CS 446 / ECE 449 — Homework 5

your NetID here

Version 1.0

Instructions.

- Homework is due **Tuesday, April 13, at noon CST**; no late homework accepted.
- Everyone must submit individually at gradescope under `hw5` and `hw5code`.
- The "written" submission at `hw5` **must be typed**, and submitted in any format gradescope accepts (to be safe, submit a PDF). You may use LaTeX, markdown, google docs, MS word, whatever you like; but it must be typed!
- When submitting at `hw5`, gradescope will ask you to mark out boxes around each of your answers; please do this precisely!
- Please make sure your NetID is clear and large on the first page of the homework.
- Your solution **must** be written in your own words. Please see the course webpage for full academic integrity information. Briefly, you may have high-level discussions with at most 3 classmates, whose NetIDs you should place on the first page of your solutions, and you should cite any external reference you use; despite all this, your solution must be written in your own words.
- We reserve the right to reduce the auto-graded score for `hw5code` if we detect funny business (e.g., your solution lacks any algorithm and hard-codes answers you obtained from someone else, or simply via trial-and-error with the autograder).
- When submitting to `hw5code`, only upload `hw5_vae.py` and `hw5_gan.py`. Additional files will be ignored.

1. Variational Auto-Encoders [Written]

We use VAEs to learn the distribution of the data x. Let z denote the unobserved latent variable. We refer to the approximated posterior $q_\phi(z|x)$ as the encoder and to the conditional distribution $p_\theta(x|z)$ as the decoder. Use these names to answer the following questions.

(a) We are interested in modeling data $x \in \{0,1\}^G$. Hence, we choose $p_\theta(x|z)$ to follow G independent Bernoulli distributions. Recall, a Bernoulli distribution has a probability density function of

$$P(k) = \begin{cases} 1-p & \text{if } k = 0 \\ p & \text{if } k = 1 \end{cases}.$$

Write down the explicit form for $p_\theta(x|z)$. Use $\hat{y}_j$ to denote the $j^{\text{th}} \in [1, G]$ dimension of the decoder's output.

(b) We further assume that $z \in \mathbb{R}^2$ and that $q_\phi(z|x)$ follows a multi-variate Gaussian distribution with an identity covariance matrix. What is the output dimension of the encoder and why?

(c) We want to maximize the log-likelihood $\log p_\theta(x)$. To this end we introduce a joint distribution $p_\theta(x, z)$ and reformulate the log-likelihood via

$$\log p_\theta(x) = \log \sum_z q_\phi(z|x) \frac{p_\theta(x,z)}{q_\phi(z|x)}.$$

Use Jensen's inequality to obtain a bound on the log-likelihood and divide the bound into two parts, one of which is the Kullback-Leibler divergence $\text{KL}(q_\phi(z|x), p(z))$.

(d) State at least two properties of the KL-divergence.

(e) Recall, the evidence lower bound (ELBO) of the log likelihood, $\log p_\theta(x)$, is

$$\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - \text{KL}(q_\phi(z|x)||p(z)). \tag{1}$$

We can also write the ELBO as

$$\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z) + \log p(z) - \log(q_\phi(z|x))]. \tag{2}$$

Practically, will training a VAE using the formulation in Eq. 1 be the same as the one in Eq. 2? If not, why use one formulation over another?

(f) Observe that the ELBO in Eq. 1 works for any q_ϕ distribution. Is it a good idea to choose $q_\phi(z|x) := \mathcal{N}(0, I)$? In other words, why is an encoder necessary?

(g) Let

$$q_\phi(z|x) = \frac{1}{\sqrt{2\pi\sigma_\phi^2}} \exp\left(-\frac{1}{2\sigma_\phi^2}(z - \mu_\phi)^2\right).$$

What is the value for the KL-divergence $\text{KL}(q_\phi(z|x), q_\phi(z|x))$ and why?

(h) Further, let

$$p(z) = \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left(-\frac{1}{2\sigma_p^2}(z - \mu_p)^2\right).$$

Note the difference of the means for $p(z)$ and $q_\phi(z|x)$ while their standard deviations are identical. Assume that $\sigma = \sigma_\phi = \sigma_p$. What is the value for the KL-divergence $\text{KL}(q_\phi(z|x), p(z))$ in terms of μ_p, μ_ϕ and σ?

(i) Now, let $q_\phi(z|x)$ and $p(z)$ be arbitrary probability distributions. We want to find that $q_\phi(z|x)$ which maximizes

$$\sum_z q_\phi(z|x) \log p_\theta(x|z) - \text{KL}(q_\phi(z|x), p(z))$$

subject to $\sum_z q_\phi(z|x) = 1$. Ignore the non-negativity constraints. State the Lagrangian and compute its stationary point, i.e., solve for $q_\phi(z|x)$ which depends on $p_\theta(x|z)$ and $p(z)$. Make sure to get rid of the Lagrange multiplier.

(j) Which of the following terms should $q_\phi(z|x)$ be equal to: (1) $p(z)$; (2) $p_\theta(x|z)$; (3) $p_\theta(z|x)$; (4) $p_\theta(x,z)$.

Solution.

(a) $p_\theta(x|z) = \prod_{j=1}^{G} \hat{y}_j^{x_j} \cdot (1-\hat{y}_j)^{1-x_j}$

(b) The output dimension is 2, since the latent dimenion is 2 and we only need to learn the mean of the distribution.

(c)
$$\log p_\theta(x) \geq \sum_z q_\phi(z|x) \log \frac{p_\theta(x,z)}{q_\phi(z|x)} = \sum_z q_\phi(z|x) \log p_\theta(x|z) - \mathrm{KL}(q_\phi(z|x), p(z))$$

(d) non-negative, zero if $q_\phi(z|x) = p(z)$

(e) (1) In practice the two formulation will not be the same. This is because the training algorithm for VAE requires sampling z. In the first formulation, KL uses an analytic form and do not estimate the KL-term from samples. (2) One might choose to use the second formulation when the analytic form of KL do not exist or is difficult to compute.

(f) No. Ideally, q_ϕ should approximate $p_\theta(z|x)$ for the ELBO to be tight. Otherwise, samples from q_ϕ may result in $p_\theta(x|z) \approx 0$, which may require many samples to get a gradient signal.

(g) 0, KL-divergence between two identical distributions

(h)
$$\mathrm{KL}(q_\phi(z|x), p(z)) = \frac{1}{2\sigma^2}(\mu_\phi - \mu_p)^2$$

(i)
$$\sum_z q_\phi(z|x) \log p_\theta(x|z) - \sum_z q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z)} + \lambda \left(\sum_z q_\phi(z|x) - 1 \right)$$
$$q_\phi(z|x) = \frac{p_\theta(x|z)p(z)}{\sum_{\hat{z}} p_\theta(x|\hat{z})p(\hat{z})}$$

(j)
$$p_\theta(z|x) = \frac{p_\theta(x|z)p(z)}{\sum_{\hat{z}} p_\theta(x|\hat{z})p(\hat{z})}$$

2. Variational Auto-Encoders [Coding]

In this assignment, you will implement a Variational Autoencoder and train it on MNIST digits. Each datapoint x in the MNIST dataset is a 28×28 grayscale image (i.e., pixel values are between 0 and 1) of a handwritten digit in $\{0, \ldots, 9\}$, and a label indicating which number. The prior over each digit's latent representation z is a multivariate standard normal distribution, i.e., $z \sim \mathcal{N}(0, I)$. For all questions, we set the dimension of the latent space D_z to 2. Given the latent representation z for an image, the distribution over all 784 pixels in the image is given by a product of independent Bernoullis, whose characteristc probabilities are given by the output of a neural network $f_\theta(z)$ (the decoder):

$$p_\theta(x|z) = \prod_{d=1}^{784} \text{Ber}(x_d | f_\theta(z)). \tag{3}$$

Relevant files: *HW5_vae.py*, *HW5_utils.py*.

(a) **Decoder Architecture**. Given a latent representation z, the decoder produces a 784-dimensional vector representing the Bernoulli distribution characteristic probability, i.e., the probability for every pixel in the image being labeled 1. Define the decoder parameters in the method *__init__* of the *Decoder* class and implement the corresponding *forward* function. The decoder architecture is a multi-layer perceptron (i.e., a fully-connected neural network), with two hidden layers, followed each by a non linearity: *tanh* after the first layer and *sigmoid* after the second layer. The hidden dimension is set to 500 units.

(b) **Distributions**.

i. Implement the method *logpdf_diagonal_gaussian*, that given a latent representation z, a mean μ and the variance σ^2 outputs the log-likelihood of the normal distribution $\mathcal{N}(\mu, \sigma^2 I)$.

ii. Implement a function *logpdf_bernoulli*, that given a sample x, a probability p, outputs the log-likelihood of a Bernoulli distribution.

iii. Implement the function *sample_diagonal_gaussian* which uses the reparametrization trick to sample z from Diagonal Gaussian $z \sim \mathcal{N}(\mu, \sigma^2 I)$.

iv. Implement the function *sample_Bernoulli* which samples a configuration x from a Bernoulli distribution characterized by a probability p.

(c) **Variational Objective**. Complete the function *elbo* with the ELBO loss implementation corresponding to Eq. 2.

(d) **Training**. Train the model for 200 epochs. **Hint:** Run the *main* function and make sure the number of epochs is set-up correctly in *parse_args*.

(e) **Visualization**.

i. **Samples from the generative model**. Complete the method *visualize_data_space* following the instructions:

- Sample a z from the prior $p(z)$. Use *sample_diagonal_gaussian*.
- Use the generative model to parameterize a Bernoulli distribution over x given z. Use *self.decoder* and *array_to_image*. Plot this distribution $p(x|z)$.
- Sample x from the distribution $p(x|z)$. Plot this sample.
- Repeat the steps above for 10 samples z from the prior. Concatenate all your plots into one 10×2 figure where the first column is the distribution over x and the second column is a sample from this distribution. Each row will be a new sample from the prior. Hint: use the function *concat_images*.
- Attach the figure to your report.

ii. **Latent space visualization**. Produce a scatter plot in the latent space, where each point in the plot represents a different image in the training set. Complete the method *visualize_latent_space* following the instructions:

- Encode each image in the training set. Use *self.encoder*.
- Plot the mean vector μ of $q_\phi(z|x)$ in the 2D latent space with a scatter plot. Make sure to color each point according to the class label (0 to 9).
- Attach the scatter plot to your report.

iii. **Interpolation between two classes**. Complete the method *visualize_inter_class_interpolation* following the instructions:

- Sample 3 pairs of data points (*self.train_images*) with different classes (*self.train_labels*).
- Encode the data in each pair, and take the mean vectors. Note that the encoder procuces a meam vector and a variance one.
- Interpolate between these mean vectors. We denote the output by z_α, with $\alpha \in [0, 1]$ and the interpolation step being 0.1. Hint: use the function *interpolate_mu*.
- Along the interpolation, plot the distributions $p(x|z_\alpha)$ in the same figure
- Use *concat_images* to concatenate these plots into one figure.
- Attach the plot to your report.

Solution.

(a)
```
class Decoder(nn.Module):
    def __init__(self,latent_dimension,hidden_units,data_dimension):
        super(Decoder, self).__init__()
        self.fc1 = nn.Linear(latent_dimension, hidden_units)
        self.fc2 = nn.Linear(hidden_units, data_dimension)

    def forward(self, z):
        hidden = torch.tanh(self.fc1(z))
        y = torch.sigmoid(self.fc2(hidden))
        return y
```

(b) i.
```
def logpdf_diagonal_gaussian(z, mu, sigma_square):
    log_det_sigma = torch.sum(torch.log(sigma_square),-1)
    z_mu_diff = z-mu
    sigma_inverse = 1/sigma_square
    k = z.shape[-1]
    s = z_mu_diff.shape[-1]
    logprob = -0.5*(log_det_sigma +
            torch.bmm(z_mu_diff.view(-1,1,s),
            (sigma_inverse * z_mu_diff).view(-1,s,1)).view(-1)
             + k * np.log(np.pi*2))
    return logprob
```

ii.
```
def logpdf_bernoulli(x, p):
    logprob=torch.sum(x*torch.log(p)+(1-x)*torch.log(1-p),-1)
    return logprob
```

iii.
```
def sample_diagonal_gaussian(mu, sigma_square):
    sample = mu + torch.sqrt(sigma_square)*
            torch.randn(sigma_square.shape)
    return sample
```

iv.
```
def sample_Bernoulli(p):
```

```
            x = torch.rand(p.shape)
            x = (x<p).float()
            return x
```

(c)

(d)
```
def elbo_loss(self, sampled_z, mu, sigma_square, x, p):
        # log_p_z(z) log probability of z under prior
        z_mu=torch.FloatTensor([0]*self.latent_dimension)
        z_sigma=torch.FloatTensor([1]*self.latent_dimension)
        log_p_z=self.logpdf_diagonal_gaussian(sampled_z, z_mu,z_sigma)

        # log_p(x|z) - conditional probability of data given latents.
        log_p = self.logpdf_bernoulli(x, p)

        # TODO: implement the ELBO loss using log_q, log_p_z and log_p
        elbo = torch.mean(-log_q + log_p_z + log_p)
        return elbo
```

i.
```
def visualize_data_space(self):
        # TODO: Sample 10 z from prior
        z_mu = torch.FloatTensor([0]*self.latent_dimension)
        z_mu = z_mu.expand(10,-1)

        z_sigma = torch.FloatTensor([1]*self.latent_dimension)
        z_sigma = z_sigma.expand(10,-1)

        sampled_z = self.sample_diagonal_gaussian(z_mu, z_sigma)

        # TODO: For each z, plot p(x|z)
        ps = []
        p = self.decoder(sampled_z)
        for i in range(10):
                ps.append(array_to_image(p[i,:].detach().numpy()))

        # TODO: Sample x from p(x|z)
        xs = []
        x = self.sample_Bernoulli(p)

        for i in range(10):
                xs.append(array_to_image(x[i,:].detach().numpy()))

        # TODO: Concatenate plots into a figure
        concated_image=concat_images(ps+xs,10,self.latent_dimension)

        # TODO: Save the generated figure
        fig = plt.figure(figsize = (15,15))
        plt.axis('off')
        plt.imshow(concated_image, cmap='Greys')
        plt.savefig("./vis_data_space.pdf")
        plt.clf()
```

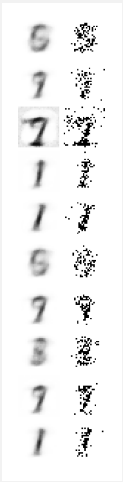


Figure 1: Data space visualization

ii.
```
def visualize_latent_space(self):
        # TODO: Encode the training data
        mu, sigma_square = self.encoder(self.train_images)

        # TODO: Take the mean vector of each encoding
        mean = mu.detach().numpy()

        # TODO: Plot these mean vectors in the latent space
        # Colour each point depending on the class label
         colours = ["Red", "Yellow", "Olive", "Lime", "Green",
                 "Aqua","Blue","Navy","Fuchsia", "Purple"]

         fig = plt.figure(figsize = (10,10))
         for i in range(10):
                 sample_index = np.nonzero(self.train_labels.numpy()
                 sample_index = sample_index[:,i])[0]

                 cur_digit_means = mean[sample_index]
                 plt.scatter(cur_digit_means[:,0],
                         cur_digit_means[:,1],c=colours[i])

         # TODO: Save the generated figure
         plt.savefig("./vis_latent_space.pdf")
         plt.clf()
```

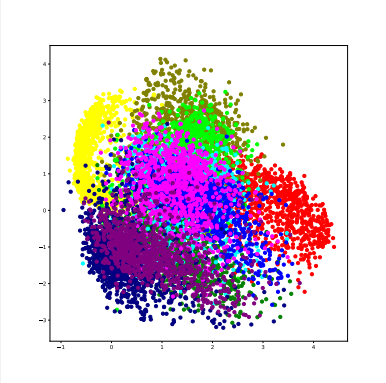


Figure 2: Latent space visualization

iii.
```
def visualize_inter_class_interpolation(self):
        # TODO: Sample 3 pairs of data with different classes
        pairs = []
        while len(pairs) < 3:
                i1 = int(np.random.rand() * 10000)
                i2 = int(np.random.rand() * 10000)
                check = self.train_labels[i1].equal(self.train_labels[i2])
                if not check:
                        pairs.append([self.train_images[i1],
                                self.train_images[i2]])

        # TODO: Encode the data in each pair,
        # and store the mean vectors
        pairs_mus = []
        for p in pairs:
                mu1, sigma_square1 = self.encoder(p[0])
                mu2, sigma_square2 = self.encoder(p[1])
                pairs_mus.append([mu1,mu2])

        # TODO: Encode the data in each pair,
        # and store the mean vectors
        pairs_mus = []
        for p in pairs:
                mu1, sigma_square1 = self.encoder(p[0])
                mu2, sigma_square2 = self.encoder(p[1])
                pairs_mus.append([mu1,mu2])

        # TODO: Along the interpolation, plot the distributions
        # p(x|z_alphas)
        plots = []
        for z in z_alphas:
                p = self.decoder(z)
                plots.append(array_to_image(p.detach().numpy()))

         # Concatenate these plots into one figure
         final_plot = concat_images(plots, 3,11)
         fig = plt.figure(figsize = (10,10))
```

```
plt.axis('off')
plt.imshow(final_plot, cmap='Greys')
plt.savefig("./vis_class_interolation.pdf")
plt.clf()
```

Figure 3: Inter-class interpolation

3. Generative Adversarial Networks [Written]

Here we discuss distribution-comparison-related problems in Generative Adversarial Networks (GANs).

(a) What is the cost function for classical GANs? Use $D_\omega(x)$ as the discriminator and $G_\theta(x)$ as the generator.

(b) Assume arbitrary capacity for both discriminator and generator. In this case we refer to the discriminator using $D(x)$, and denote the distribution on the data domain induced by the generator via $p_G(x)$. State an equivalent problem to the one asked for in part (a), by using $p_G(x)$.

(c) Assuming arbitrary capacity, derive the optimal discriminator $D^*(x)$ in terms of $p_{\text{data}}(x)$ and $p_G(x)$.
Hint: you can think of fixing generator $G(\cdot)$ to find the optimal value for discriminator $D(\cdot)$.

(d) Assume arbitrary capacity and an optimal discriminator $D^*(x)$ from (c), show that the optimal generator $G^*(x)$ generates the distribution $p^*_G = p_{\text{data}}$, where $p_{\text{data}}(x)$ is the data distribution.
Hint: you may need the Jensen-Shannon divergence:

$$\text{JSD}(p_{\text{data}}, p_G) = \frac{1}{2}\text{KL}(p_{\text{data}}, M) + \frac{1}{2}\text{KL}(p_G, M),$$

where $M = \frac{1}{2}(p_{\text{data}} + p_G)$.

(e) More recently, researchers have proposed to use the Wasserstein distance instead of divergences to train the models since the KL divergence often fails to give meaningful information for training. Consider three distributions, $\mathbb{P}_1 \sim U[0,1]$, $\mathbb{P}_2 \sim U[0.5, 1.5]$, and $\mathbb{P}_3 \sim U[1,2]$, where $U[a,b]$ is uniform distribution over $[a,b]$. Calculate $\text{KL}(\mathbb{P}_1, \mathbb{P}_2)$, $\text{KL}(\mathbb{P}_1, \mathbb{P}_3)$, $\mathbb{W}_1(\mathbb{P}_1, \mathbb{P}_2)$, and $\mathbb{W}(\mathbb{P}_1, \mathbb{P}_3)$, where $\mathbb{W}_1(\cdot,\cdot)$ is the Wasserstein-1 distance between two distributions.
Hint: this subproblem requires no *real* mathematical computation. What you need to do is to understand the intuitive meaning of KL-divergence and Wasserstein distance. You may find wiki of *Earth mover's distance* and *Wasserstein metric* useful.

Solution.

(a)

$$\min_\theta \max_\omega \sum_x \log D_\omega(x) + \sum_z \log\left(1 - D_\omega(G_\theta(z))\right)$$

(b)

$$\min_{p_G} \max_D \int_x p_{\text{data}}(x) \log D(x) dx + \int_x p_G(x) \log(1 - D(x)) dx$$

(c) We have:

$$\begin{aligned} L(D,G) &= \int_x p_{\text{data}}(x) \log D(x) dx + \int_x p_G(x) \log(1 - D(x)) dx \\ &= \int_x \left\{ p_{\text{data}}(x) \log D(x) + p_G(x) \log(1 - D(x)) \right\} dx \end{aligned}$$

When fixing $G(\cdot)$, we can take derivative of $p_{\text{data}}(x) \log D(x) + p_G(x) \log(1 - D(x))$ and obtain the critical point as

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}.$$

Note, this $D^*(x)$ achieves maximum value of $L(D,G)$ for every fixed G.

(d)

$$\begin{aligned}
&- \int_x p_{\text{data}}(x) \log D^*(x) + p_G(x) \log(1 - D^*(x)) dx \\
=& - \int_x p_{\text{data}}(x) \log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)} + p_G(x) \frac{p_G(x)}{p_{\text{data}}(x) + p_G(x)} dx \\
=& - 2\text{JSD}(p_{\text{data}}, p_G) + \log(4)
\end{aligned}$$

Hence, optimally, $p_{\text{data}} = p_G$.

(e) Since the distributions does not have identical support domain,

$$\begin{aligned}
\text{KL}(\mathbb{P}_1, \mathbb{P}_2) &= \infty \\
\text{KL}(\mathbb{P}_1, \mathbb{P}_3) &= \infty
\end{aligned}$$

For the Wasserstein distance, an optimal transport map between the 1-d Uniform distributions is simply a horizontal shift of appropriate size:

$$\begin{aligned}
\mathbb{W}(\mathbb{P}_1, \mathbb{P}_2) &= 0.5 \\
\mathbb{W}(\mathbb{P}_1, \mathbb{P}_3) &= 1.0
\end{aligned}$$

4. Generative Adversarial Networks [Coding]

In this problem, you need to implement a Generative Adversarial Network and train it on MNIST digits.

Table 1: **Discriminator Architecture**

Layer No.	*Layer Type*	*Kernel Size*	*Stride*	*Padding*	*Output Channels*
1	conv2d	3	1	1	2
2	ReLU	-	-	-	2
3	MaxPool	2	2	-	2
4	conv2d	3	1	1	4
5	ReLU	-	-	-	4
6	MaxPool	2	2	-	4
7	conv2d	3	1	0	8
8	ReLU	-	-	-	8
9	Linear	-	-	-	1

Table 2: **Generator Architecture**

Layer No.	*Layer Type*	*Kernel Size*	*Stride*	*Padding*	*Bias*	*Output Channels*
1	Linear	-	-	-	✓	1568
2	LeakyReLU(0.2)	-	-	-	-	1568
3	Upsample(scale=2)	-	-	-	✗	32
4	conv2d	3	1	1	✓	16
5	LeakyReLU(0.2)	-	-	-	-	16
6	Upsample(scale=2)	-	-	-	✗	16
7	conv2d	3	1	1	✓	8
8	LeakyReLU(0.2)	-	-	-	-	8
9	conv2d	3	1	1	✓	1
10	sigmoid	-	-	-	-	1

(a) Implement a discriminator `DNet` in `hw5_gan.py` with architecture in Tab. 1. Layers contain bias if corresponding `torch` function has an option for adding one.

Remark 1: From layer 8 to layer 9, you need to flat each data entry from a matrix to a vector.

(b) Implement a generator `GNet` in `hw5_gan.py` with architecture in Tab. 2.

Remark 2: From layer 2 to layer 3, you need to reshape each data to size $(32, 7, 7)$ in the format of *CHW*. Note, $1568 = 32 \times 7 \times 7$.

Remark 3: For (a) and (b), please define layers in `__init__` with **exactly the same** order as they appear in Tab. 1 and Tab. 2.

Remark 4: We have listed **all** layers for discriminator and generator. No need to add any extra components.

(c) Implement the weight initialization function `_weight_init` in `DNet` and `GNet`: use `kaiming_uniform` for weights and 0 for the bias if the layer contains bias.

Hint: to iterate over all layers an `nn.Module` has, you may find `self.children()` useful. See `children()` function explained in `https://pytorch.org/docs/stable/generated/torch.nn.Module.html`.

(d) Implement the loss function for the discriminator: `_get_loss_d` of `GAN` class in `hw5_gan.py`.

Hint: you may find `torch.nn.BCEWithLogitsLoss` useful.

(e) Implement the loss function for the generator: `_get_loss_g` of `GAN` class in `hw5_gan.py`.
Hint: you may find `torch.nn.BCEWithLogitsLoss` useful.

(f) Attach generated images after training.
Remark 5: the provided code default saves images during training. You can choose three of the saved ones and indicate the corresponding epochs.
Remark 6: with default training settings, you should obtain reasonable generated images after around 30 epochs.

Solution.

(a)
```
class DNet(nn.Module):
    """This is discriminator network."""

    def __init__(self):
        super(DNet, self).__init__()
        self.conv1 = nn.Conv2d(
            1, 2, kernel_size=3, stride=1, padding=1
        )  # out: 28 -> 14
        self.conv2 = nn.Conv2d(2, 4, kernel_size=3, stride=1, padding=1)
# out: 14 -> 7
        self.conv3 = nn.Conv2d(4, 8, kernel_size=3, stride=1, padding=0)
# out: 5 -> 5
        self.fc = nn.Linear(5 * 5 * 8, 1)

        self._weight_init()

    def _weight_init(self):
        for m in self.children():
            if isinstance(m, torch.nn.Linear) or isinstance(m, torch.nn.Conv2d):
                torch.nn.init.kaiming_uniform_(m.weight.data)
                if m.bias is not None:
                    m.bias.data.fill_(0)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv3(x))
        x = x.view(-1, 5 * 5 * 8)
        return self.fc(x)
```

(b)
```
class GNet(nn.Module):
    """This is generator network."""

    def __init__(self, zdim):
        super(GNet, self).__init__()
        self.first_dim = 32
        self.fc1 = nn.Linear(zdim, 7 * 7 * self.first_dim)

        self.up1 = nn.Upsample(scale_factor=2)
        self.conv1 = nn.Conv2d(self.first_dim, 16, 3, stride=1, padding=1)
```

```
            self.up2 = nn.Upsample(scale_factor=2)
            self.conv2 = nn.Conv2d(16, 8, 3, stride=1, padding=1)

            self.conv3 = nn.Conv2d(8, 1, 3, stride=1, padding=1)

            self._weight_init()

        def _weight_init(self):
            for m in self.children():
                if isinstance(m, torch.nn.Linear) or isinstance(m, torch.nn.Conv2d):
                    torch.nn.init.kaiming_uniform_(m.weight.data)
                    if m.bias is not None:
                        m.bias.data.fill_(0)

        def forward(self, z):
            z = torch.nn.LeakyReLU(0.2)(self.fc1(z))
            z = z.view(z.size()[0], self.first_dim, 7, 7)
            z = torch.nn.LeakyReLU(0.2)(self.conv1(self.up1(z)))
            z = torch.nn.LeakyReLU(0.2)(self.conv2(self.up2(z)))
            z = torch.sigmoid(self.conv3(z))
            return z
```

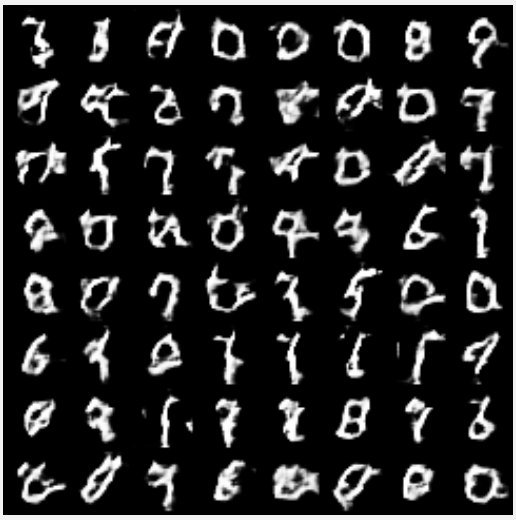
(c) See `_weight_init` in (a) and (b).

(d)
```
        def __init__(self, zdim=64):
            torch.manual_seed(2)
            self._dev = torch.device("cuda" if torch.cuda.is_available() else "cpu")

            self._zdim = zdim
            self.disc = DNet().to(self._dev)
            self.gen = GNet(self._zdim).to(self._dev)

        def _get_loss_d(self, batch_size, batch_data, z):
            """This function computes loss for discriminator.

            Parameters
            ----------
                batch_size: #data per batch.
                batch_data: data from dataset.
                z: random latent variable.
            """
            target_d = (
                torch.cat((torch.ones(batch_size), torch.zeros(batch_size)), dim=0)
                .to(self._dev)
                .view(-1, 1)
            )
            xhat = self.gen(z)
            logit = self.disc(torch.cat((batch_data, xhat.detach()), 0))
            criterion = nn.BCEWithLogitsLoss()
            loss_d = criterion(logit, target_d)
```

```
        return loss_d

    def _get_loss_g(self, batch_size, z):
        """This function computes loss for generator.

        Parameters
        ----------
            batch_size: #data per batch.
            z: random latent variable.
        """
        target_g = torch.ones(batch_size).to(self._dev).view(-1, 1)
        xhat = self.gen(z)
        logit = self.disc(xhat)
        criterion = nn.BCEWithLogitsLoss()
        loss_g = criterion(logit, target_g)
        return loss_g

    def train(self, iter_d=1, iter_g=1, n_epochs=100, batch_size=256, lr=0.0002):

        # first download
        f_name = "train-images-idx3-ubyte.gz"
        download(BASE_URL + f_name, f_name)

        print("Processing dataset ...")
        train_data = GANDataset(
            f"./data/{f_name}",
            self._dev,
            transform=transforms.Compose([transforms.Normalize((0.0,), (255.0,))])
        )
        print(f"... done. Total {len(train_data)} data entries.")

        train_loader = DataLoader(
            train_data,
            batch_size=batch_size,
            shuffle=True,
            num_workers=0,
            drop_last=True,
        )

        dopt = optim.Adam(self.disc.parameters(), lr=lr, weight_decay=0.0)
        dopt.zero_grad()
        gopt = optim.Adam(self.gen.parameters(), lr=lr, weight_decay=0.0)
        gopt.zero_grad()

        for epoch in tqdm(range(n_epochs)):
            for batch_idx, data in tqdm(
                enumerate(train_loader), total=len(train_loader)
            ):

                z = 2 * torch.rand(data.size()[0], self._zdim, device=self._dev) -

                if batch_idx == 0 and epoch == 0:
```

```
                    plt.imshow(data[0, 0, :, :].detach().cpu().numpy())
                    plt.savefig("goal.pdf")

                if batch_idx == 0 and epoch % 10 == 0:
                    with torch.no_grad():
                        tmpimg = self.gen(z)[0:64, :, :, :].detach().cpu()
                    save_image(
                        tmpimg, "test_{0}.png".format(epoch), nrow=8, normalize=T
                    )

                dopt.zero_grad()
                for k in range(iter_d):
                    loss_d = self._get_loss_d(batch_size, data, z)
                    # print("E: %d; B: %d; DLoss: %f" % (epoch, batch_idx, loss_d
                    loss_d.backward()
                    dopt.step()
                    dopt.zero_grad()

                gopt.zero_grad()
                for k in range(iter_g):
                    loss_g = self._get_loss_g(batch_size, z)
                    loss_g.backward()
                    # print("E: %d; B: %d; GLoss: %f" % (epoch, batch_idx, loss_g
                    gopt.step()
                    gopt.zero_grad()

            print(f"E: {epoch}; DLoss: {loss_d.item()}; GLoss: {loss_g.item()}")
```

(a) Epoch 10.



(b) Epoch 50.

(c) Epoch 90.

Figure 4: Visualization.

(e)