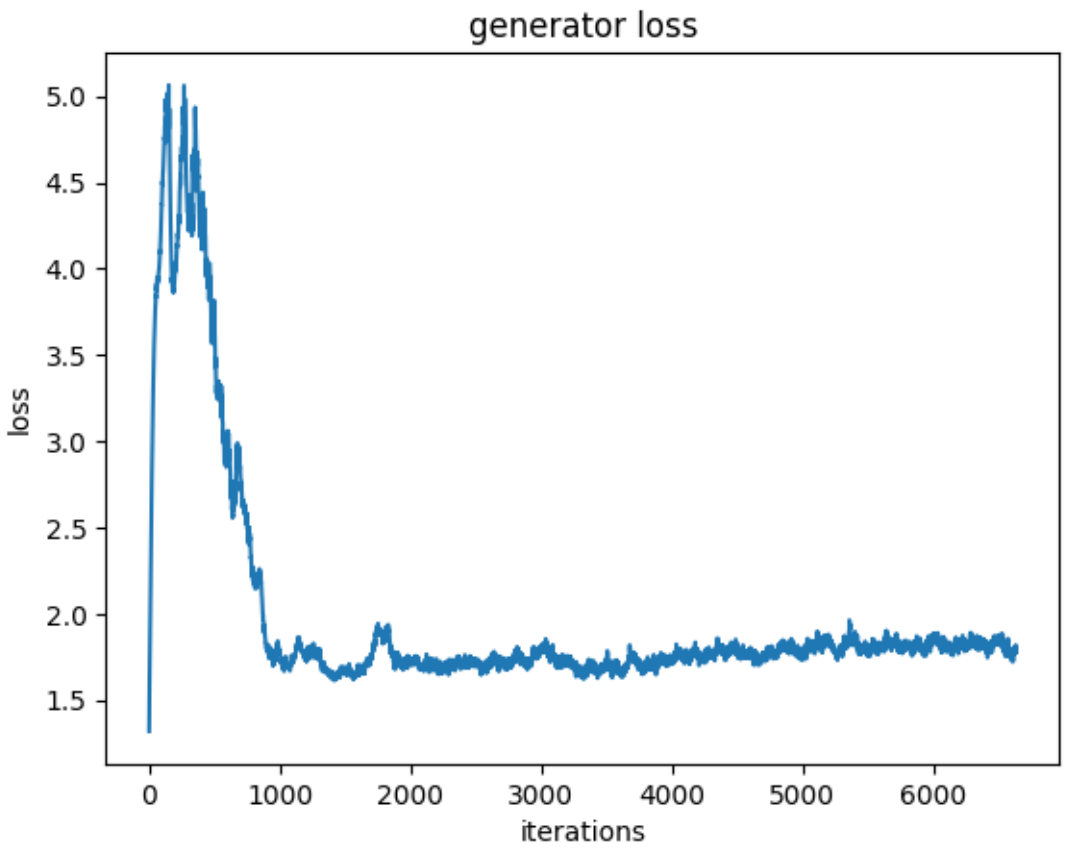

discriminator loss
loss
iterations
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
1000
2000
3000
4000
5000
6000



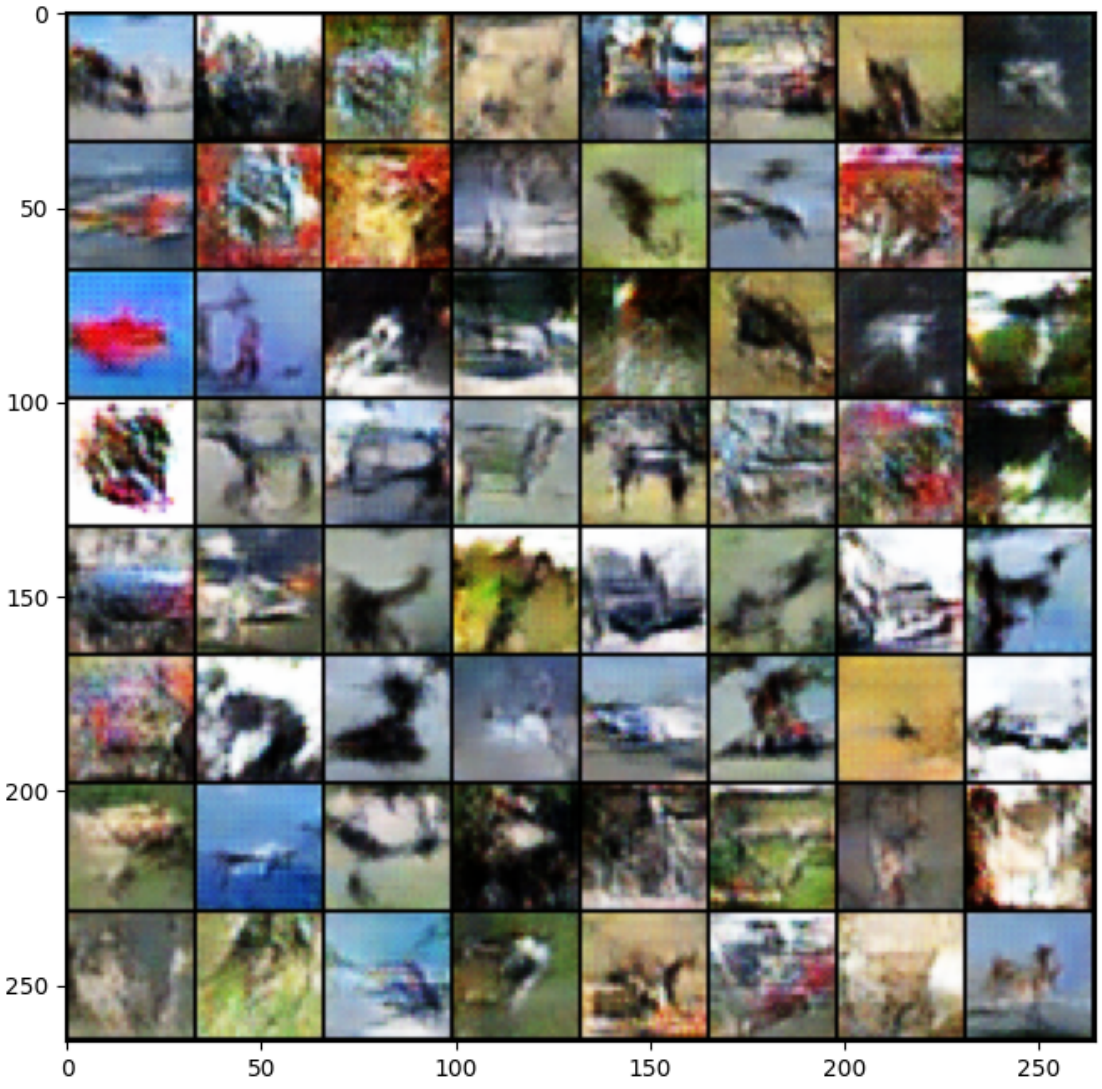

```
Iteration 5900/9750: dis loss = 0.8100, gen loss = 1.1012
Iteration 6000/9750: dis loss = 0.9137, gen loss = 2.9812
Iteration 6100/9750: dis loss = 0.5759, gen loss = 2.3535
Iteration 6200/9750: dis loss = 0.6620, gen loss = 2.0671
Iteration 6300/9750: dis loss = 0.7001, gen loss = 1.4048
Iteration 6400/9750: dis loss = 0.7809, gen loss = 1.3146
Iteration 6500/9750: dis loss = 0.7246, gen loss = 1.3761
Iteration 6600/9750: dis loss = 0.8548, gen loss = 2.4276
```

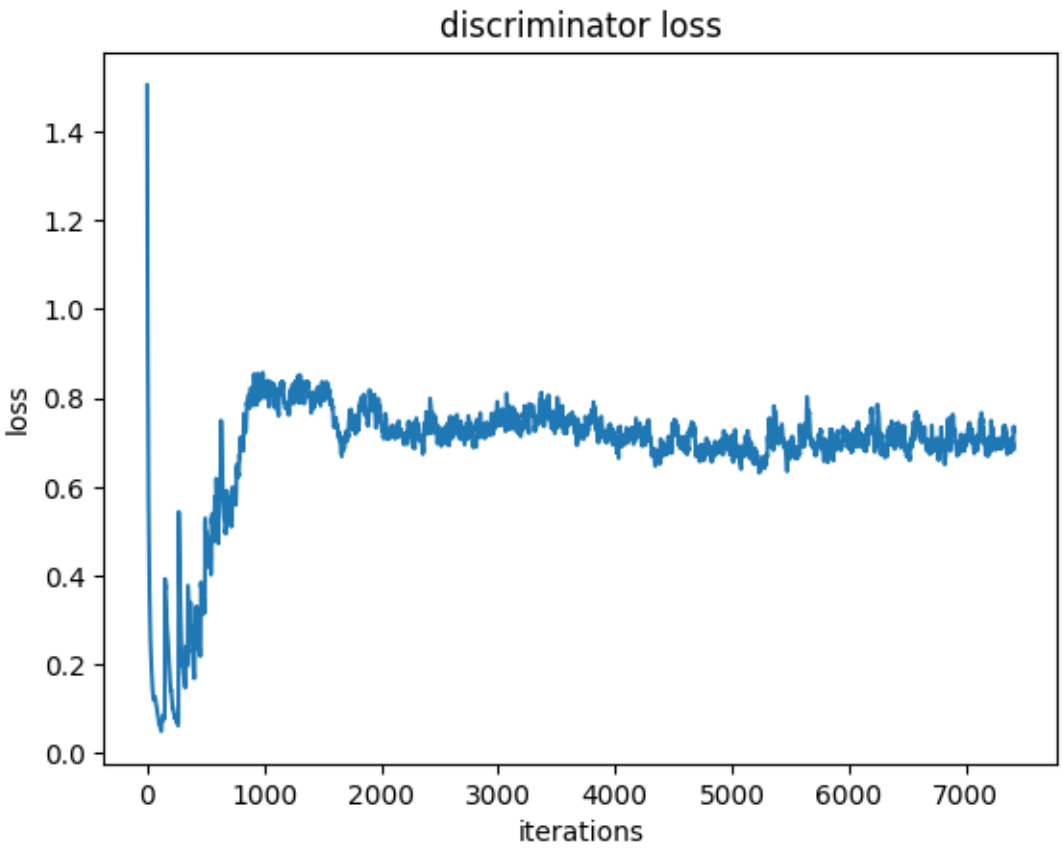

0
50
100
150
200
250
0
50
100
150
200
250

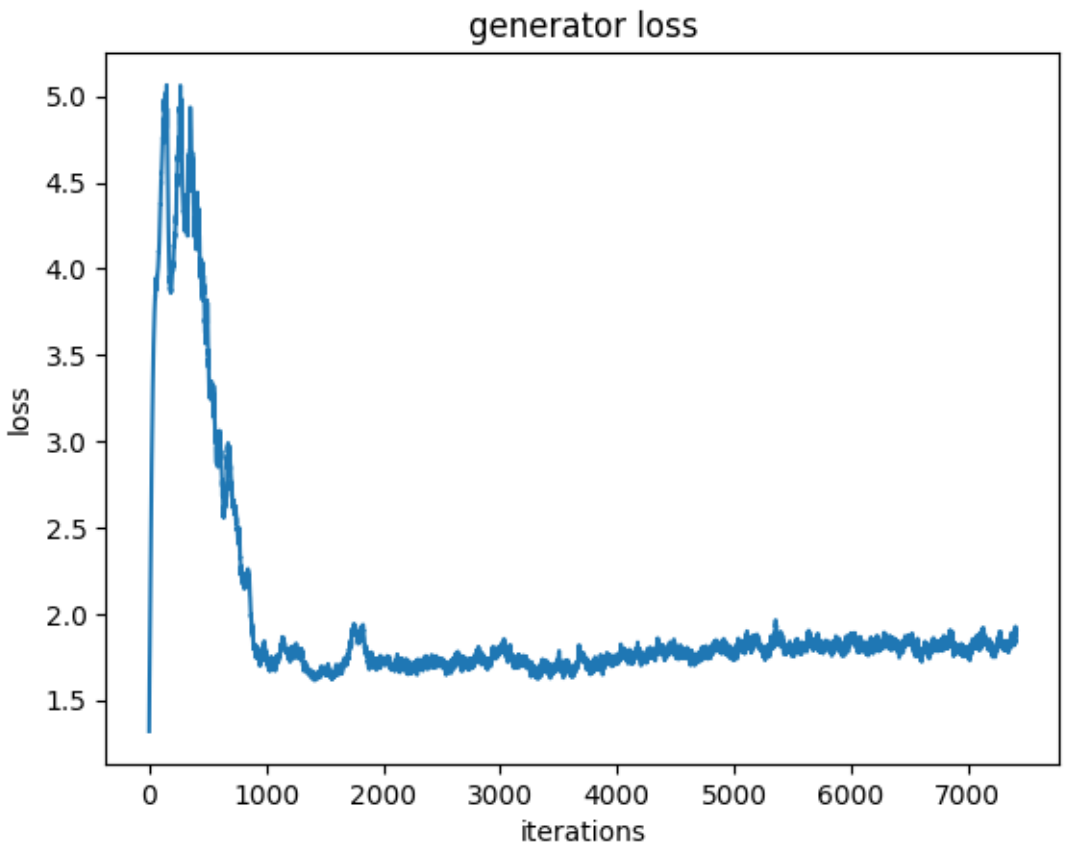

discriminator loss
loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
1000
2000
3000
4000
5000
6000
iterations



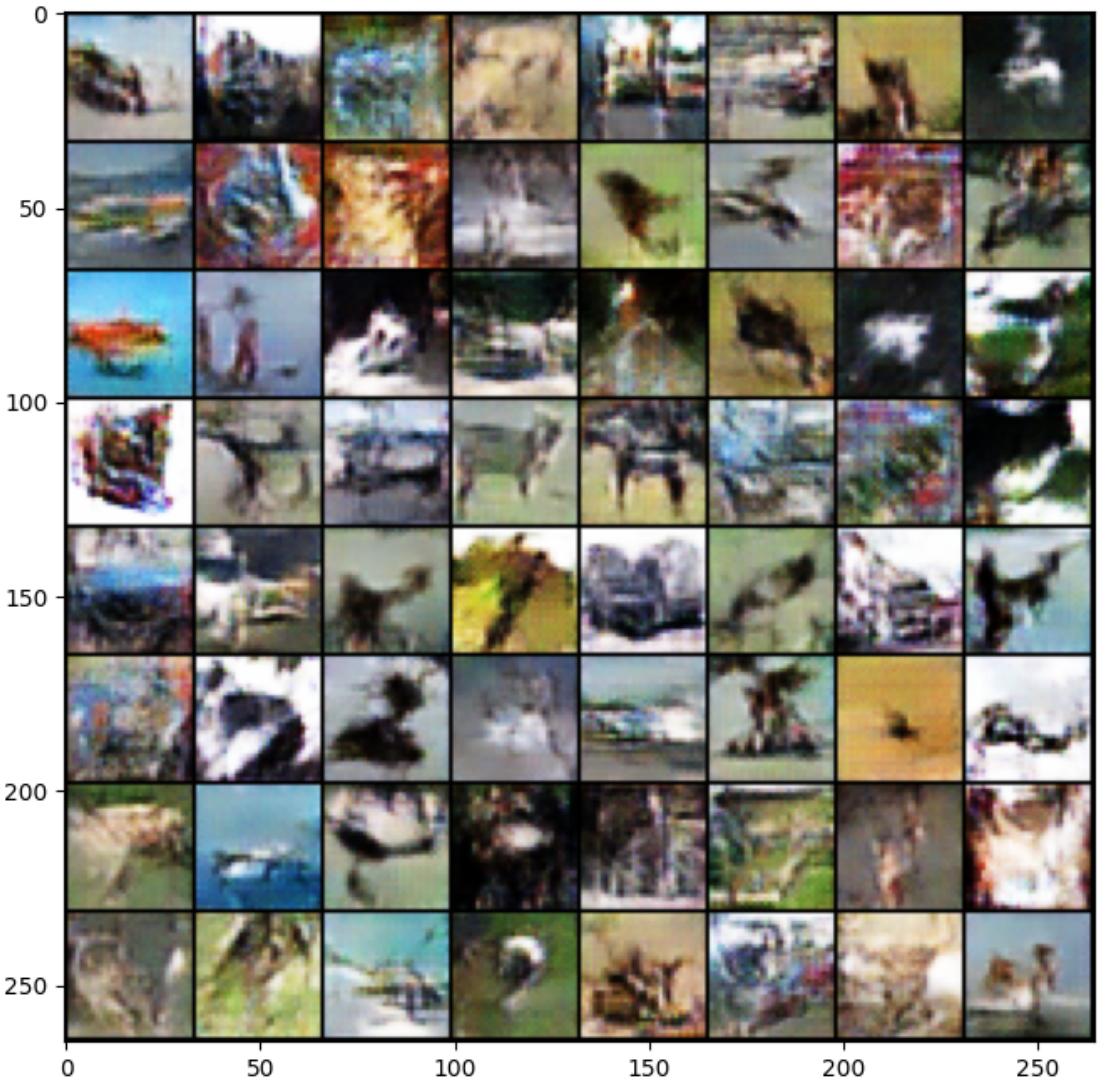

```
Iteration 6700/9750: dis loss = 0.6683, gen loss = 1.4182
Iteration 6800/9750: dis loss = 0.7022, gen loss = 2.2863
Iteration 6900/9750: dis loss = 0.6663, gen loss = 1.2937
Iteration 7000/9750: dis loss = 0.7200, gen loss = 2.1051
Iteration 7100/9750: dis loss = 0.7285, gen loss = 1.2205
Iteration 7200/9750: dis loss = 0.6111, gen loss = 1.7708
Iteration 7300/9750: dis loss = 0.7391, gen loss = 1.8559
Iteration 7400/9750: dis loss = 0.6098, gen loss = 2.1959
```

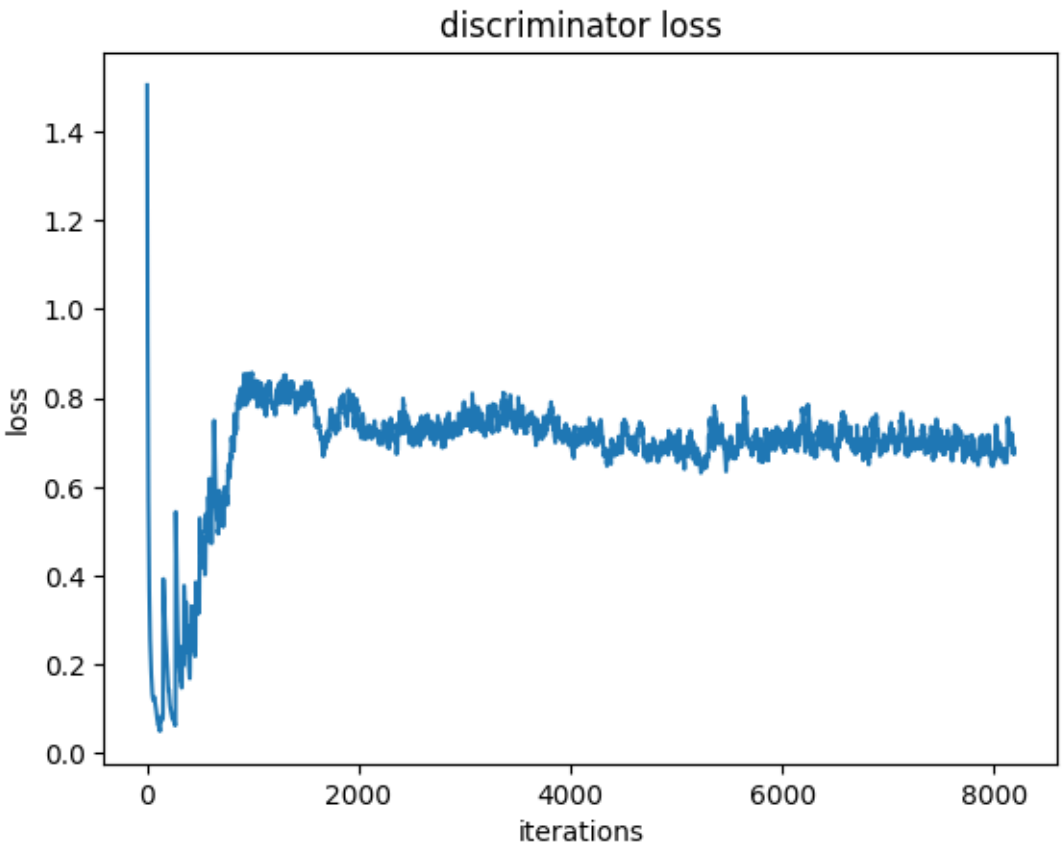

0
50
100
150
200
250
0
50
100
150
200
250

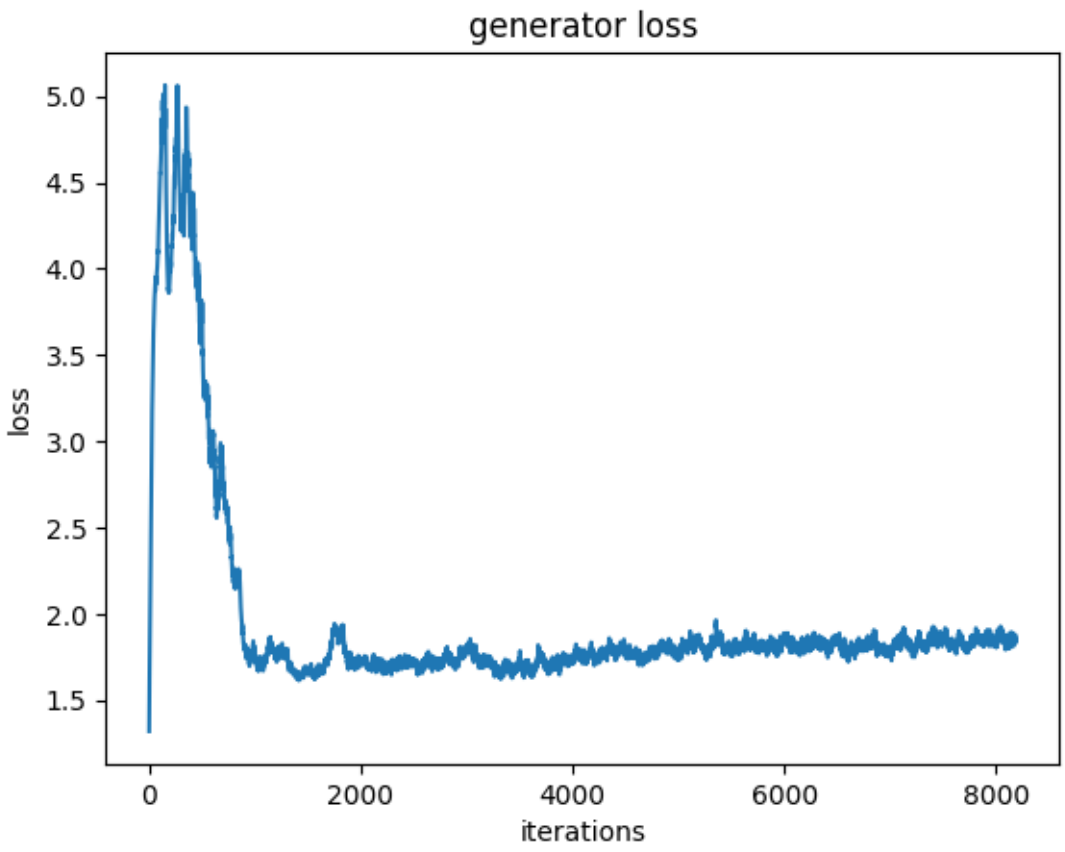

discriminator loss
loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
1000
2000
3000
4000
5000
6000
7000
iterations




```
Iteration 7500/9750: dis loss = 0.5480, gen loss = 2.0510
Iteration 7600/9750: dis loss = 0.5656, gen loss = 2.4216
Iteration 7700/9750: dis loss = 0.6139, gen loss = 2.2831
Iteration 7800/9750: dis loss = 0.6078, gen loss = 1.2666
Iteration 7900/9750: dis loss = 0.5747, gen loss = 1.4137
Iteration 8000/9750: dis loss = 0.6807, gen loss = 1.4610
Iteration 8100/9750: dis loss = 0.8121, gen loss = 1.0371
```

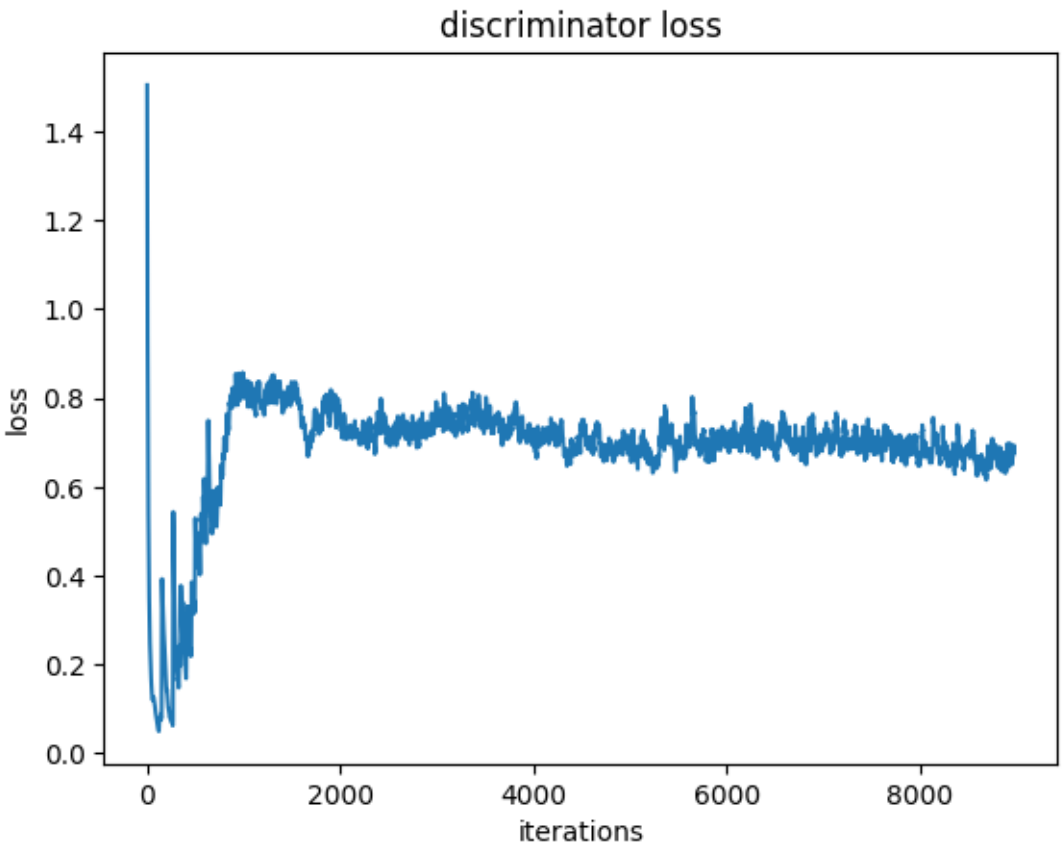

0
50
100
150
200
250
0
50
100
150
200
250

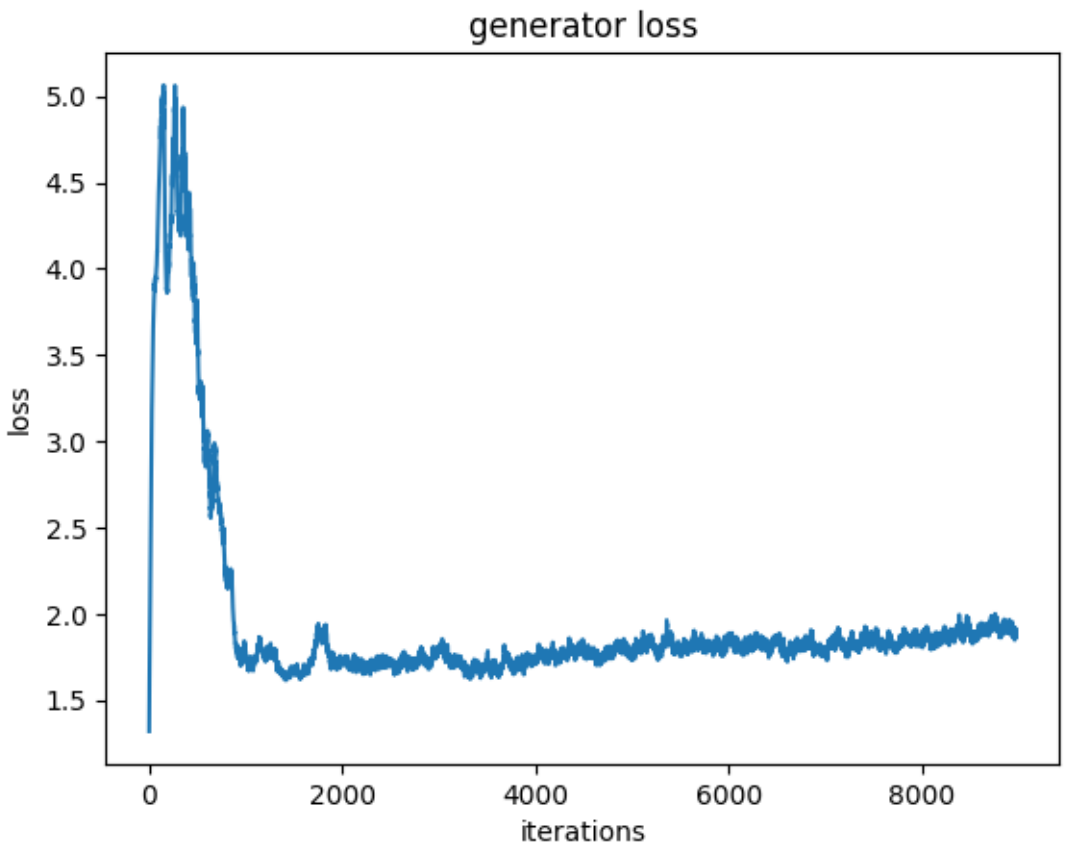

discriminator loss
loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
iterations



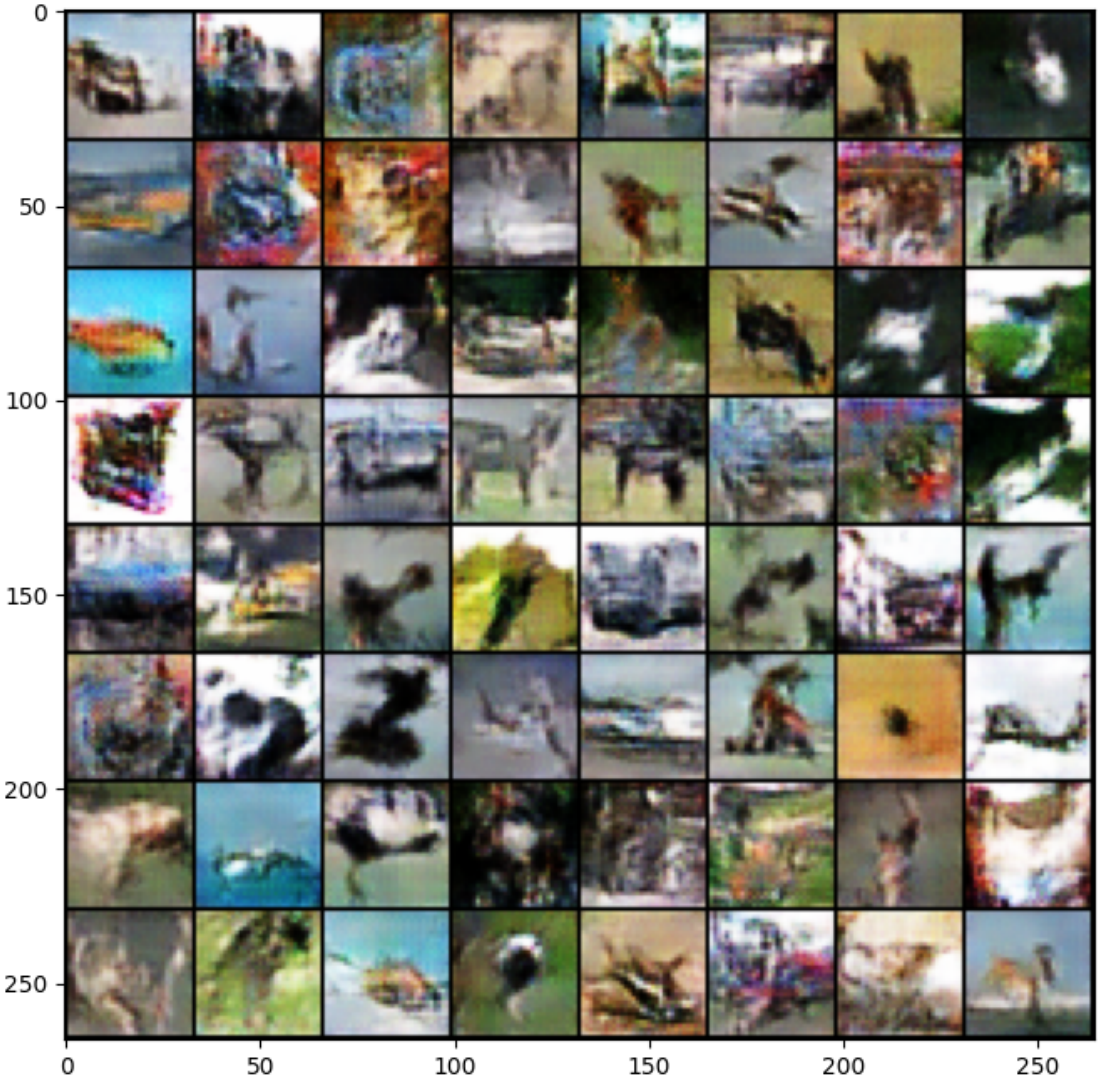

```
Iteration 8200/9750: dis loss = 0.6012, gen loss = 1.7530
Iteration 8300/9750: dis loss = 0.7022, gen loss = 1.3945
Iteration 8400/9750: dis loss = 0.7626, gen loss = 2.6163
Iteration 8500/9750: dis loss = 0.6637, gen loss = 1.3971
Iteration 8600/9750: dis loss = 0.7972, gen loss = 2.3584
Iteration 8700/9750: dis loss = 0.6032, gen loss = 2.1146
Iteration 8800/9750: dis loss = 0.6014, gen loss = 1.9404
Iteration 8900/9750: dis loss = 0.8745, gen loss = 0.8104
```

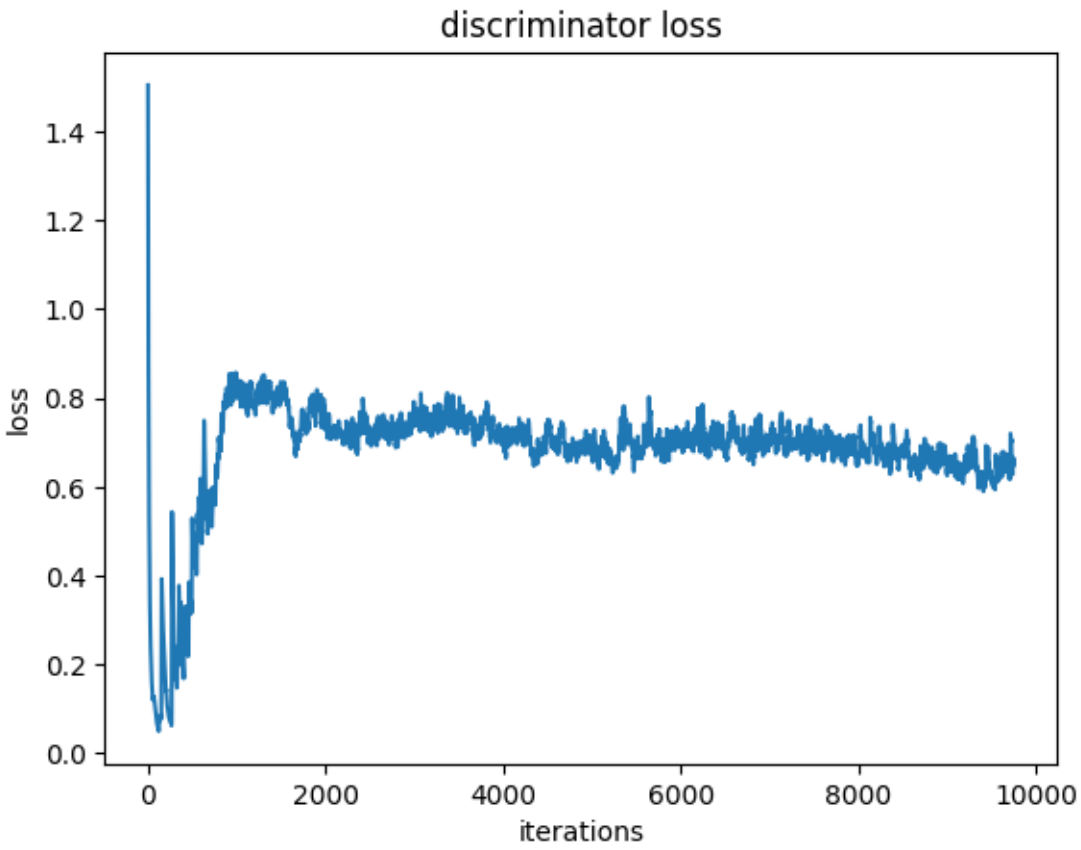

0
50
100
150
200
250
0
50
100
150
200
250

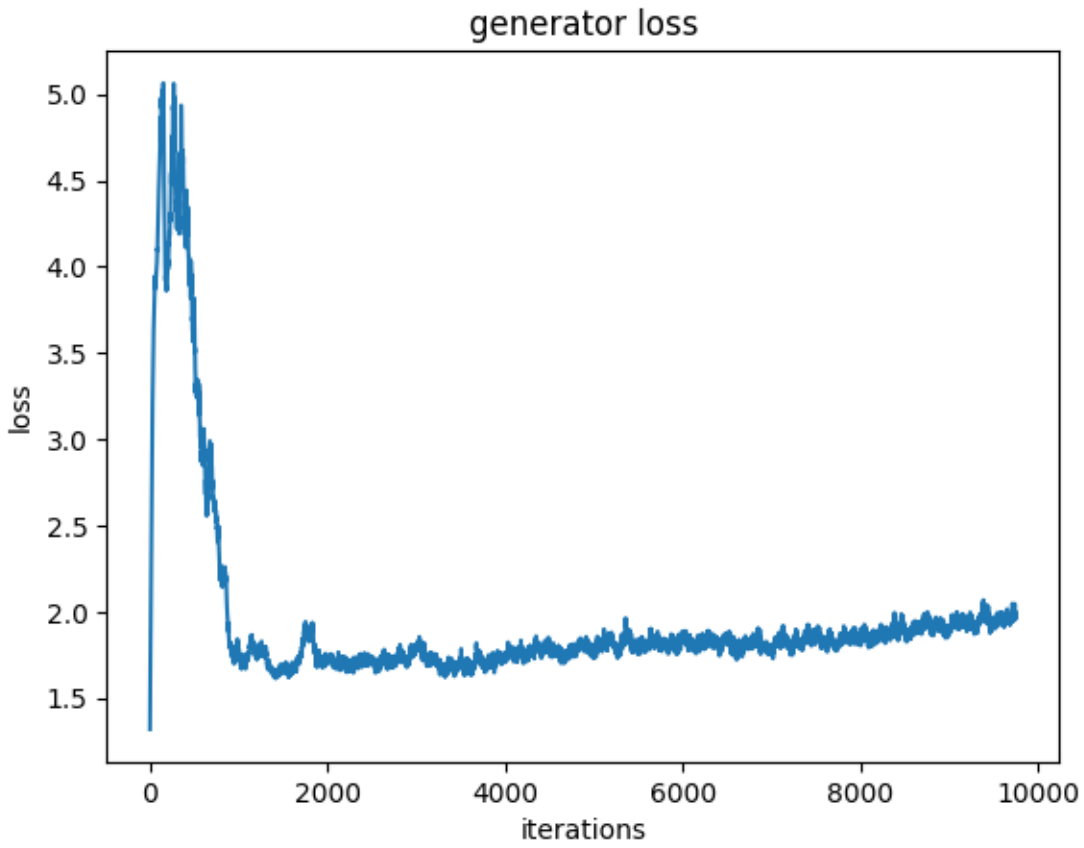

discriminator loss
loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
iterations




```
Iteration 9000/9750: dis loss = 0.5948, gen loss = 2.1644
Iteration 9100/9750: dis loss = 0.5480, gen loss = 2.3501
Iteration 9200/9750: dis loss = 0.5816, gen loss = 1.8595
Iteration 9300/9750: dis loss = 0.7708, gen loss = 1.6167
Iteration 9400/9750: dis loss = 0.5508, gen loss = 1.9674
Iteration 9500/9750: dis loss = 0.5605, gen loss = 2.1007
Iteration 9600/9750: dis loss = 0.8092, gen loss = 1.0856
Iteration 9700/9750: dis loss = 0.5465, gen loss = 1.6152
```


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


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



… Done!

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:

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

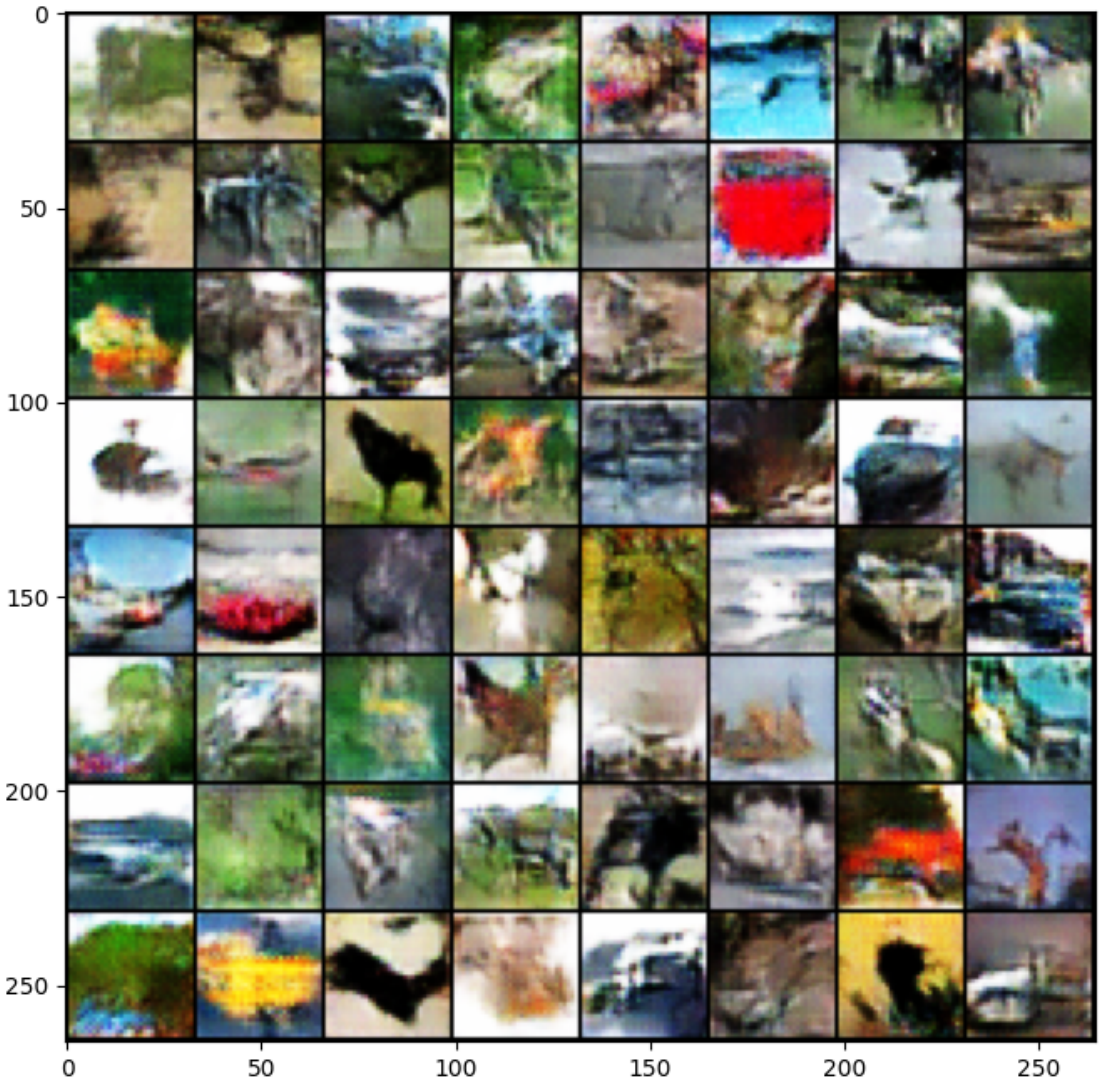
The function to minimize is the same, with the difference being that when training the network we fix a set of input data and find the optimal model parameters, while in activation maximization we fix the model parameters and find the optimal input.

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:

[18]:
```
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:

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

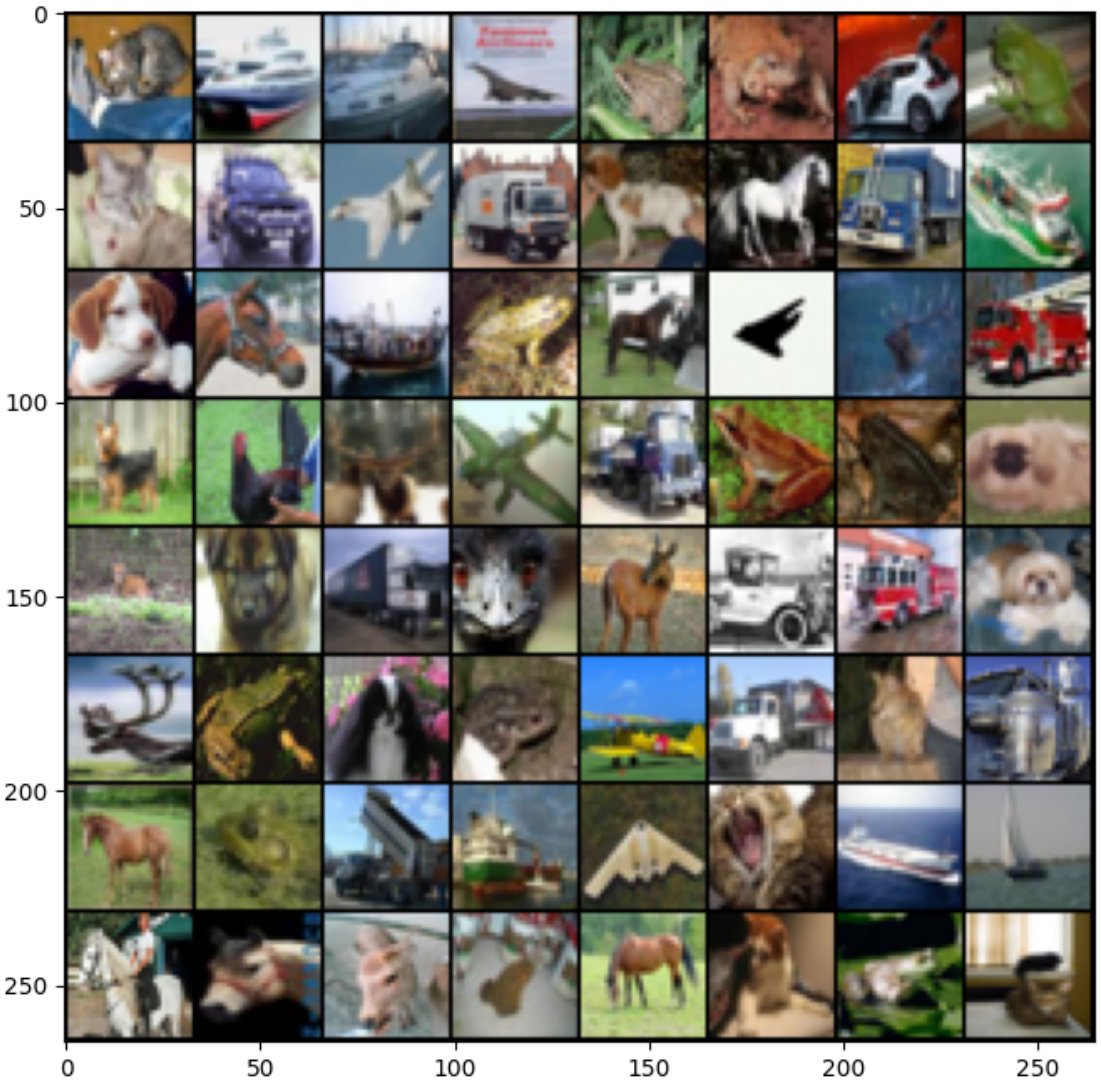
This time, we always start from a zero vector.

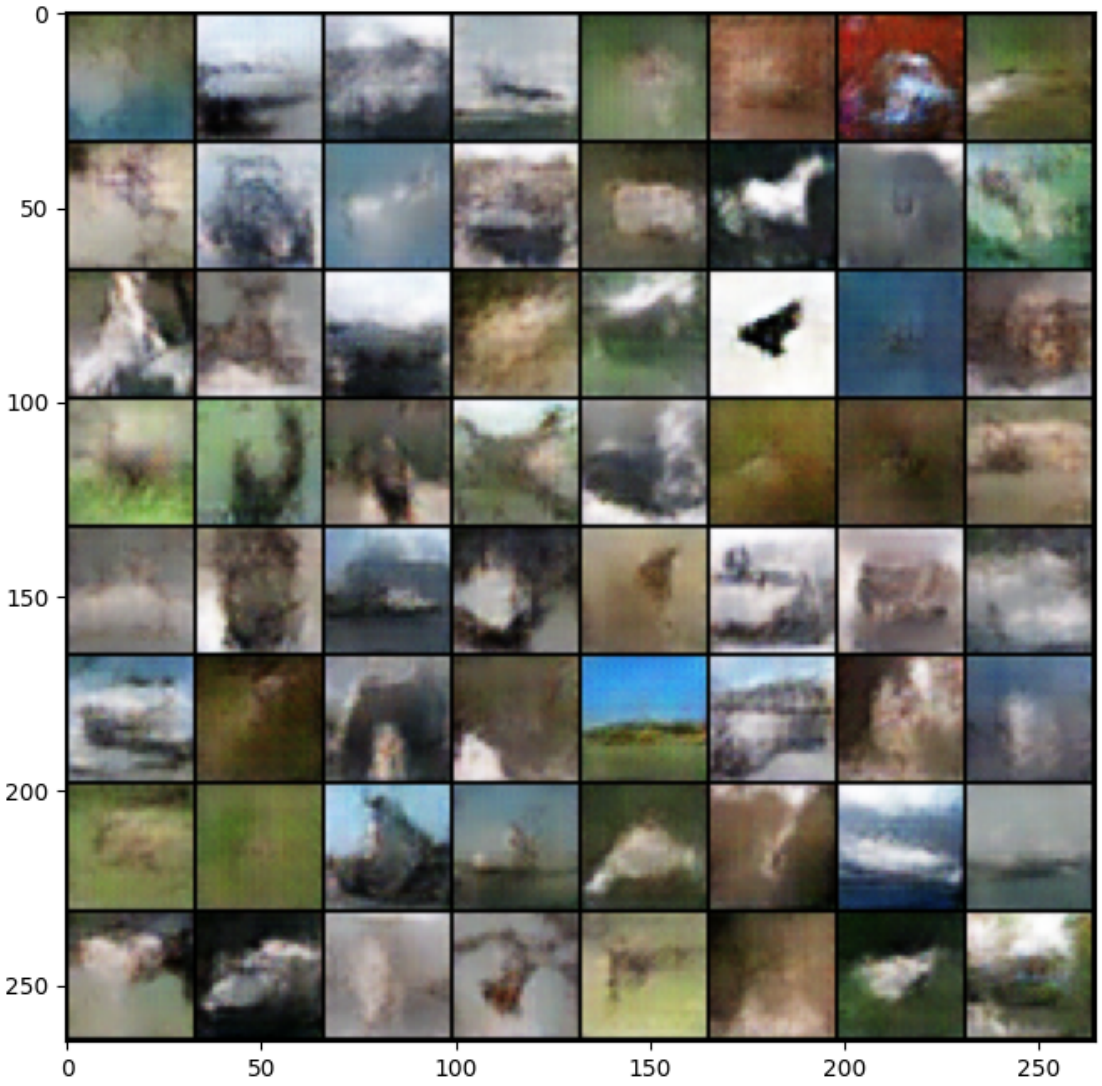
6.1 For this part, you need to complete code blocks marked with "Prob 2-4" above. Then run the following block.

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

[23]:
```
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 = 0.0138
```


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



7 Submission Instruction

See the pinned Piazza post for detailed instruction.

[ ]: