Due Date: April 29th 23:59, 2020
Instructions

- *Read all instructions and questions carefully before you begin.*
- *For all questions, show your work!*
- *The repository for this homework is* https://github.com/CW-Huang/IFT6135H20_assignment

Problem 1

Variational Autoencoders (VAEs) are probabilistic generative models to model data distribution $p(\boldsymbol{x})$. In this question, you will be asked to train a VAE on the *Binarised MNIST* dataset, using the negative ELBO loss as shown in class. Note that each pixel in this image dataset is binary: The pixel is either black or white, which means each datapoint (image) a collection of binary values. You have to model the likelihood $p_\theta(\boldsymbol{x}|\boldsymbol{z})$, i.e. the decoder, as a product of bernoulli distributions.[1]

1. **(`unittest`, 4 pts)** Implement the function '`log_likelihood_bernoulli`' in '`q1_solution.py`' to compute the log-likelihood $\log p(\boldsymbol{x})$ for a given binary sample $\boldsymbol{x}$ and Bernoulli distribution $p(\boldsymbol{x})$. $p(\boldsymbol{x})$ will be parameterized by the mean of the distribution $p(\boldsymbol{x} = 1)$, and this will be given as input for the function.

2. **(`unittest`, 4 pts)** Implement the function '`log_likelihood_normal`' in '`q1_solution.py`' to compute the log-likelihood $\log p(\boldsymbol{x})$ for a given float vector $\boldsymbol{x}$ and isotropic Normal distribution $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2))$. Note that $\boldsymbol{\mu}$ and $\log(\boldsymbol{\sigma}^2)$ will be given for Normal distributions.

3. **(`unittest`, 4 pts)** Implement the function '`log_mean_exp`' in '`q1_solution.py`' to compute the following equation[2] for each $\boldsymbol{y}_i$ in a given $Y = \{\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_i, \ldots \boldsymbol{y}_M\}$;

$$\log \frac{1}{K} \sum_{k=1}^{K} \exp\left(y_i^{(k)} - a_i\right) + a_i,$$

   where $a_i = \max_k y_i^{(k)}$. Note that $\boldsymbol{y}_i = [y_i^{(1)}, y_i^{(2)}, \ldots, y_i^{(k)}, \ldots, y_i^{(K)}]$s.

4. **(`unittest`, 4 pts)** Implement the function '`kl_gaussian_gaussian_analytic`' in '`q1_solution.py`' to compute KL divergence $D_{\text{KL}}\left(q(\boldsymbol{z}|\boldsymbol{x})\|p(\boldsymbol{z})\right)$ via analytic solution for given p and q. Note that p and q are multivariate normal distributions with diagonal covariance.

5. **(`unittest`, 4 pts)** Implement the function '`kl_gaussian_gaussian_mc`' in '`q1_solution.py`' to compute KL diveregence $D_{\text{KL}}\left(q(\boldsymbol{z}|\boldsymbol{x})\|p(\boldsymbol{z})\right)$ by using Monte Carlo estimate for given p and q. Note that p and q are multivariate normal distributions with diagonal covariance.

[1] The binarized MNIST is not interchangeable with the MNIST dataset available on `torchvision`. So the data loader as well as dataset will be provided.

[2] This is a type of log-sum-exp trick to deal with numerical underflow issues: the generation of a number that is too small to be represented in the device meant to store it. For example, probabilities of pixels of image can get really small. For more details of numerical underflow in computing log-probability, see `http://blog.smola.org/post/987977550/log-probabilities-semirings-and-floating-point`.

6. **(report, 15 pts)** Consider a latent variable model $p_\theta(\boldsymbol{x}) = \int p_\theta(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})dz$. The prior is define as $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}_L)$ and $\boldsymbol{z} \in \mathbb{R}^L$. Train a VAE with a latent variable of 100-dimensions ($L = 100$). Use the provided network architecture and hyperparameters described in '`vae.ipynb`'[3]. Use ADAM with a learning rate of 3×10^{-4}, and train for 20 epochs. Evaluate the model on the validation set using the **ELBO**. Marks will neither be deducted nor awarded if you do not use the given architecture. Note that for this question you have to:

   (a) Train a model to achieve an average per-instance ELBO of ≥ -102 on the validation set, and report the ELBO of your model. The ELBO on validation is written as:

$$\frac{1}{|\mathcal{D}_{\text{valid}}|} \sum_{\boldsymbol{x}_i \in \mathcal{D}_{\text{valid}}} \mathcal{L}_{\text{ELBO}}(\boldsymbol{x}_i) \geq -102$$

   Feel free to modify the above hyperparameters (except the latent variable size) to ensure it works.

7. **(report, 15 pts)** Evaluate *log-likelihood* of the trained VAE models by using importance sampling, which was covered during the lecture. Use the codes described in '`vae.ipynb`'. The formula is reproduced here with additional details:

$$\log p(\boldsymbol{x} = \boldsymbol{x}_i) \approx \log \frac{1}{K} \sum_{k=1}^{K} \frac{p_\theta(\boldsymbol{x} = \boldsymbol{x}_i|\boldsymbol{z}_i^{(k)})\ p(\boldsymbol{z} = \boldsymbol{z}_i^{(k)})}{q_\phi(\boldsymbol{z} = \boldsymbol{z}_i^{(k)}|\boldsymbol{x}_i)}; \quad \text{for all } k\text{: } \boldsymbol{z}_i^{(k)} \sim q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)$$

   and $\boldsymbol{x}_i \in \mathcal{D}$.

   (a) Report your evaluations of the trained model on the test set using the log-likelihood estimate ($\frac{1}{N}\sum_{i=1}^{N} \log p(\boldsymbol{x}_i)$), where N is the size of the test dataset. Use $K = 200$ as the number of importance samples, D as the dimension of the input ($D = 784$ in the case of MNIST), and $L = 100$ as the dimension of the latent variable.

1. The answer is submitted to gradescope.
2. The answer is submitted to gradescope.
3. The answer is submitted to gradescope.
4. The answer is submitted to gradescope.
5. The answer is submitted to gradescope.
6. In this section I trained a model which was given in 'vae.ipynb'. Our goal was to achieve an average per-instance ELBO of ≥ -102 on the validation set. I showed the results of ELBO on the valid dataset in Figure 1. **Wasserstein distance:**

[3]This file is executable in Google Colab. You can also convert `vae.ipynb` to `vae.py` using the Colab.

```
[10]
        optimizer.zero_grad()
        z_mean, z_logvar, x_mean = vae(x)
        loss = vae.loss(x, z_mean, z_logvar, x_mean)
        loss.backward()
        optimizer.step()

    # evaluate ELBO on the valid dataset
    with torch.no_grad():
        total_loss = 0.
        total_count = 0
        for x in valid:
            total_loss += vae.loss(x, *vae(x)) * x.size(0)
            total_count += x.size(0)
        print('-elbo: ', (total_loss / total_count).item())
```

```
-elbo:  166.2794189453125
-elbo:  144.00030517578125
-elbo:  129.95834350585938
-elbo:  120.61237335205078
-elbo:  116.21745300292969
-elbo:  113.4291763305664
-elbo:  110.90296173095703
-elbo:  109.21097564697266
-elbo:  107.70173645019531
-elbo:  106.41007232666016
-elbo:  105.61347198486328
-elbo:  104.72156524658203
-elbo:  104.18782043457031
-elbo:  103.55335235595703
-elbo:  103.24345397949219
-elbo:  102.77406311035156
-elbo:  102.38977813720703
-elbo:  102.0827407836914
-elbo:  101.8590316772461
-elbo:  101.36197662353516
```

Figure 1: Report ELBO on the valid dataset

7. The evaluations of the trained model on the test set using the log-likelihood estimate is shown in Figure 2. :

```
[14]
        # Reshape images and posterior to evaluate probabilities
        x_flat = x[:, None].repeat(1, K, 1, 1, 1).reshape(M*K, -1)
        z_mean_flat = z_mean[:, None, :].expand_as(z_samples).reshape(M*K, -1)
        z_logvar_flat =  z_logvar[:, None, :].expand_as(z_samples).reshape(M*K, -1)
        ZEROS = torch.zeros(z_mean_flat.size())

        # Calculate all the probabilities!
        log_p_x_z = log_likelihood_bernoulli(torch.sigmoid(x_mean_flat), x_flat).view(M, K)
        log_q_z_x = log_likelihood_normal(z_mean_flat, z_logvar_flat, z_samples_flat).view(M, K)
        log_p_z = log_likelihood_normal(ZEROS, ZEROS, z_samples_flat).view(M, K)

        # Recombine them.
        w = log_p_x_z + log_p_z - log_q_z_x
        log_p = log_mean_exp(w)

        # Accumulate
        total_loss += log_p.sum()
        total_count += M

print('log p(x):', (total_loss / total_count).item())

log p(x): -95.82352447509766
```

Figure 2: evaluations on the test set using the log-likelihood estimatete

Problem 2

Generative Adversarial Network (GAN) enables the estimation of distributional measure between arbitrary empirical distributions. In this question, you will first implement a function to estimate the Squared Hellinger as well as one to estimate the Earth mover distance. This will allow you to look at and contrast some properties of the f-divergence [4] and the Earth-Mover distance [5].

We provide samplers [6] to generate the different distributions that you will need for this question. In the same repository, we also provide the architecture of a neural network function Critic : $\mathcal{X} \to \mathbb{R}$ s.t. $\mathcal{X} \subset \mathbb{R}^2$ in model.py. For training, you may use SGD with a learning rate of $1e-3$ and a mini batch size of 512.

1. **(`report`, 4 pts)** Provide the objective function of the Squared Hellinger in your report (See Nowozin et al. – Footnote 4).

2. **(`unittest`, 4 pts)** Implement the function '`vf_squared_hellinger`' in '`q2_solution.py`' to compute the objective function for estimating the Squared Hellinger distance. Please, consider the definition given in Nowozin. We give more instruction in the code template.

3. **(`report`, 4 pts)** In your report, provide the objective function of the Wasserstein distance and the objective function of the "*Lipschitz Penalty*" [7]

4. **(`unittest`, 4 pts)** Implement the functions '`vf_wasserstein_distance`' and '`lp_reg`' in '`q2_solution.py`' to compute the objective function of the Wasserstein distance and compute the "*Lipschitz Penalty*". Consider that the norm used in the regularizer is the $L2$-norm.

5. **(`report`, 10 pts)** Let $u \sim U[0,1]$ be the uniform random variable with support in the interval $[0,1]$. We define p the distribution of $(0,u) \in \mathbb{R}^2$ and q the distribution of $(\theta,u) \in \mathbb{R}^2$ (We provide a function generating the p and q in the file q2_samplers.py). Plot the estimated Squared Hellinger distance between p and q and the estimated Earth-Mover distance between p and q for $\theta \in [0,2]$ with interval of 0.1 (i.e. 21 points). The x-axis should be the value of θ and the y-axis should be your estimate. In your report, provide two plots: a plot of the estimate of the Squared Hellinger with respect to θ and a plot of the estimate of the Wasserstein distance with respect to θ. Also, provide a one or two lines explanation of the behaviour you observe.

[4] Relevant reference on f-divergence: https://arxiv.org/abs/1606.00709
[5] Relevant reference on Wasserstein GAN: https://arxiv.org/abs/1701.07875
[6] See the assignment repository https://github.com/CW-Huang/IFT6135H20_assignment
[7] See Section 5 of https://arxiv.org/pdf/1709.08894.pdf

1. Squared Hellinger distance objective function is :

$$\mathcal{F}(\theta, \omega) = \mathbb{E}_{\boldsymbol{x}\sim P}[g_f(V_\omega(\boldsymbol{x}))] + \mathbb{E}_{\boldsymbol{y}\sim Q}[-^*(g_f(V_\omega(\boldsymbol{y})))]$$

Where Q is the generative model and we can parametrize it using a vector θ. We know that x are samples from the distribution P and y are samples from the distribution Q. In the original paper they used $\boldsymbol{x} \sim Q$ but I find using different variable helps to understand easier. From table 2 in the paper we know that the output activation

$$g_f = 1 - exp(-v), \qquad g_f^* = \frac{t}{1-t}$$

Also $V_\omega()$ is the same as critic in our case. The implemented version of the objective is shown in Figure 5.

```
def vf_squared_hellinger(x, y, critic):
    """
    Complete me. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests might fail.

    *** The notation used for the parameters follow the one from Nowazin et al: https://ar
    In other word, x are samples from the distribution P and y are samples from the distri

    :param p: (FloatTensor) - shape: (batchsize x featuresize) - Samples from a distributi
    :param q: (FloatTensor) - shape: (batchsize x featuresize) - Samples from a distributi
    :param critic: (Module) - torch module used to compute the Squared Hellinger.
    :return: (FloatTensor) - shape: (1,) - Estimate of the Squared Hellinger
    """

    gf_y = 1 - torch.exp(-critic(y))
    gf_x = 1 - torch.exp(-critic(x))
    return torch.mean(gf_x) - torch.mean(gf_y / (torch.tensor([1.0]) - gf_y))
      Shima Shahfar, 7 days ago • Course Materials
```

Figure 3: Squared Hellinger distance function.

2. The answer is submitted to gradescope.

3. The objective function for Wasserstein distance is $\mathcal{W}(\mu, \boldsymbol{v}) = \mathbb{E}_{\boldsymbol{x}\sim\mu}[f(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{y}\sim\boldsymbol{v}}[f(\boldsymbol{y})]$. The objective function of the "*Lipschitz Penalty*" is equal to

$$lp - penalty = \lambda \mathbb{E}_{\hat{x}\sim\tau}[(||f(\hat{x})||_2 - 1)^2]$$

Where τ is the domain of $\hat{x}$ and $\hat{x} = t * \boldsymbol{x} + (1-t) * \boldsymbol{y}$, $t \sim \mathcal{U}[0,1]$ ans $\boldsymbol{x} \sim \mu$, $\boldsymbol{y} \sim \boldsymbol{v}$ are real and generated samples.

4. The answer is submitted to gradescope.

5. Given the plots provided for this part, we can conclude that Squared Hellinger distance is more probable to converge faster.

   - **Wasserstein distance:**

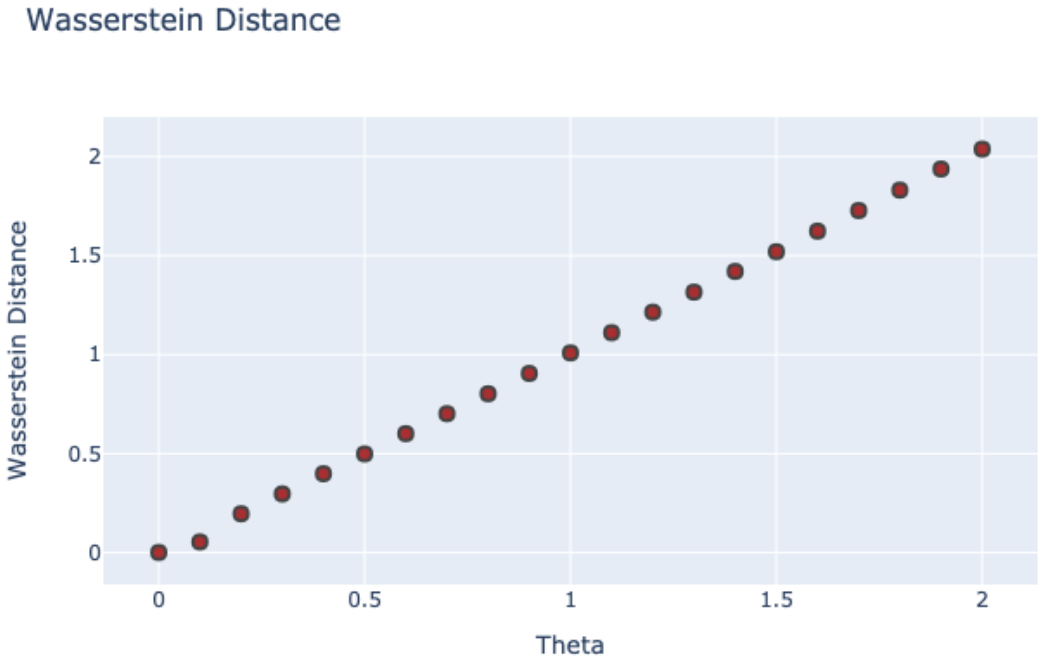

Figure 4: The estimate of Wasserstein distance with respect to θ.

- **Squared Hellinger distance:**

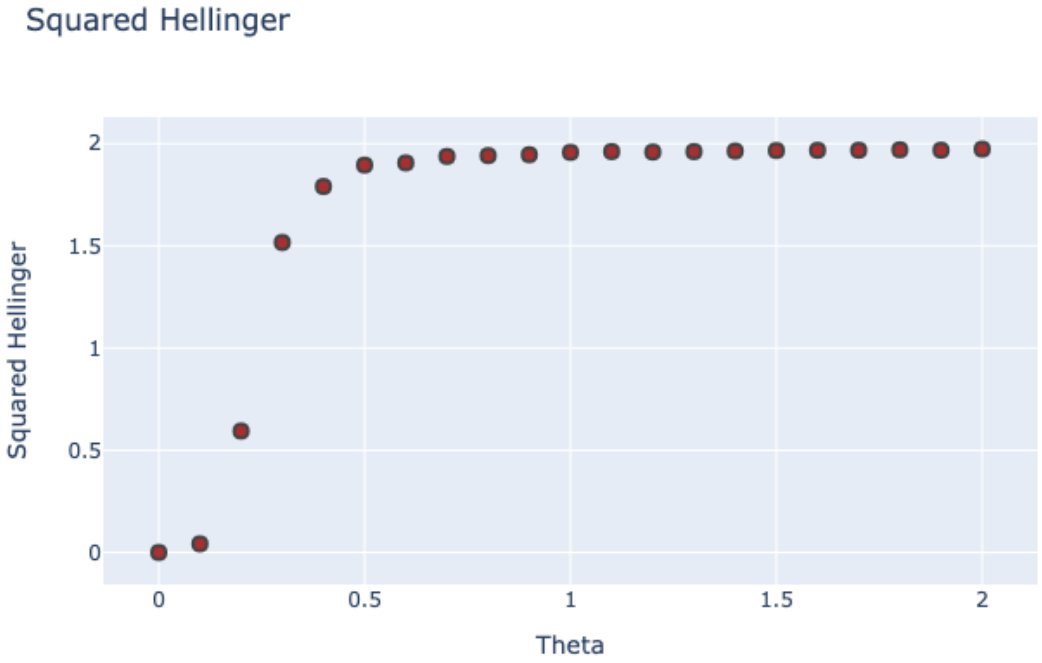

Figure 5: The estimate of Squared Hellinger distance with respect to θ.

Problem 3

Recent years have shown an explosion of research into using deep learning and computer vision algorithms to generate images.

In this final question, you will use the GAN framework train a generator to generate a distribution of high dimensional images $\mathcal{X} \subset \mathbb{R}^{32\times32\times3}$, namely the **Street View House Numbers** dataset (SVHN) [8]. We will consider the prior distribution $p(z) = \mathcal{N}(\mathbf{0}, I)$ the isotropic gaussian distribution. We provide a function for sampling from the SVHN datasets in 'q3_samplers.py'.

Hyperparameters & Training Pointers We provide code for the GANs network as well as the hyperparameters you should be using. We ask you to code the training procedure to train the GANs as well as the qualitative exploration that you will include in your report. You can re-use the WGAN-lp objective you wrote in the the previous question.

Qualitative Evaluation In your report,

1. **(`report`, 8 pts) Provide visual samples.** Comment the quality of the samples from each model (e.g. blurriness, diversity).

[8] The SVHN dataset can be downloaded at http://ufldl.stanford.edu/housenumbers/

2. (`report`, **8 pts) We want to see if the model has learned a disentangled representation in the latent space.** Sample a random z from your prior distribution. Make small perturbations to your sample z for *each dimension* (e.g. for a dimension i, $z'_i = z_i + \epsilon$). ϵ has to be large enough to see some visual difference. For each dimension, observe if the changes result in visual variations (that means variations in $g(z)$). You do not have to show all dimensions, just a couple that result in interesting changes.

3. (`report`, **8 pts) Compare between interpolating in the data space and in the latent space.** Pick two random points z_0 and z_1 in the latent space sampled from the prior.

   (a) For $\alpha = 0, 0.1, 0.2 \ldots 1$ compute $z'_\alpha = \alpha z_0 + (1 - \alpha) z_1$ and plot the resulting samples $x'_\alpha = g(z'_\alpha)$.

   (b) Using the data samples $x_0 = g(z_0)$ and $x_1 = g(z_1)$ and for $\alpha = 0, 0.1, 0.2 \ldots 1$ plot the samples $\hat{x}_\alpha = \alpha x_0 + (1 - \alpha) x_1$.

   Explain the difference with the two schemes to interpolate between images.

   Answer: In this assignment we were asked to first write a training loop for WGAN-LP using the "*Lipschitz Penalty*" which we implemented in problem 2 and train it on SVHN dataset. In Figure **??** you can find the training loop I've used for this part. For this part I've plotted samples using the 256 fixed noises so that I can analyse the improvements of the generator. The generated images are blurry and less diverse in the beginning of the training. I have plotted after every 5K iterations but we can see that as we go on in the training the generator is first learning some simple structures and then some blurry numbers. For instance the generator learned to generate some numbers after 10000 iterations (see Figure 13). But when you look iteration 5000 (Figure 12) the images are more blurry and less diverse.

```
# COMPLETE TRAINING PROCEDURE
for iteration in range(n_iter):
    # train critic
    for critic_iter in range(n_critic_updates):
        images, labels = next(train_loader)
        real_data = images.to(device)
        noise = torch.randn((train_batch_size, z_dim, 1, 1)).to(device)
        generated_data = generator(noise)

        lipschitz_penalty = lp_coeff * lp_reg(
            real_data, generated_data, critic
        )
        w_dist = (critic(real_data) - critic(generated_data)).mean()
        c_loss = -w_dist + lipschitz_penalty

        critic.zero_grad()
        c_loss.backward()
        optim_critic.step()

    # train generator
    noise = torch.randn((train_batch_size, z_dim, 1, 1)).to(device)
    generated_data = generator(noise)
    f_generated = critic(generated_data).mean()
    g_loss = -f_generated

    generator.zero_grad()
    g_loss.backward()
    optim_generator.step()
```

Figure 6: Training Loop for WGAN-LP

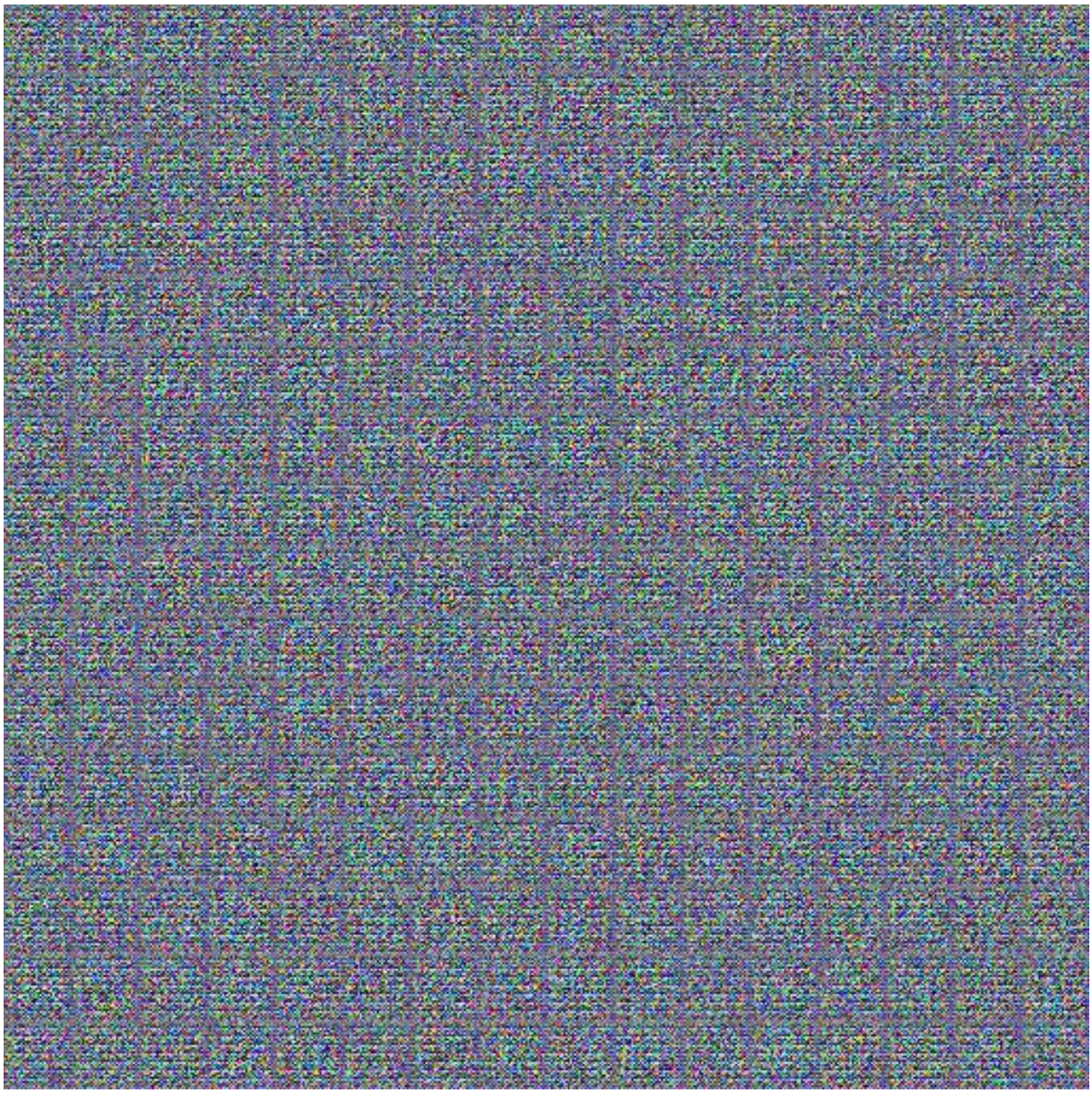

Figure 7: Iteration 0

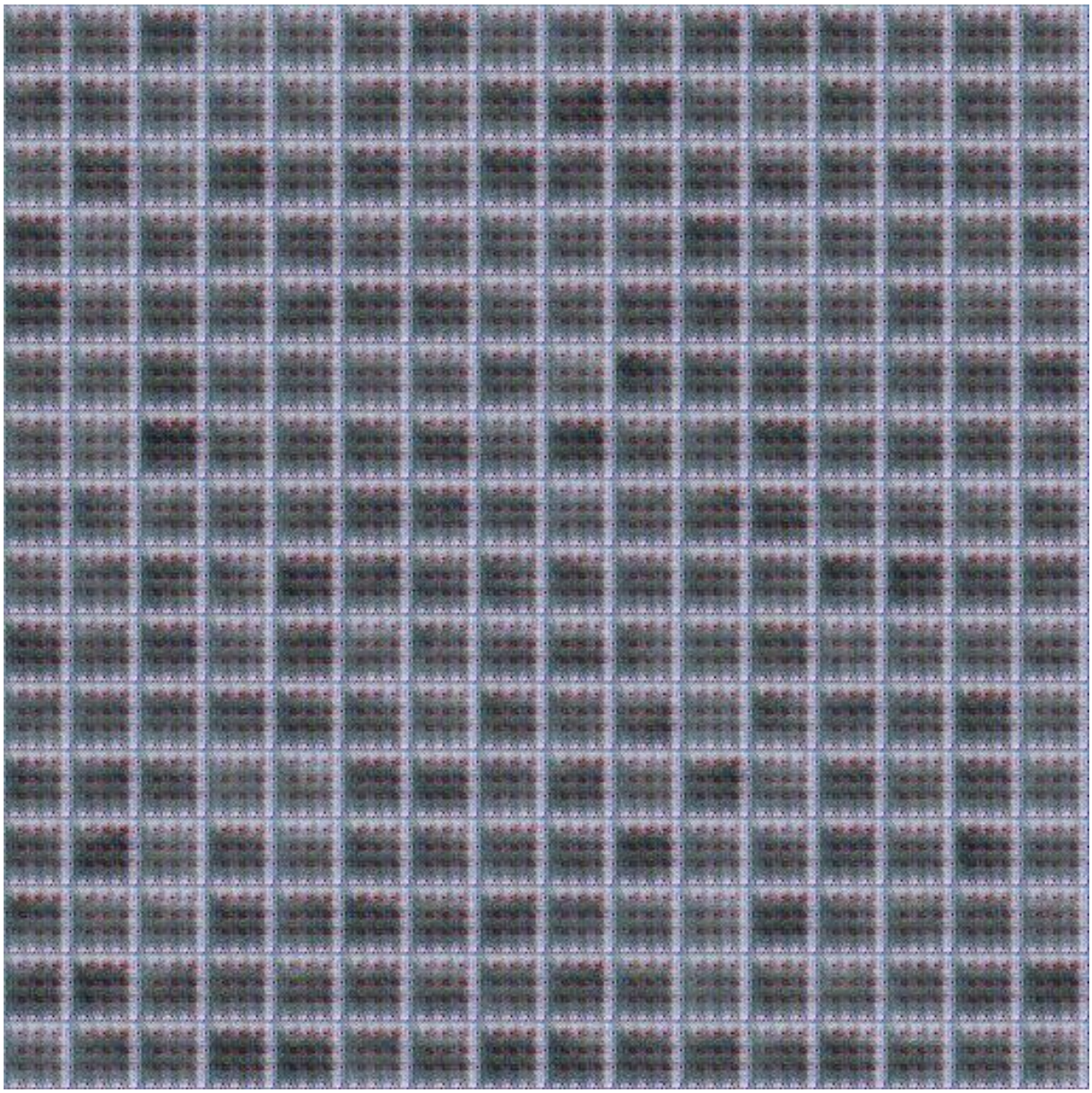

Figure 8: Iteration 100

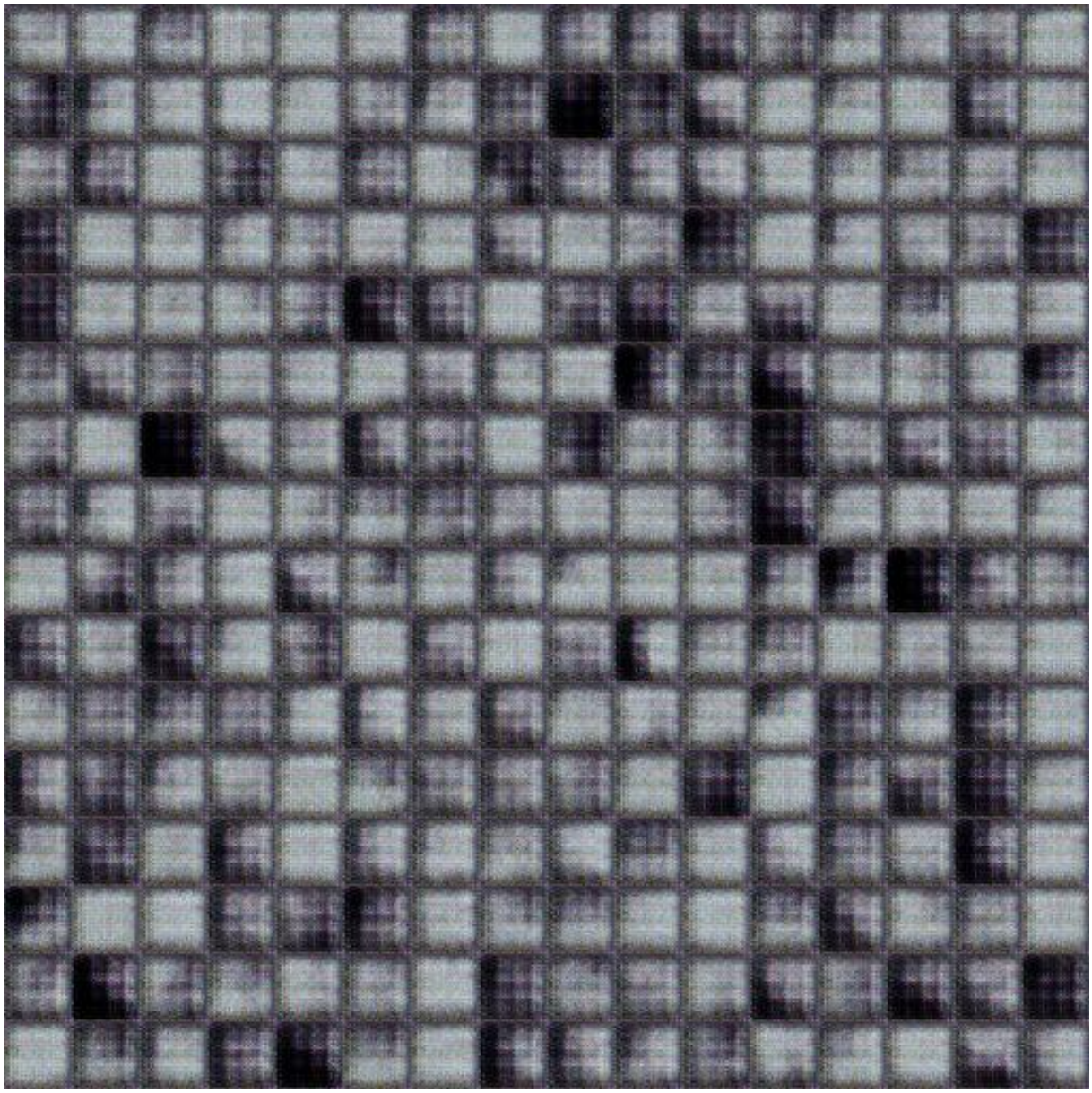

Figure 9: Iteration 200

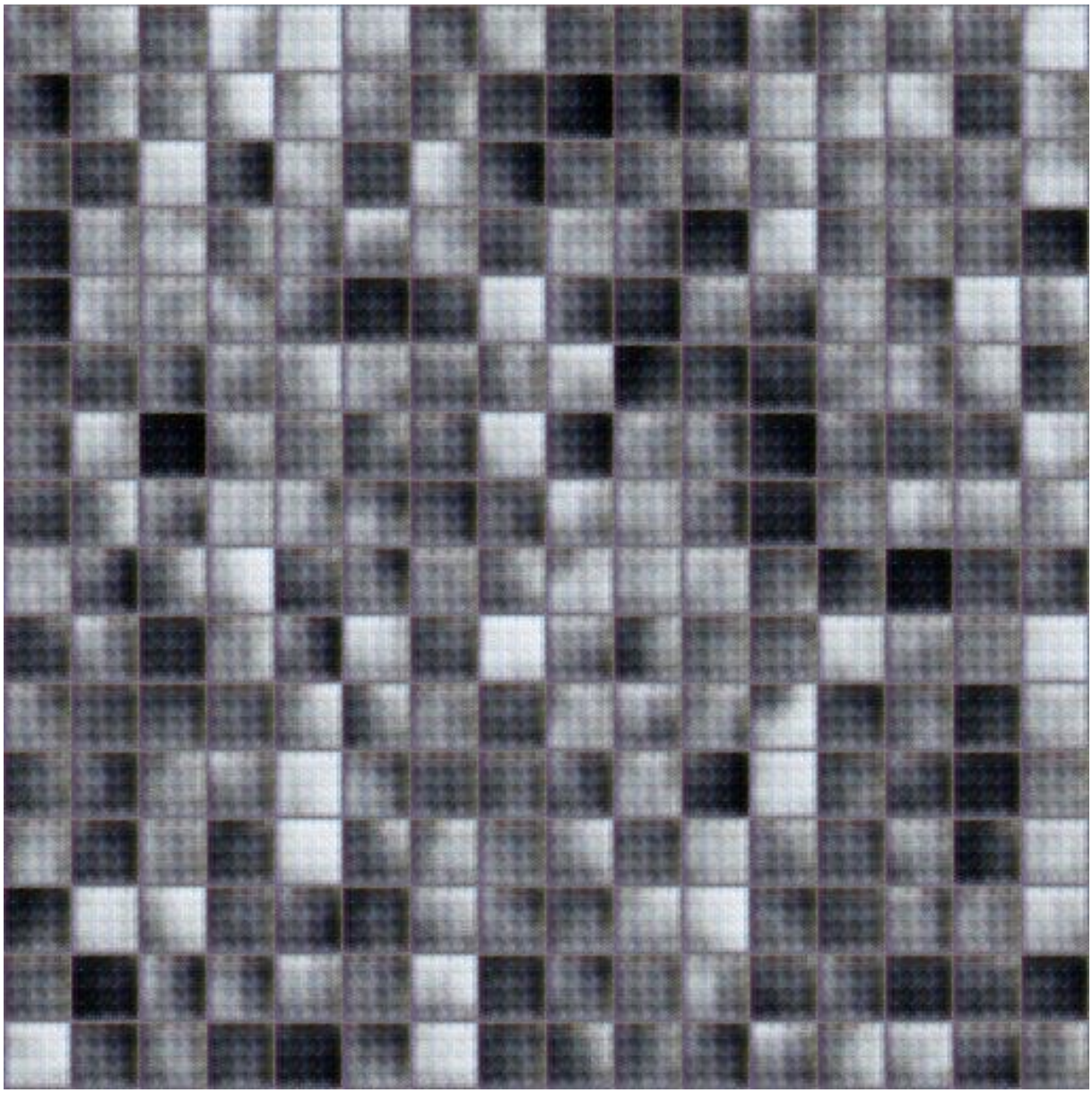

Figure 10: Iteration 300

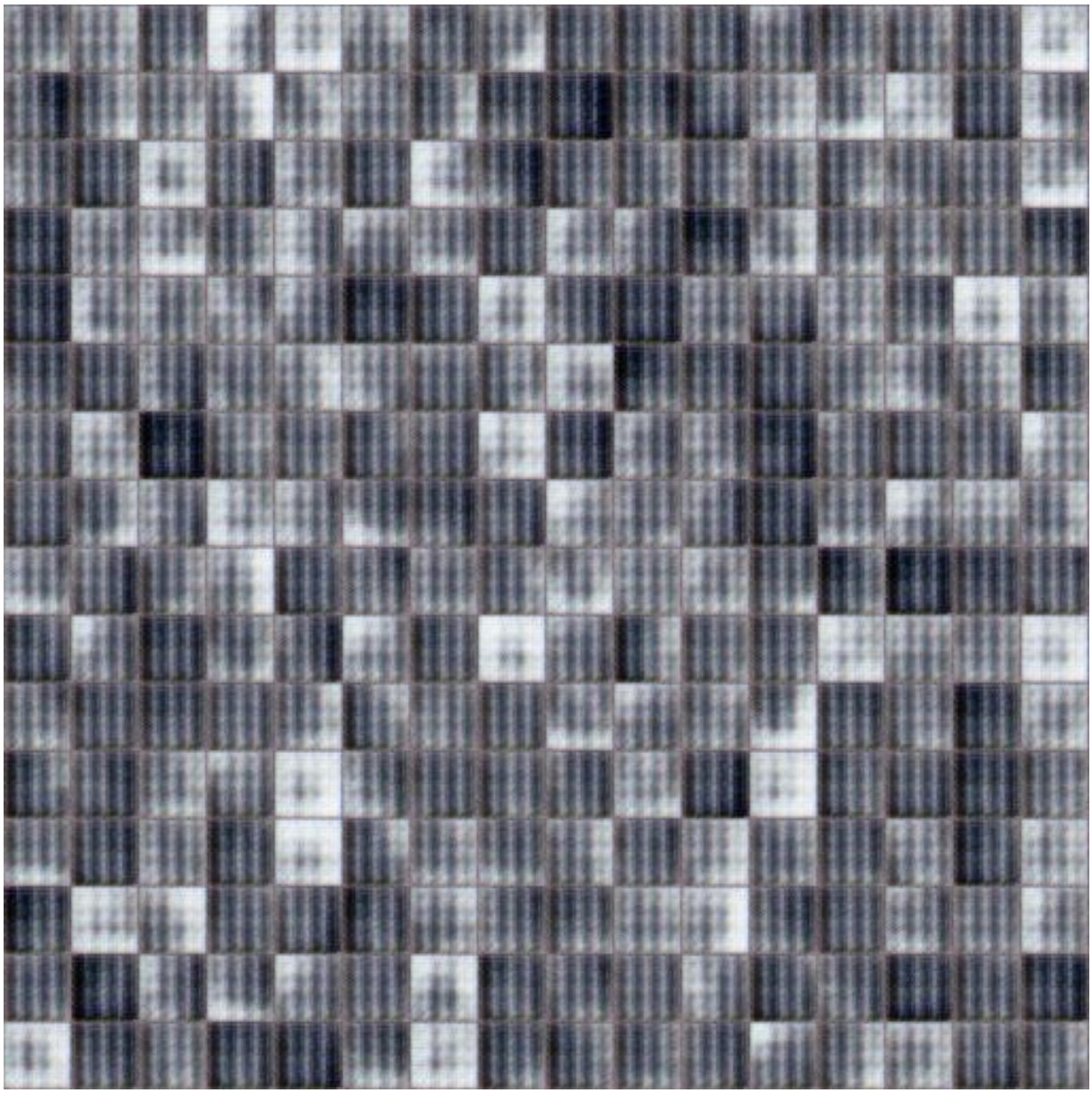

Figure 11: Iteration 400

Figure 12: Iteration 5000

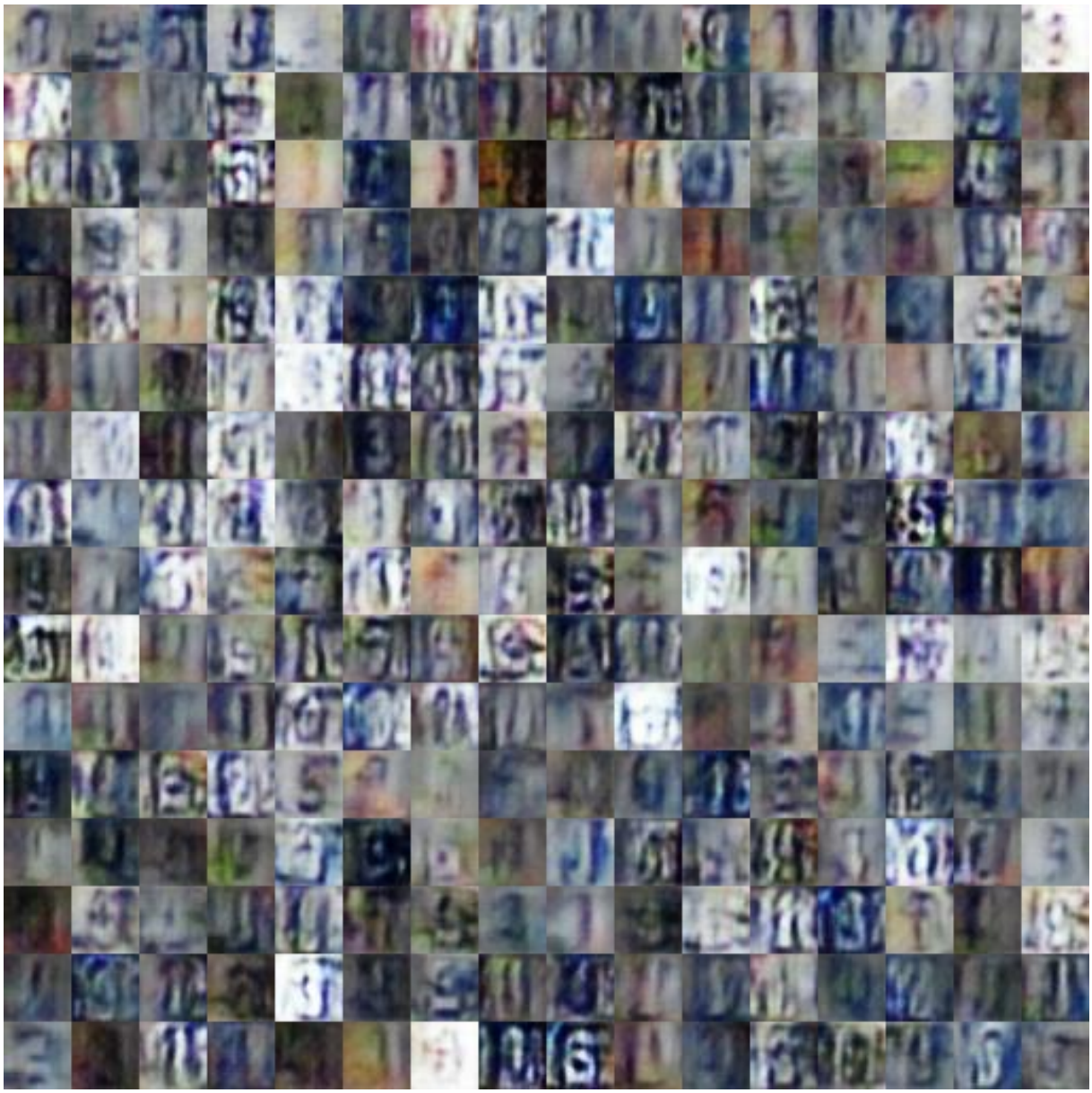

Figure 13: Iteration 10000

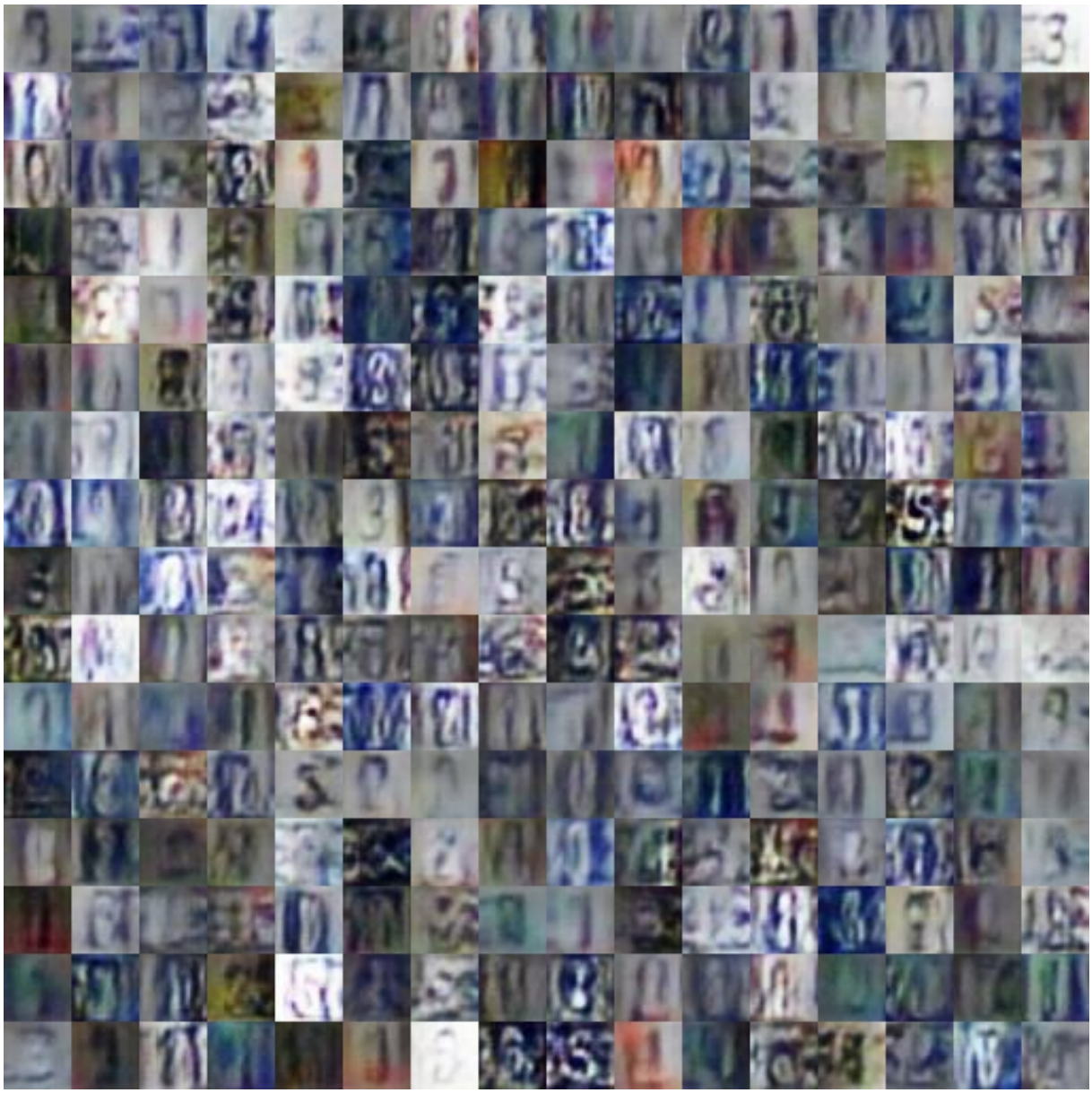

Figure 14: Iteration 15000

Figure 15: Iteration 20000

Figure 16: Iteration 25000

Figure 17: Iteration 30000

Figure 18: Iteration 35000

Figure 19: Iteration 40000

Figure 20: Iteration 45000

Figure 21: Iteration 50000

2. For this part I have used $epsilon = 1e1$ which is quite large but its results were interesting to me. Also, the diversity of colors in generated samples seems different to me which is interesting. I have plotted samples from

$$dimension = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 99]$$

for one sample.
Here are the results for one sample:

Figure 22: 21 different dimensions - top left is captured from dimension = 0 and bottom right is captured from dimension = 99

The interesting thing is that you can see it moves from something look like 1 to 2 then we can see the shifting in colors and it moves around something between 5, 6, and 8.

3. Here are the codes and results for part 1 and 2:

Figure 23: plotted samples with different alpha

```
alphas = np.linspace(0, 1, 11)

z0 = Variable(torch.randn(1, 100, 1, 1)).to(device)
z1 = Variable(torch.randn(1, 100, 1, 1)).to(device)

for alpha in alphas:
    z_alpha = (alpha * z0) + ((1 - alpha) * z1)

    x_alpha = generator(z_alpha)
    filename = '{}_interp1_{}.png'.format("generator", alpha)
    plot_sample(x_alpha, filename)
```

Figure 24: Code for interpolation$_1$

Figure 25: plotted samples with different alpha

```
alphas = np.linspace(0, 1, 11)

z0 = Variable(torch.randn(1, 100, 1, 1)).to(device)
z1 = Variable(torch.randn(1, 100, 1, 1)).to(device)
x0 = generator(z0)#.detach().cpu().numpy()
x1 = generator(z1)#.detach().cpu().numpy()



for alpha in alphas:
    x_hat_alpha = (alpha * x0) + ((1 - alpha) * x1)
    alpha = str(alpha).replace('.', '_')
    filename = '{}_interp2_{}.png'.format("generator", alpha)
    plot_sample(x_hat_alpha, filename)
```

Figure 26: Code for interpolation$_2$

Appendix

1. **Question 1 experiments:**

vae

April 29, 2020

Solution template for the question 1.6-1.7. This template consists of following steps. Except the step 2, you don't need to modify it to answer the questions. 1. Initialize libraries 2. **Insert the answers for the questions 1.1~1.5 below (this is the part you need to fill)** 3. Define data loaders 4. Define VAE network architecture 5. Initialize the model and optimizer 6. Train the model 7. Save the model 8. Load the model 9. Evaluate the model with importance sampling

Initialize libraries

[0]:
```
import math
from torchvision.datasets import utils
import torch.utils.data as data_utils
import torch
import os
import numpy as np
from torch import nn
from torch.nn.modules import upsampling
from torch.functional import F
from torch.optim import Adam
```

Insert **the answers for the questions 1.1~1.5 below**

[0]:
```
def log_likelihood_bernoulli(mu, target):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests␣
↪might fail.

    *** note. ***

    :param mu: (FloatTensor) - shape: (batch_size x input_size) - The mean of␣
↪Bernoulli random variables p(x=1).
    :param target: (FloatTensor) - shape: (batch_size x input_size) - Target␣
↪samples (binary values).
    :return: (FloatTensor) - shape: (batch_size,) - log-likelihood of target␣
↪samples on the Bernoulli random variables.
    """
    # init
    batch_size = mu.size(0)
    mu = mu.view(batch_size, -1)
    target = target.view(batch_size, -1)
```

```
    # log_likelihood_bernoulli
    ll = (target * torch.log(mu)) + ((1 - target) * torch.log(1 - mu))
    return ll.sum(dim=1)


def log_likelihood_normal(mu, logvar, z):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests
    might fail.

    *** note. ***

    :param mu: (FloatTensor) - shape: (batch_size x input_size) - The mean of
    Normal distributions.
    :param logvar: (FloatTensor) - shape: (batch_size x input_size) - The log
    variance of Normal distributions.
    :param z: (FloatTensor) - shape: (batch_size x input_size) - Target samples.
    :return: (FloatTensor) - shape: (batch_size,) - log probability of the
    sames on the given Normal distributions.
    """
    # init
    batch_size = mu.size(0)
    mu = mu.view(batch_size, -1)
    logvar = logvar.view(batch_size, -1)
    z = z.view(batch_size, -1)

    # log normal
    ll = -((logvar + np.log(2 * np.pi)) + ((z - mu) ** 2) / logvar.exp()) / 2
    return ll.sum(dim=1)


def log_mean_exp(y):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests
    might fail.

    *** note. ***

    :param y: (FloatTensor) - shape: (batch_size x sample_size) - Values to be
    evaluated for log_mean_exp. For example log proabablies
    :return: (FloatTensor) - shape: (batch_size,) - Output for log_mean_exp.
    """
    # init
    batch_size = y.size(0)
    sample_size = y.size(1)
```

```
    # log_mean_exp
    cmax, _ = y.max(dim=1)
    log_mean = (y - cmax.unsqueeze(1)).exp().mean(dim=1).log()
    result = log_mean + cmax
    return result


def kl_gaussian_gaussian_analytic(mu_q, logvar_q, mu_p, logvar_p):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests
 ↪might fail.

    *** note. ***

    :param mu_q: (FloatTensor) - shape: (batch_size x input_size) - The mean of
 ↪first distributions (Normal distributions).
    :param logvar_q: (FloatTensor) - shape: (batch_size x input_size) - The log
 ↪variance of first distributions (Normal distributions).
    :param mu_p: (FloatTensor) - shape: (batch_size x input_size) - The mean of
 ↪second distributions (Normal distributions).
    :param logvar_p: (FloatTensor) - shape: (batch_size x input_size) - The log
 ↪variance of second distributions (Normal distributions).
    :return: (FloatTensor) - shape: (batch_size,) - kl-divergence of KL(q||p).
    """
    # init
    batch_size = mu_q.size(0)
    mu_q = mu_q.view(batch_size, -1)
    logvar_q = logvar_q.view(batch_size, -1)
    mu_p = mu_p.view(batch_size, -1)
    logvar_p = logvar_p.view(batch_size, -1)

    # kld
    kld = 0.5 * (
        (logvar_q - logvar_p).exp()
        + ((mu_q - mu_p) ** 2) / logvar_p.exp()
        + (logvar_p - logvar_q)
        - 1
    )
    return kld.sum(dim=1)


def kl_gaussian_gaussian_mc(mu_q, logvar_q, mu_p, logvar_p, num_samples=1):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests
 ↪might fail.
```

```
    *** note. ***

    :param mu_q: (FloatTensor) - shape: (batch_size x input_size) - The mean of␣
↪first distributions (Normal distributions).
    :param logvar_q: (FloatTensor) - shape: (batch_size x input_size) - The log␣
↪variance of first distributions (Normal distributions).
    :param mu_p: (FloatTensor) - shape: (batch_size x input_size) - The mean of␣
↪second distributions (Normal distributions).
    :param logvar_p: (FloatTensor) - shape: (batch_size x input_size) - The log␣
↪variance of second distributions (Normal distributions).
    :param num_samples: (int) - shape: () - The number of sample for Monte␣
↪Carlo estimate for KL-divergence
    :return: (FloatTensor) - shape: (batch_size,) - kl-divergence of KL(q||p).
    """
    # init
    batch_size = mu_q.size(0)
    input_size = np.prod(mu_q.size()[1:])
    mu_q = (
        mu_q.view(batch_size, -1)
        .unsqueeze(1)
        .expand(batch_size, num_samples, input_size)
    )
    logvar_q = (
        logvar_q.view(batch_size, -1)
        .unsqueeze(1)
        .expand(batch_size, num_samples, input_size)
    )
    mu_p = (
        mu_p.view(batch_size, -1)
        .unsqueeze(1)
        .expand(batch_size, num_samples, input_size)
    )
    logvar_p = (
        logvar_p.view(batch_size, -1)
        .unsqueeze(1)
        .expand(batch_size, num_samples, input_size)
    )

    z = torch.normal(mean=mu_q, std=logvar_q.exp())
    # z_q = (z - mu_q) / logvar_q.exp()
    # z_p = (z - mu_p) / logvar_p.exp()
    # kld = (logvar_p - logvar_q) + 0.5 * (z_p ** 2 - z_q ** 2)
    # kld.mean(dim=1).sum(dim=1)
    kld = (
        log_likelihood_normal(mu_q, logvar_q, z)
        - log_likelihood_normal(mu_p, logvar_p, z)
    ) / num_samples
```

```
    return kld
```

Define data loaders

[0]:
```
def get_data_loader(dataset_location, batch_size):
    URL = "http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/"
    # start processing
    def lines_to_np_array(lines):
        return np.array([[int(i) for i in line.split()] for line in lines])
    splitdata = []
    for splitname in ["train", "valid", "test"]:
        filename = "binarized_mnist_%s.amat" % splitname
        filepath = os.path.join(dataset_location, filename)
        utils.download_url(URL + filename, dataset_location)
        with open(filepath) as f:
            lines = f.readlines()
        x = lines_to_np_array(lines).astype('float32')
        x = x.reshape(x.shape[0], 1, 28, 28)
        # pytorch data loader
        dataset = data_utils.TensorDataset(torch.from_numpy(x))
        dataset_loader = data_utils.DataLoader(x, batch_size=batch_size,␣
↪shuffle=splitname == "train")
        splitdata.append(dataset_loader)
    return splitdata
```

[0]:
```
train, valid, test = get_data_loader("binarized_mnist", 64)
```

```
Downloading http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/
binarized_mnist_train.amat to binarized_mnist/binarized_mnist_train.amat

HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))



Downloading http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/
binarized_mnist_valid.amat to binarized_mnist/binarized_mnist_valid.amat

HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))



Downloading http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/
binarized_mnist_test.amat to binarized_mnist/binarized_mnist_test.amat

HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

Define VAE network architecture

[0]:
```
class Encoder(nn.Module):
    def __init__(self, latent_size):
        super(Encoder, self).__init__()
        self.mlp = nn.Sequential(
            nn.Linear(784, 300),
            nn.ELU(),
            nn.Linear(300, 300),
            nn.ELU(),
            nn.Linear(300, 2 * latent_size),
        )

    def forward(self, x):
        batch_size = x.size(0)
        z_mean, z_logvar = self.mlp(x.view(batch_size, 784)).chunk(2, dim=-1)
        return z_mean, z_logvar

class Decoder(nn.Module):
    def __init__(self, latent_size):
        super(Decoder, self).__init__()
        self.mlp = nn.Sequential(
            nn.Linear(latent_size, 300),
            nn.ELU(),
            nn.Linear(300, 300),
            nn.ELU(),
            nn.Linear(300, 784),
        )

    def forward(self, z):
        return self.mlp(z) - 5.

class VAE(nn.Module):
    def __init__(self, latent_size):
        super(VAE, self).__init__()
        self.encode = Encoder(latent_size)
        self.decode = Decoder(latent_size)

    def forward(self, x):
        z_mean, z_logvar = self.encode(x)
        z_sample = z_mean + torch.exp(z_logvar / 2.) * torch.
 ↪randn_like(z_logvar)
        x_mean = self.decode(z_sample)
        return z_mean, z_logvar, x_mean

    def loss(self, x, z_mean, z_logvar, x_mean):
        ZERO = torch.zeros(z_mean.size())
        # kl = kl_gaussian_gaussian_mc(z_mean, z_logvar, ZERO, ZERO,␣
 ↪num_samples=1000).mean()
```

```
        kl = kl_gaussian_gaussian_analytic(z_mean, z_logvar, ZERO, ZERO).mean()
        recon_loss = -log_likelihood_bernoulli(
            torch.sigmoid(x_mean.view(x.size(0), -1)),
            x.view(x.size(0), -1),
        ).mean()
        return recon_loss + kl
```

Initialize a model and optimizer

[0]:
```
vae = VAE(100)
params = vae.parameters()
optimizer = Adam(params, lr=3e-4)
print(vae)
```

```
VAE(
  (encode): Encoder(
    (mlp): Sequential(
      (0): Linear(in_features=784, out_features=300, bias=True)
      (1): ELU(alpha=1.0)
      (2): Linear(in_features=300, out_features=300, bias=True)
      (3): ELU(alpha=1.0)
      (4): Linear(in_features=300, out_features=200, bias=True)
    )
  )
  (decode): Decoder(
    (mlp): Sequential(
      (0): Linear(in_features=100, out_features=300, bias=True)
      (1): ELU(alpha=1.0)
      (2): Linear(in_features=300, out_features=300, bias=True)
      (3): ELU(alpha=1.0)
      (4): Linear(in_features=300, out_features=784, bias=True)
    )
  )
)
```

Train the model

[0]:
```
for i in range(20):
    # train
    for x in train:
        optimizer.zero_grad()
        z_mean, z_logvar, x_mean = vae(x)
        loss = vae.loss(x, z_mean, z_logvar, x_mean)
        loss.backward()
        optimizer.step()

    # evaluate ELBO on the valid dataset
    with torch.no_grad():
        total_loss = 0.
        total_count = 0
```

```
        for x in valid:
            total_loss += vae.loss(x, *vae(x)) * x.size(0)
            total_count += x.size(0)
        print('-elbo: ', (total_loss / total_count).item())
```

```
-elbo:  163.41235351562 5
-elbo:  142.49197387695312
-elbo:  128.6122283935547
-elbo:  121.07291412353516
-elbo:  116.50720977783203
-elbo:  113.65373992919922
-elbo:  111.42896270751953
-elbo:  109.68109130859375
-elbo:  108.13333892822266
-elbo:  107.07254028320312
-elbo:  105.8725357055664
-elbo:  105.00170135498047
-elbo:  104.41787719726562
-elbo:  103.88336181640625
-elbo:  103.32199096679688
-elbo:  102.98141479492188
-elbo:  102.48633575439453
-elbo:  102.15198516845703
-elbo:  101.75672912597656
-elbo:  101.50662231445312
```

Save the model

[0]:
```
torch.save(vae, 'model.pt')
```

```
/usr/local/lib/python3.6/dist-packages/torch/serialization.py:360: UserWarning:
Couldn't retrieve source code for container of type VAE. It won't be checked for
correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
/usr/local/lib/python3.6/dist-packages/torch/serialization.py:360: UserWarning:
Couldn't retrieve source code for container of type Encoder. It won't be checked
for correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
/usr/local/lib/python3.6/dist-packages/torch/serialization.py:360: UserWarning:
Couldn't retrieve source code for container of type Decoder. It won't be checked
for correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
```

Load the model

[0]:
```
vae = torch.load('model.pt')
```

Evaluate the log $p_\theta(x)$ of the model on test by using importance sampling

[0]:
```
total_loss = 0.
total_count = 0
with torch.no_grad():
    #x = next(iter(test))
    for x in test:
        # init
        K = 200
        M = x.size(0)

        # Sample from the posterior
        z_mean, z_logvar = vae.encode(x)
        eps = torch.randn(z_mean.size(0), K, z_mean.size(1))
        z_samples = z_mean[:, None, :] + torch.exp(z_logvar / 2.)[:, None, :] *␣
↪eps # Broadcast the noise over the mean and variance

        # Decode samples
        z_samples_flat = z_samples.view(-1, z_samples.size(-1)) # Flatten out␣
↪the z samples
        x_mean_flat = vae.decode(z_samples_flat) # Push it through

        # Reshape images and posterior to evaluate probabilities
        x_flat = x[:, None].repeat(1, K, 1, 1, 1).reshape(M*K, -1)
        z_mean_flat = z_mean[:, None, :].expand_as(z_samples).reshape(M*K, -1)
        z_logvar_flat =  z_logvar[:, None, :].expand_as(z_samples).reshape(M*K,␣
↪-1)
        ZEROS = torch.zeros(z_mean_flat.size())

        # Calculate all the probabilities!
        log_p_x_z = log_likelihood_bernoulli(torch.sigmoid(x_mean_flat),␣
↪x_flat).view(M, K)
        log_q_z_x = log_likelihood_normal(z_mean_flat, z_logvar_flat,␣
↪z_samples_flat).view(M, K)
        log_p_z = log_likelihood_normal(ZEROS, ZEROS, z_samples_flat).view(M, K)

        # Recombine them.
        w = log_p_x_z + log_p_z - log_q_z_x
        log_p = log_mean_exp(w)

        # Accumulate
        total_loss += log_p.sum()
        total_count += M

print('log p(x):', (total_loss / total_count).item())
```

```
log p(x): -95.81806945800781
```

[0]:
```

```