XCS236 Problem Set 3

Due Sunday, July 6 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs236-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit **all files indicated in the question** to the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into the `src/submission/` directory. When editing files in `src/submission/`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files outside the `src/submission/` directory.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```


Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

Instructor Note

PS3 is a fair mixture of written and coding questions. Fortunately, training these models will take substantially less time than PS2. However, it is also in your best interest to make sure that your code is properly vectorized so that the models run faster. If you plan on training the models on GPU, make sure your code can also run on CPU since this we will be running the autograder on CPU.

Here are the expected train times for each part of the assignment:

1. 5 minutes for `MAF` flow model
2. 5 minutes for `GAN` with nonsaturating loss
3. 10 minutes for `CGAN` with nonsaturating loss
4. 5 minutes for `GAN` with wasserstein gp loss

The following is a checklist of various functions you need to implement in the submission, in chronological order:

1. `forward` in `src/submission/flow_network.py`
2. `inverse` in `src/submission/flow_network.py`
3. `log_probs` in `src/submission/flow_network.py`
4. `loss_nonsaturating_g` in `src/submission/gan.py`
5. `loss_nonsaturating_d` in `src/submission/gan.py`
6. `conditional_loss_nonsaturating_g` in `src/submission/gan.py`
7. `conditional_loss_nonsaturating_d` in `src/submission/gan.py`
8. `loss_wasserstein_gp_g` in `src/submission/gan.py`
9. `loss_wasserstein_gp_d` in `src/submission/gan.py`

You will submit the entire `submission` directory. To do so run the command below from the `src` directory

```
zip -r submission.zip submission
```

or you can choose to zip the folder up manually.

Your submission zip should contain:

1. `src/submission/__init__.py`
2. `src/submission/flow_network.py`
3. `src/submission/gan.py`

1 Flow models

In this problem, we will implement a Masked Autoregressive Flow (MAF) model on the Moons dataset, where we define $p_{\text{data}}(\boldsymbol{x})$ over a 2-dimensional space ($\boldsymbol{x} \in \mathbb{R}^n$ where $n = 2$). Recall that MAF is comprised of Masked Autoencoder for Distribution Estimation (MADE) blocks, which has a special masking scheme at each layer such that the autoregressive property is preserved. In particular, we consider a Gaussian autoregressive model:

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i \mid \boldsymbol{x}_{<i}) \tag{1}$$

such that the conditional Gaussians $p(x_i \mid \boldsymbol{x}_{<i}) = \mathcal{N}\left(x_i \mid \mu_i, (\exp(\alpha_i))^2\right)$ are parameterized by neural networks $\mu_i = f_{\mu_i}(\boldsymbol{x}_{<i})$ and $\alpha_i = f_{\alpha_i}(\boldsymbol{x}_{<i})$. Note that α_i denotes the log standard deviation of the Gaussian $p(x_i \mid \boldsymbol{x}_{<i})$. As seen in lecture, a normalizing flow uses a series of deterministic and invertible mappings $f : \mathbb{R}^n \to \mathbb{R}^n$ such that $x = f(z)$ and $z = f^{-1}(x)$ to transform a simple prior distribution p_z (e.g. isotropic Gaussian) into a more expressive one. In particular, a normalizing flow which composes k invertible transformations $\{f_j\}_{j=1}^{k}$ such that $\boldsymbol{x} = f^k \circ f^{k-1} \circ \cdots \circ f^1(\boldsymbol{z}_0)$ takes advantage of the change-of-variables property:

$$\log p(\boldsymbol{x}) = \log p_z\left(f^{-1}(\boldsymbol{x})\right) + \sum_{j=1}^{k} \log \left| \det \left(\frac{\partial f_j^{-1}(\boldsymbol{x}_j)}{\partial \boldsymbol{x}_j} \right) \right| \tag{2}$$

In MAF, the forward mapping is: $x_i = \mu_i + z_i \cdot \exp(\alpha_i)$, and the inverse mapping is: $z_i = (x_i - \mu_i)/\exp(\alpha_i)$. The log of the absolute value of the determinant of the Jacobian is:

$$\log \left| \det \left(\frac{\partial f^{-1}}{\partial \boldsymbol{x}} \right) \right| = -\sum_{i=1}^{n} \alpha_i \tag{3}$$

where μ_i and α_i are as defined above.

Your job is to implement and train a 5-layer MAF model on the Moons dataset for 100 epochs by modifying the `MADE` and `MAF` classes in the `flow_network.py` file. Note that we have provided an implementation of the sequential-ordering masking scheme for MADE. Also note that f_k is the same as f^k.

(a) **[5 points (Coding)]** Implement the `forward` function in the `MADE` class in `submission/flow_network.py`. The forward pass describes the mapping $x_i = \mu_i + z_i \cdot \exp(\alpha_i)$.

(b) **[5 points (Coding)]** Implement the `inverse` function in the `MADE` class in `submission/flow_network.py`. The inverse pass describes the mapping $z_i = (x_i - \mu_i)/\exp(\alpha_i)$.

(c) **[5 points (Coding)]** Implement the `log_probs` function in the `MAF` class in `submission/flow_network.py`. To train the MAF model for 50 epochs, execute

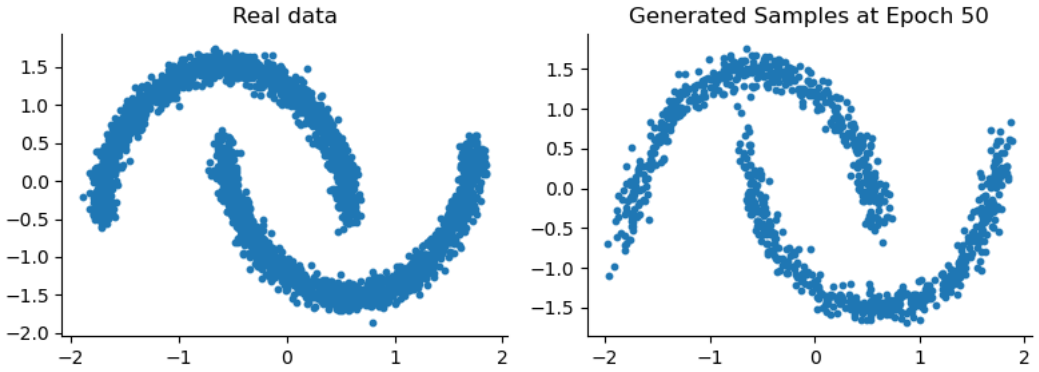
```
python main.py --model flow
```

For GPU acceleration run run the command below. **Note:** we are not supporting MPS GPUs as it trains slower than CPU-enabled training on Apple Silicon devices.

```
python main.py --model flow --device gpu
```

Hint: you should be getting a validation/test loss of around -1.2 nats.

Visualize 1000 samples drawn the model after is has been trained, which you can do by finding the figure in `maf/samples_epoch50.png`. Your samples will not be perfect, but should for the most part resemble the shape of the original dataset. For reference, it should look like the image below:



2 Generative adversarial networks (GANs)

In this problem, we will implement a generative adversarial network (GAN) that models a high-dimensional data distribution $p_{\text{data}}(\boldsymbol{x})$, where $\boldsymbol{x} \in \mathbb{R}^n$. To do so, we will define a generator $G_\theta : \mathbb{R}^k \to \mathbb{R}^n$; we obtain samples from our model by first sampling a k-dimensional random vector $\boldsymbol{z} \sim \mathcal{N}(0, I)$ and then returning $G_\theta(\boldsymbol{z})$. We will also define a discriminator $D_\phi : \mathbb{R}^n \to (0, 1)$ that judges how realistic the generated images $G_\theta(\boldsymbol{z})$ are, compared to samples from the data distribution $x \sim p_{\text{data}}(\boldsymbol{x})$. Because its output is intended to be interpreted as a probability, the last layer of the discriminator is frequently the **sigmoid** function,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

which constrains its output to fall between 0 and 1. For convenience, let $h_\phi(\boldsymbol{x})$ denote the activation of the discriminator right before the sigmoid layer, i.e. let $D_\phi(\boldsymbol{x}) = \sigma(h_\phi(\boldsymbol{x}))$. The values $h_\phi(\boldsymbol{x})$ are also called the discriminator's **logits**.

There are several common variants of the loss functions used to train GANs. They can all be described as a procedure where we alternately perform a gradient descent step on $L_D(\phi; \theta)$ with respect to ϕ to train the discriminator D_ϕ, and a gradient descent step on $L_G(\theta; \phi)$ with respect to θ to train the generator G_θ:

$$\min_\phi L_D(\phi; \theta), \qquad \min_\theta L_G(\theta; \phi). \tag{5}$$

In lecture, we talked about the following losses, where the discriminator's loss is given by

$$L_D(\phi; \theta) = -\mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D_\phi(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{z} \sim \mathcal{N}(0,I)}[\log (1 - D_\phi (G_\theta (\boldsymbol{z})))] \tag{6}$$

and the generator's loss is given by the **minimax loss**

$$L_G^{\text{minimax}}(\theta; \phi) = \mathbb{E}_{\boldsymbol{z} \sim \mathcal{N}(0,I)}[\log (1 - D_\phi (G_\theta (\boldsymbol{z})))] \tag{7}$$

(a) **[3 points (Written)]** Unfortunately, this form of loss for L_G suffers from a *vanishing gradient* problem. In terms of the discriminator's logits, the minimax loss is

$$L_G^{\text{minimax}}(\theta; \phi) = \mathbb{E}_{\boldsymbol{z} \sim \mathcal{N}(0,I)}[\log (1 - \sigma (h_\phi (G_\theta (\boldsymbol{z}))))] \tag{8}$$

Show that the derivative of L_G^{minimax} with respect to θ is approximately 0 if $D (G_\theta (\boldsymbol{z})) \approx 0$, or equivalently, if $h_\phi (G_\theta (\boldsymbol{z})) \ll 0$. You may use the fact that $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Why is this problematic for the training of the generator when the discriminator successfully identifies a fake sample $G_\theta(\boldsymbol{z})$?

To help you start the proof: Using the chain rule and the fact that $\sigma'(x) = \sigma(x)(1 - \sigma(x))$,

$$\frac{\partial L_G^{\text{minimax}}}{\partial \theta} = \mathbb{E}_{\boldsymbol{z} \sim \mathcal{N}(0,I)} \left[-\frac{\sigma'(h_\phi(G_\theta(\boldsymbol{z})))}{1 - \sigma(h_\phi(G_\theta(\boldsymbol{z})))} \frac{\partial}{\partial \theta} h_\phi (G_\theta (\boldsymbol{z})) \right] =$$

(b) **[7 points (Coding)]** Because of this vanishing gradient problem, in practice, L_G^{minimax} is typically replaced with the **non-saturating loss**

$$L_G^{\text{non-saturating}}(\theta; \phi) = -E_{\boldsymbol{z} \sim \mathcal{N}(0,I)}[\log D_\phi (G_\theta (\boldsymbol{z}))] \tag{9}$$

To turn the non-saturating loss into a concrete algorithm, we will take alternating gradient steps on Monte Carlo estimates of L_D and $L_G^{\text{non-saturating}}$:

$$L_D(\phi; \theta) \approx -\frac{1}{m} \sum_{i=1}^{m} \log D_\phi \left(\boldsymbol{x}^{(i)} \right) - \frac{1}{m} \sum_{i=1}^{m} \log \left(1 - D_\phi \left(G_\theta \left(\boldsymbol{z}^{(i)} \right) \right) \right) \tag{10}$$

$$L_G^{\text{non-saturating}}(\theta;\phi) \approx -\frac{1}{m}\sum_{i=1}^{m} \log D_\phi\left(G_\theta\left(\boldsymbol{z}^{(i)}\right)\right) \tag{11}$$

where m is the batch size, and for $i = 1, ..., m$, we sample $\boldsymbol{x}^{(i)} \sim p_{\text{data}}(\boldsymbol{x})$ and $\boldsymbol{z}^{(i)} \sim \mathcal{N}(0, I)$.

Implement and train a non-saturating GAN on Fashion MNIST for one epoch. Read through `main.py`, and in `submission/gan.py`, implement the `loss_nonsaturating_g` and `loss_nonsaturating_d` functions.

To train the model, execute:

```
python main.py --model gan --out_dir gan_nonsat
```

For GPU acceleration run run the command below. **Note:** we are not supporting MPS GPUs as it trains slower than CPU-enabled training on Apple Silicon devices.

```
python main.py --model gan --out_dir gan_nonsat --device gpu
```

You may monitor the GAN's output in the `gan_nonsat` directory. Note that because the GAN is only trained for one epoch, we cannot expect the model's output to produce very realistic samples, but they should be roughly recognizable as clothing items.

Hint: Note that $1 - \sigma(x) = \sigma(-x)$.

For reference, the generated examples should look similar to the image below:

3 Divergence minimization

Now, let us analyze some theoretical properties of GANs. For convenience, we will denote $p_\theta(\boldsymbol{x})$ to be the distribution whose samples are generated by first sampling $\boldsymbol{z} \sim \mathcal{N}(0, I)$ and then returning the sample $G_\theta(\boldsymbol{z})$. With this notation, we may compactly express the discriminator's loss as

$$L_D(\phi; \theta) = -\mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D_\phi(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{x} \sim p_\theta(\boldsymbol{x})}[\log(1 - D_\phi(\boldsymbol{x}))] \tag{12}$$

(a) **[3 points (Written)]** Show that L_D is minimized when $D_\phi = D^*$, where

$$D^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_\theta(\boldsymbol{x}) + p_{\text{data}}(\boldsymbol{x})} \tag{13}$$

Hint: for a fixed $\boldsymbol{x}$,what t minimizes $f(t) = -p_{\text{data}}(\boldsymbol{x}) \log t - p_\theta(\boldsymbol{x}) \log(1 - t)$?

To help you get started with the proof: If we break the expectation up, we see that

$$\begin{aligned} L_D(\phi; \theta) &= -\mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D_\phi(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{x} \sim p_\theta(\boldsymbol{x})}[\log(1 - D_\phi(\boldsymbol{x}))] \\ &= -\int p_{\text{data}}(\boldsymbol{x}) \log D_\phi(\boldsymbol{x}) d\boldsymbol{x} - \int p_\theta(\boldsymbol{x}) \log(1 - D_\phi(\boldsymbol{x})) d\boldsymbol{x} \\ &= -\int \left(p_{\text{data}}(\boldsymbol{x}) \log D_\phi(\boldsymbol{x}) + p_\theta(\boldsymbol{x}) \log(1 - D_\phi(\boldsymbol{x})) \right) d\boldsymbol{x} \\ &= \int f(D_\phi(\boldsymbol{x})) d\boldsymbol{x} \end{aligned}$$

We can set $L_D'(\phi; \theta) = 0$ to obtain the optimal L_D'. This yields

$$L_D'(\phi; \theta) = \frac{d}{dD_\phi(\boldsymbol{x})} \int f(D_\phi(\boldsymbol{x})) d\boldsymbol{x} = \int \frac{d}{dD_\phi(\boldsymbol{x})} f(D_\phi(\boldsymbol{x})) d\boldsymbol{x} = 0$$

Now try to apply the hint!

(b) **[1.50 points (Written)]** Recall that $D_\phi(\boldsymbol{x}) = \sigma(h_\phi(\boldsymbol{x}))$. Show that the logits $h_\phi(\boldsymbol{x})$ of the discriminator estimate the log of the likelihood ratio of $\boldsymbol{x}$ under the true distribution compared to the model's distribution; that is, show that if $D_\phi = D^*$, then

$$h_\phi(\boldsymbol{x}) = \log \frac{p_{\text{data}}(\boldsymbol{x})}{p_\theta(\boldsymbol{x})} \tag{14}$$

To help you get started, note that

$$D_\phi(\boldsymbol{x}) = \sigma(h_\phi(\boldsymbol{x})) = \frac{1}{1 + e^{-h_\phi(\boldsymbol{x})}}$$

Setting this to the expression for $D^*(\boldsymbol{x})$ in part 3a solution, we find that

(c) **[1.50 points (Written)]** Consider a generator loss defined by the sum of the minimax loss and the non-saturating loss,

$$L_G(\theta; \phi) = \mathbb{E}_{\boldsymbol{x} \sim p_\theta(\boldsymbol{x})}[\log(1 - D_\phi(\boldsymbol{x}))] - \mathbb{E}_{\boldsymbol{x} \sim p_\theta(\boldsymbol{x})}[\log D_\phi(\boldsymbol{x})] \tag{15}$$

Show that if $D_\phi = D^*$, then

$$L_G(\theta; \phi) = \text{KL}(p_\theta(\boldsymbol{x}) \,||\, p_{\text{data}}(\boldsymbol{x})) \tag{16}$$

To get started

$$\begin{aligned} L_G(\theta; \phi) &= \mathbb{E}_{p_\theta(\boldsymbol{x})}[\log(1 - D_\phi(\boldsymbol{x}))] - \mathbb{E}_{p_\theta(\boldsymbol{x})}[\log D_\phi(\boldsymbol{x})] \\ &= \mathbb{E}_{p_\theta(\boldsymbol{x})}\left[\log \frac{1 - D_\phi(\boldsymbol{x})}{D_\phi(\boldsymbol{x})}\right] \end{aligned}$$

(d) **[1.50 points (Written)]** Recall that when training VAEs, we minimize the negative ELBO, an upper bound to the negative log likelihood. Show that the negative log likelihood, $-\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log p_\theta(\boldsymbol{x})]$, can be written as a KL divergence plus an additional term that is constant with respect to θ. We are asking if the KL divergence is equal to L_G, so after finding the expression, you will be able to deduce that. Note that the constant term is constant with respect to θ, so it can be another expectation.

Does this mean that a VAE decoder trained with ELBO and a GAN generator trained with the L_G defined in the previous part 3c are implicitly learning the same objective? Explain.

4 Conditional GAN with projection discriminator

So far, we have trained GANs that sample from a given dataset of images. However, many datasets come with not only images, but also labels that specify the class of that particular image. In the MNIST dataset, we have both the digit's image as well as its numerical identity. It is natural to want to generate images that correspond to a particular class.

Formally, an *unconditional* GAN is trained to produce samples $\boldsymbol{x} \sim p_\theta(\boldsymbol{x})$ that mimic samples $\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})$ from a data distribution. In the class-conditional setting, we instead have labeled data $(\boldsymbol{x}, y) \sim p_{\text{data}}(\boldsymbol{x}, y)$ and seek to train a model $p_\theta(\boldsymbol{x}, y)$. Since it is the class conditional generator $p_\theta(\boldsymbol{x} \mid y)$ that we are interested in, we will express $p_\theta(\boldsymbol{x}, y) = p_\theta(\boldsymbol{x} \mid y)p_\theta(y)$. We will set $p_\theta(\boldsymbol{x} \mid y)$ to be the distribution given by $G_\theta(\boldsymbol{z}, y)$, where $\boldsymbol{z} \sim \mathcal{N}(0, I)$ as usual. For simplicity, we will assume $p_{\text{data}}(y) = \frac{1}{m}$ and set $p_\theta(y) = \frac{1}{m}$, where m is the number of classes. In this case, the discriminator's loss becomes

$$L_D(\phi; \theta) = -\mathbb{E}_{(\boldsymbol{x},y)\sim p_{\text{data}}(\boldsymbol{x},y)}[\log D_\phi(\boldsymbol{x}, y)] - \mathbb{E}_{(\boldsymbol{x},y)\sim p_\theta(\boldsymbol{x},y)}[\log(1 - D_\phi(\boldsymbol{x}, y))] \tag{17}$$

$$= -\mathbb{E}_{(\boldsymbol{x},y)\sim p_{\text{data}}(\boldsymbol{x},y)}[\log D_\phi(\boldsymbol{x}, y)] - \mathbb{E}_{y\sim p_\theta(y)}[\mathbb{E}_{\boldsymbol{z}\sim\mathcal{N}(0,I)}[\log(1 - D_\phi(G_\theta(\boldsymbol{z}, y), y))]] \tag{18}$$

Therefore, the main difference for the conditional GAN is that we must structure our generator $G_\theta(\boldsymbol{z}, y)$ and discriminator $D_\phi(\boldsymbol{x}, y)$ to accept the class label y as well. For the generator, one simple way to do so is to encode y as a one-hot vector $\boldsymbol{y}$ and concatenate it to $\boldsymbol{z}$, and then apply neural network layers normally. (A one-hot representation of a class label y is an m-dimensional vector $\boldsymbol{y}$ that is 1 in the yth entry and 0 everywhere else.)

In practice, the effectiveness of the model is strongly dependent on the way the discriminator depends on y. One heuristic with which to design the discriminator is to mimic the form of the theoretically optimal discriminator. That is, we can structure the neural network used to model D_ϕ based on the form of D^*, where D^* minimizes L_D. To calculate the theoretically optimal discriminator, though, it is necessary to make some assumptions.

(a) **[3 points (Written)]** Suppose that when $(\boldsymbol{x}, y) \sim p_{\text{data}}(\boldsymbol{x}, y)$, there exists a feature mapping φ under which $\varphi(\boldsymbol{x})$ becomes a mixture of m unit Gaussians, with one Gaussian per class label y. Assume that when $(\boldsymbol{x}, y) \sim p_\theta(\boldsymbol{x}, y), \varphi(\boldsymbol{x})$ also becomes a mixture of m unit Gaussians, again with one Gaussian per class label y. Concretely, we assume that the ratio of the conditional probabilities can be written as

$$\frac{p_{\text{data}}(\boldsymbol{x} \mid y)}{p_\theta(\boldsymbol{x} \mid y)} = \frac{\mathcal{N}(\varphi(\boldsymbol{x}) \mid \boldsymbol{\mu_y}, I)}{\mathcal{N}(\varphi(\boldsymbol{x}) \mid \hat{\boldsymbol{\mu}}_{\boldsymbol{y}}, I)} \tag{19}$$

where $\boldsymbol{\mu_y}$ and $\hat{\boldsymbol{\mu}}_{\boldsymbol{y}}$ are the means of the Gaussians for p_{data} and p_θ respectively.

Show that under this simplifying assumption, the optimal discriminator's logits $h^*(\boldsymbol{x}, y)$ can be written in the form

$$h^*(\boldsymbol{x}, y) = \boldsymbol{y}^T(A\varphi(\boldsymbol{x}) + \boldsymbol{b}) \tag{20}$$

for some matrix A and vector $\boldsymbol{b}$, where $\boldsymbol{y}$ is a one-hot vector denoting the class y. In this problem, the discriminator's output and logits are related by $D_\phi(\boldsymbol{x}, y) = \sigma(h_\phi(\boldsymbol{x}, y))$. In order to express $\mu_y - \hat{\mu}_y$ in terms of y, given that y is a one-hot vector, see if you can write $\mu_1 - \hat{\mu}_1$ as a matrix multiplication of y and a matrix whose rows are $\mu_i - \hat{\mu}_i$.

Hint: use the result from problem 3b. Along with that hint, try expanding the PDF for the p terms using the fact that they are normal distributions with known parameters.

To help you get started:

$$h_\phi(x, y) = \log \frac{p_{\text{data}}(\boldsymbol{x},y)}{p_\theta(\boldsymbol{x},y)}$$
$$= \log \frac{p_{\text{data}}(\boldsymbol{x}|y)}{p_\theta(\boldsymbol{x}|y)} + \log \frac{p_{\text{data}}(y)}{p_\theta(y)}$$
$$= \log \frac{p_{\text{data}}(\boldsymbol{x}|y)}{p_\theta(\boldsymbol{x}|y)} =$$

(b) **[6 points (Coding)]** Implement and train a conditional GAN on Fashion MNIST for one epoch. The discriminator has the structure described in part 4, with φ, A and b parameterized by a neural network with a final linear layer, and the generator accepts a one-hot encoding of the class. In `submission/gan.py`, implement the `conditional_loss_nonsaturating_g` and `conditional_loss_nonsaturating_d` functions.

To train the model, execute:

```
python main.py --model cgan --out_dir gan_nonsat_conditional
```

For GPU acceleration run run the command below. **Note:** we are not supporting MPS GPUs as it trains slower than CPU-enabled training on Apple Silicon devices.

```
python main.py --model cgan --out_dir gan_nonsat_conditional --device gpu
```

You may monitor the GAN's output in the `gan_nonsat_conditional` directory. You should be able to roughly recognize the categories that correspond to each column. The final image generated after training should resemble the following image below:

For reference, the generated examples should look similar to the image below:

5 Wasserstein GAN

In many cases, the GAN algorithm can be thought of as minimizing a divergence between a data distribution $p_{\text{data}}(\boldsymbol{x})$ and the model distribution $p_\theta(\boldsymbol{x})$. For example, the minimax GAN discussed in the lectures minimizes the Jensen-Shannon divergence, and the loss in problem 3c minimizes the KL divergence. In this problem, we will explore an issue with these divergences and one potential way to fix it. Note that for subproblems (a) to (d), x is not bolded denoting a scalar. This is because $\theta \in \mathbb{R}$ suggests that we are working with a single-variable gaussian distribution.

(a) **[2 points (Written)]** Let $p_\theta(x) = \mathcal{N}(x \mid \theta, \epsilon^2)$ and $p_{\text{data}}(x) = \mathcal{N}(x \mid \theta_0, \epsilon^2)$ be normal distributions with standard deviation ϵ centered at $\theta \in \mathbb{R}$ and $\theta_0 \in \mathbb{R}$ respectively. Show that

$$\text{KL}(p_\theta(x) \,||\, p_{\text{data}}(x)) = \frac{(\theta - \theta_0)^2}{2\epsilon^2} \tag{21}$$

To help you get started:

$$\text{KL}(p_\theta(x) \,||\, p_{\text{data}}(x)) = \mathbb{E}_{x \sim \mathcal{N}(\theta, \epsilon^2)} \left[\log \frac{\exp(-\frac{1}{2\epsilon^2}(x-\theta)^2)}{\exp(-\frac{1}{2\epsilon^2}(x-\theta_0)^2)} \right] =$$

(b) **[1.50 points (Written)]** Suppose $p_\theta(x)$ and $p_{\text{data}}(x)$ both place probability mass in only a very small part of the domain; that is, consider the limit $\epsilon \to 0$. What happens to $\text{KL}(p_\theta(x) \,||\, p_{\text{data}}(x))$ and its derivative with respect to θ, assuming that $\theta \neq \theta_0$? Why is this problematic for a GAN trained with the loss function L_G defined in problem 3c?

(c) **[1.50 points (Written)]** To avoid this problem, we'll propose an alternative objective for the discriminator and generator. Consider the following alternative objectives:

$$L_D(\phi; \theta) = \mathbb{E}_{x \sim p_\theta(x)}[D_\phi(x)] - \mathbb{E}_{x \sim p_{\text{data}}(x)}[D_\phi(x)] \tag{22}$$

$$L_G(\theta; \phi) = -\mathbb{E}_{x \sim p_\theta(x)}[D_\phi(x)] \tag{23}$$

where D_ϕ is no longer constrained to functions that output a probability; instead D_ϕ can be a function that outputs any real number. As defined, however, these losses are still problematic. Again consider the limit $\epsilon \to 0$; that is, let $p_\theta(x)$ be the distribution that outputs $\theta \in \mathbb{R}$ with probability 1, and let $p_{\text{data}}(x)$ be the distribution that outputs $\theta_0 \in \mathbb{R}$ with probability 1. Why is there no discriminator D_ϕ that minimizes this new objective L_D?

(d) **[1.50 points (Written)]** Let's tweak the alternate objective so that an optimal discriminator exists. Consider the same objective L_D and the same limit $\epsilon \to 0$. Now, suppose that D_ϕ is restricted to differentiable functions whose derivative with respect to x is always between -1 and 1. It can still output any real number. Is there now a discriminator D_ϕ out of this class of functions that minimizes L_D? Briefly describe what the optimal D_ϕ looks like as a function of x.

(e) **[7 points (Coding)]** The Wasserstein GAN with gradient penalty (WGAN-GP) enables stable training by penalizing functions whose derivatives are too large. It achieves this by adding a penalty on the 2-norm of the gradient of the discriminator at various points in the domain. It is defined by

$$L_D(\phi; \theta) = \mathbb{E}_{\boldsymbol{x} \sim p_\theta(\boldsymbol{x})}[D_\phi(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[D_\phi(\boldsymbol{x})] + \lambda \mathbb{E}_{\boldsymbol{x} \sim r_\theta(\boldsymbol{x})}[(\|\nabla D_\phi(\boldsymbol{x})\|_2 - 1)^2] \tag{24}$$

$$L_G(\theta; \phi) = -\mathbb{E}_{\boldsymbol{x} \sim p_\theta(\boldsymbol{x})}[D_\phi(\boldsymbol{x})] \tag{25}$$

where $r_\theta(\boldsymbol{x})$ is defined by sampling $\alpha \sim \text{Uniform}([0, 1])$, $\boldsymbol{x}_1 \sim p_\theta(\boldsymbol{x})$, and $\boldsymbol{x}_2 \sim p_{\text{data}}(\boldsymbol{x})$, and returning $\alpha \boldsymbol{x}_1 + (1 - \alpha)\boldsymbol{x}_2$. The hyperparameter λ controls the strength of the penalty; a setting that usually works is $\lambda = 10$. Also note that $\|\nabla D_\phi(\boldsymbol{x})\|_2$ is the Frobenius norm.

Implement and train WGAN-GP for one epoch on Fashion MNIST. In `submission/gan.py`, implement the `loss_wasserstein_gp_g` and `loss_wasserstein_gp_d` functions.

To train the model, execute:

```
python main.py --model gan --out_dir gan_wass_gp --loss_type wasserstein_gp
```

For GPU acceleration run run the command below. **Note:** we are not supporting MPS GPUs as it trains slower than CPU-enabled training on Apple Silicon devices.

```
python main.py --model gan --out_dir gan_wass_gp --loss_type wasserstein_gp --device gpu
```

You may monitor the GAN's output in the `gan_wass_gp` directory. The generated images should resemble the image below:

Hint: Avoid using for loops and try a vectorized approach. For the vectorization, consider if you can get a common term, such that differentiating that term with respect to $x^{(i)}$ gives you the required gradient for that input.

We are trying to obtain the following matrix of derivatives (we will then take the norm of each row, for use in the gradient penalty):

$$\texttt{grad} = \begin{bmatrix} \frac{\partial D(\boldsymbol{x}^{(1)})}{\partial \boldsymbol{x}^{(1)}} \\ \vdots \\ \frac{\partial D(\boldsymbol{x}^{(m)})}{\partial \boldsymbol{x}^{(m)}} \end{bmatrix}$$

In principle, we could compute $\frac{\partial D(\boldsymbol{x}^{(i)})}{\partial \boldsymbol{x}^{(i)}}$ using a `for` loop over each element in the batch and then stack the resulting derivates. However, this is inefficient. Instead notice that

$$\frac{\partial D(\boldsymbol{x}^{(i)})}{\partial \boldsymbol{x}^{(i)}} = \frac{\partial}{\partial \boldsymbol{x}^{(i)}} \sum_{j=1}^{m} D(\boldsymbol{x}^{(j)})$$

because each $D(\boldsymbol{x}^{(j)})$ is constant w.r.t. $\boldsymbol{x}^{(i)}$ for $i \neq j$. Therefore, if we let

$$X = \begin{bmatrix} \boldsymbol{x}^{(1)} \\ \vdots \\ \boldsymbol{x}^{(m)} \end{bmatrix}$$

we find that

$$\texttt{grad} = \frac{\partial}{\partial X} \sum_{j=1}^{m} D(\boldsymbol{x}^{(j)})$$