# XCS236 Problem Set 2

**Due Sunday, June 2 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's Note**

PS2 requires more coding than PS1 and training multiple models. It is important that you start this assignment early! It is also in your best interest to make sure that your code is properly vectorized (at no point in this assignment do you need to implement a for-loop) 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. Also, if your GPU has limited memory ($\leq$ 2 GB), you will likely run into Out-of-Memory issues on Problem 3.

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

1. 7 minutes for `vae.py`
2. 7 minutes for `gmvae.py`
3. 50 minutes for `ssvae.py`
4. $\geq$ 3 hours for `fsvae.py`. We recommend stopping training the moment your samples look somewhat decent (maybe around 50000 iterations). Note that we periodically save models for you.

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

1. `sample_gaussian` in `src/submission/utils.py`
2. `negative_elbo_bound` in `src/submission/models/vae.py`
3. `log_normal` in `src/submission/utils.py`
4. `log_normal_mixture` in `src/submission/utils.py`
5. `negative_elbo_bound` in `src/submission/models/gmvae.py`
6. `negative_iwae_bound` in `src/submission/models/vae.py`
7. `negative_iwae_bound` in `src/submission/models/gmvae.py`
8. `negative_elbo_bound` in `src/submission/models/ssvae.py`
9. **Extra Credit:** `negative_elbo_bound` **in** `src/submission/models/fsvae.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.

# 1 Implementing the Variational Autoencoder (VAE)

For this problem we will be using `PyTorch` to implement the variational autoencoder (VAE) and learn a probabilistic model of the MNIST dataset of handwritten digits. Formally, we observe a sequence of binary pixels $\mathbf{x} \in \{0,1\}^d$, and let $\mathbf{z} \in \mathbb{R}^k$ denote a set of latent variables. Our goal is to learn a latent variable model $p_\theta(\mathbf{x})$ of the high-dimensional data distribution $p_{\text{data}}(\mathbf{x})$.

The VAE is a latent variable model that learns a specific parameterization $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int p(\mathbf{z}) p_\theta(\mathbf{x} \mid \mathbf{z}) d\mathbf{z}$. Specifically, the VAE is defined by the following generative process:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z} \mid 0, I) \tag{1}$$

$$p_\theta(\mathbf{x} \mid \mathbf{z}) = \text{Bern}\left(\mathbf{x} \mid f_\theta\left(\mathbf{z}\right)\right) \tag{2}$$

In other words, we assume that the latent variables $\mathbf{z}$ are sampled from a unit Gaussian distribution $\mathcal{N}(\mathbf{x} \mid 0, I)$. The latent $\mathbf{z}$ are then passed through a neural network decoder $f_\theta(\cdot)$ to obtain the *logits* of the $d$ Bernoulli random variables which model the pixels in each image.

Although we would like to maximize the marginal likelihood $p_\theta(\mathbf{x})$, computation of $p_\theta(\mathbf{x}) = \int p(\mathbf{z}) p_\theta(\mathbf{x} \mid \mathbf{z}) d\mathbf{z}$ is generally intractable as it involves integration over all possible values of $\mathbf{z}$. Therefore, we posit a variational approximation to the true posterior and perform amortized inference as we have seen in class:

$$q_\phi(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\left(\mathbf{z} \mid \mu_\phi\left(\mathbf{x}\right), \text{diag}\left(\sigma_\phi^2\left(\mathbf{x}\right)\right)\right) \tag{3}$$

Specifically, we pass each image $\mathbf{x}$ through a neural network which outputs the mean $\mu_\phi$ and diagonal covariance $\text{diag}(\sigma_\phi^2(\mathbf{x}))$ of the multivariate Gaussian distribution that approximates the distribution over latent variables $\mathbf{z}$ given $\mathbf{x}$. We then maximize the lower bound to the marginal log-likelihood to obtain an expression known as the **evidence lower bound (ELBO)**:

$$\log p_\theta(\mathbf{x}) \geq \text{ELBO}(\mathbf{x}; \theta, \phi) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z}) - D_{\text{KL}}\left(q_\phi\left(\mathbf{z} \mid \mathbf{x}\right) \mid\mid p\left(\mathbf{z}\right)\right)] \tag{4}$$

Notice that the negatation of the ELBO decomposes into two terms: (1) **the reconstruction loss**: $-\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})]$, and (2) **the Kullback-Leibler (KL) term**: $D_{\text{KL}}(q_\phi(\mathbf{z} \mid \mathbf{x}) \mid\mid p(\mathbf{z}))$, e.g. -ELBO = recon. loss + KL div. Hence, VAE learning objective is to minimize the negative ELBO.

**How does this relate to Lecture's ELBO?** In lecture, we learned that ELBO objective as

$$\log p(\mathbf{x}; \theta) \geq \sum_z q(\mathbf{z}; \phi) \log p(\mathbf{z}, \mathbf{x}; \theta) + H(q(\mathbf{z}; \phi)) = \mathcal{L}(\mathbf{x}; \theta, \phi)$$
$$\log p(\mathbf{x}; \theta) = \mathcal{L}(\mathbf{x}; \theta, \phi) + D_{\text{KL}}(q(\mathbf{z}; \phi) \mid\mid p(\mathbf{z} \mid \mathbf{x}; \theta))$$

where $\theta$ is the decoder, $\phi$ is the encoder. However it is not scalable to define an encoder $\phi$ for each data point $x$, which is why we introduce the amortized distribution $q_\phi(\mathbf{z} \mid \mathbf{x})$ which learns a neural network to regressively provide $\phi$ given $\mathbf{x}$. If you were to replace $q(\mathbf{z}; \phi)$ with the amortized function, you will generate the ELBO expression defined above. The problem statements representation of the ELBO allows you to easily see that maximizing the ELBO really implies minimizing the KL Divergence between $q_\phi$ and $p(\mathbf{z})$ which means that is has an inclination to train the encoder to make $q$ similar to the prior, serving as a regularization force.

Your objective is to implement the variational autoencoder by modifying `utils.py` and `vae.py`.

(a) **[3 points (Coding)]** Implement the reparameterization trick in the function `sample_gaussian` of `utils.py`. Specifically, your answer will take in the mean `m` and variance `v` of the Gaussian distribution $q_\phi(\mathbf{z} \mid \mathbf{x})$ and return a sample $\mathbf{z} \sim q_\phi(\mathbf{z} \mid \mathbf{x})$.

(b) **[7 points (Coding)]** Next, implement `negative_elbo_bound` in the file `vae.py`. Several of the functions in `utils.py` will be helpful, so please check what is provided. Note that we ask for the *negative* ELBO, as `PyTorch` optimizers *minimize* the loss function. Additionally, since we are computing the negative ELBO over a mini-batch of data $\{\mathbf{x}^{(i)}\}_{i=1}^n$, make sure to compute the *average* $-\frac{1}{n}\sum_{i=1}^n \text{ELBO}(\mathbf{x}^{(i)}; \theta, \phi)$ over the

mini-batch. Finally, note that the ELBO itself cannot be computed exactly since exact computation of the reconstruction term is intractable. Instead we ask that you estimate the reconstruction term via Monte Carlo sampling

$$-\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})] \approx -\log p_\theta(\mathbf{x} \mid \mathbf{z}^{(1)}) \tag{5}$$

where $\mathbf{z}^{(1)} \sim q_\phi(\mathbf{z} \mid \mathbf{x})$ denotes a single sample. The function `kl_normal` in `utils.py` will be helpful.

**Note**: `negative_elbo_bound` also expects you to return the *average* reconstruction loss and *average* KL divergence.

(c) **[3 points (Coding)]** To train the VAE, run

```
python main.py --model vae --train
```

To use GPU acceleration run the command below.

```
python main.py --model vae --train --device gpu
```

Once the run is complete (20000 iterations), it will output (assuming your implementation is correct): the (1) average negative ELBO, (2) average KL term, and (3) average reconstruction loss as evaluated on a test subset that we have selected. These three numbers will be reported to `submission/VAE.pkl`. To check the accuracy of your values run

```
python grader.py 1c-0-basic
```

Since we're using stochastic optimization, you may wish to run the model multiple times and report each metric's mean and corresponding standard error. We recommend running training 5-10 times.

**Hint**: the negative ELBO on the test subset should be somewhere around 100. If you generate other NELBO values outside this range, please revisit your implementation of functions in `utils.py`, specifically `sample_gaussian`.

Also, to visualize 200 digits sampled from $p_\theta(\mathbf{x})$, you can run after training:

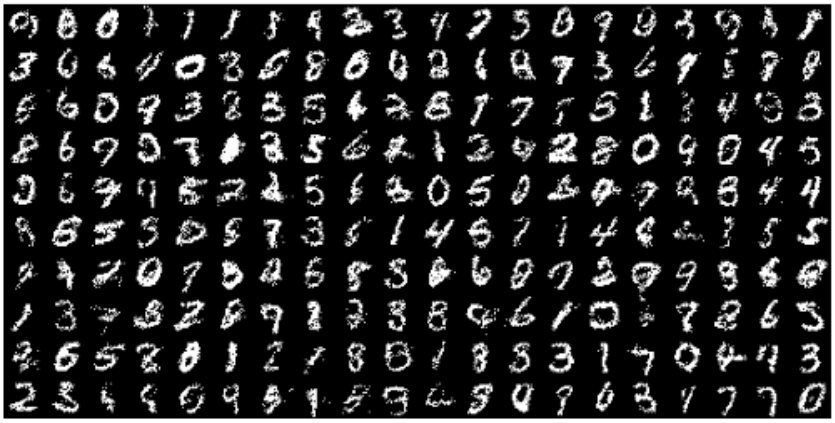
```
python main.py --model vae
```

Your samples will be saved to: `model=vae_z=10_run=0000.png`. The generated samples should look like the image below:

(d) **[5 points (Written)]** A popular variation of the normal VAE is called the $\beta$-VAE. The $\beta$-VAE optimizes the following objective:

$$\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})] - \beta D_{\mathrm{KL}}\left(q_\phi\left(\mathbf{z} \mid \mathbf{x}\right) \,||\, p\left(\mathbf{z}\right)\right) \tag{6}$$

Here, $\beta$ is a positive real number. Offer an intuitive interpretation of the impact of $\beta$ on the optimization of the VAE; specifically, what happens when $\beta = 1$? How about when $\beta$ is increased above 1? For this question, a qualitative description will suffice.



## 2 Implementing the Mixture of Gaussians VAE (GMVAE)

Recall that in Problem 1, the VAE's prior distribution was a parameter-free isotropic Gaussian $p(\mathbf{z}) = \mathcal{N}(\mathbf{z} \mid 0, I)$. While this original setup works well, there are settings in which we desire more expressivity to better model our data. In this problem we will implement the GMVAE, which has a mixture of Gaussians as the prior distribution. Specifically:

$$p_\theta(\mathbf{z}) = \sum_{i=1}^{k} \frac{1}{k} \mathcal{N}\left(\mathbf{z} \mid \mu_i, \operatorname{diag}\left(\sigma_i^2\right)\right) \tag{7}$$

where $i \in \{1, ..., k\}$ denotes the $i^{\text{th}}$ cluster index. For notational simplicity, we shall subsume our mixture of Guassian parameters $\{\mu_i, \sigma_i\}_{i=1}^k$ into our generative model parameters $\theta$. For simplicity, we have also assumed fixed uniform weights $1/k$ over the possible different clusters. Apart from the prior, the GMVAE shares an identical setup as the VAE:

$$q_\phi(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\left(\mathbf{z} \mid \mu_\phi\left(\mathbf{x}\right), \operatorname{diag}\left(\sigma_\phi^2\left(\mathbf{x}\right)\right)\right) \tag{8}$$

$$p_\theta(\mathbf{x} \mid \mathbf{z}) = \operatorname{Bern}\left(\mathbf{x} \mid f_\theta\left(\mathbf{z}\right)\right) \tag{9}$$

Although the ELBO for the GMVAE: $\mathbb{E}_{q_\phi(\mathbf{z})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})] - D_{\mathrm{KL}}(q_\phi(\mathbf{z} \mid \mathbf{x}) \mid\mid p_\theta(\mathbf{z}))$ is identical to that of the VAE, we note that the KL term $D_{\mathrm{KL}}(q_\phi(\mathbf{z} \mid \mathbf{x}) \mid\mid p_\theta(\mathbf{z}))$ cannot be computed analytically between a Gaussian distribution $q_\phi(\mathbf{z} \mid \mathbf{x})$ and a mixture of Gaussians $p_\theta(\mathbf{z})$. However, we can obtain its unbiased estimator via Monte Carlo sampling:

$$D_{\mathrm{KL}}(q_\phi(\mathbf{z} \mid \mathbf{x}) \mid\mid p_\theta(\mathbf{z})) \approx \log q_\theta(\mathbf{z}^{(1)} \mid \mathbf{x}) - \log p_\theta(\mathbf{z}^{(1)}) \tag{10}$$

$$= \underbrace{\log \mathcal{N}(\mathbf{z}^{(1)} \mid \mu_\phi(\mathbf{x}), \operatorname{diag}(\sigma_\phi^2(\mathbf{x})))}_{\texttt{log\_normal}} - \underbrace{\log \sum_{i=1}^{k} \frac{1}{k} \mathcal{N}(\mathbf{z}^{(1)} \mid \mu_i, \operatorname{diag}(\sigma_i^2))}_{\texttt{log\_normal\_mixture}} \tag{11}$$

where $\mathbf{z}^{(1)} \sim q_\phi(\mathbf{z} \mid \mathbf{x})$ denotes a single sample.

(a) **[10 points (Coding)]** Implement the (1) `log_normal` and (2) `log_normal_mixture_functions` in `utils.py`, and the

function `negative_elbo_bound` in `gmvae.py`. The function `log_mean_exp` in `utils.py` will be helpful for this problem in ensuring that your implementation is numerically stable.

(b) **[10 points (Coding)]** To train the GMVAE, run

```
python main.py --model gmvae --train
```

To use GPU acceleration run the command below.

```
python main.py --model gmvae --train --device gpu
```

Once the run is complete (20000 iterations), it will output: the average (1) negative ELBO, (2) KL term, and (3) reconstruction loss as evaluated on a test subset that we have selected. These three numbers will be reported to `submission/GMVAE.pkl`. To check the accuracy of your values run

```
python grader.py 2b-0-basic
```

Since we're using stochastic optimization, you may wish to run the model multiple times and report each metric's mean and the corresponding standard error.

**Hint**: the negative ELBO on the test subset should be somewhere around 97-99. If you generate other NELBO values outside this range, please revisit your implementation of functions in `utils.py`, specifically `sample_gaussian`.

Also, to visualize 200 digits sampled from $p_\theta(\mathbf{x})$, you can run after training:

```
python main.py --model gmvae
```

Your samples will be saved to: `model=gmvae_z=10_run=0000.png`. The generated samples should look like the image below:

# 3 Implementing the Importance Weighted Autoencoder (IWAE)

While the ELBO serves as a lower bound to the true marginal log-likelihood, it may be loose if the variational posterior $q_\phi(\mathbf{z} \mid \mathbf{x})$ is a poor approximation to the true posterior $p_\theta(\mathbf{z} \mid \mathbf{x})$. It is worth noting that, for a fixed choice of $\mathbf{x}$, the ELBO is, in expectation, the log of the unnormalized density ratio:

$$\frac{p_\theta(\mathbf{x}, \mathbf{z})}{q_\phi(\mathbf{z} \mid \mathbf{x})} = \frac{p_\theta(\mathbf{z} \mid \mathbf{x})}{q_\phi(\mathbf{z} \mid \mathbf{x})} \cdot p_\theta(\mathbf{x}) \tag{12}$$

where $\mathbf{z} \sim q_\phi(\mathbf{z} \mid \mathbf{x})$. The term $\frac{p_\theta(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})}$ is normalized such that

$$E_{q_\phi(\mathbf{z}|\mathbf{x})}[\frac{p_\theta(\mathbf{z} \mid \mathbf{x})}{q_\phi(\mathbf{z} \mid \mathbf{x})}] = \int q_\phi(\mathbf{z} \mid \mathbf{x}) \frac{p_\theta(\mathbf{z} \mid \mathbf{x})}{q_\phi(\mathbf{z} \mid \mathbf{x})} d\mathbf{z} = \int p_\theta(\mathbf{z} \mid \mathbf{x}) d\mathbf{z} = 1. \tag{13}$$

meaning that if you add the $p_\theta(\mathbf{x})$ term, it no longer integrates to 1.

As can be seen from the RHS, the density ratio is *unnormalized* since the density ratio is multiplied by the constant $p_\theta(\mathbf{x})$. We can obtain a tighter bound by averaging multiple unnormalized density ratios. This is the key idea behind IWAE, which uses $m > 1$ samples from the approximate posterior $q_\phi(\mathbf{z} \mid \mathbf{x})$ to obtain the following IWAE bound:

$$\mathcal{L}_m(\mathbf{x}; \theta, \phi) = \mathbb{E}_{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)} \overset{\text{i.i.d.}}{\sim} q_\phi(\mathbf{z}|\mathbf{x})} \left( \log \frac{1}{m} \sum_{i=1}^{m} \frac{p_\theta\left(\mathbf{x}, \mathbf{z}^{(i)}\right)}{q_\phi\left(\mathbf{z}^{(i)} \mid \mathbf{x}\right)} \right) \tag{14}$$

Notice that for the special case of $m = 1$, the IWAE objective $\mathcal{L}_m$ reduces to the standard ELBO $L_1 = \mathbb{E}_{z \sim q_\phi(\mathbf{z}|\mathbf{x})} \left( \log \frac{p_\theta(\mathbf{x}, \mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \right)$.

(a) **[10 points (Written)]** Prove that IWAE is a valid lower bound of the log-likelihood, and that the ELBO lower bounds IWAE

$$\log p_\theta(\mathbf{x}) \geq \mathcal{L}_m(\mathbf{x}) \geq \mathcal{L}_1(\mathbf{x}) \tag{15}$$

for any $m \geq 1$.

**Hint**: consider Jensen's Inequality

(b) **[2.50 points (Coding)]** Implement IWAE for VAE in the `negative_iwae_bound` function in `vae.py`. The functions `duplicate` and `log_mean_exp` defined in `utils.py` will be helpful.

(c) **[10 points (Coding)]** Run the command below to evaluate your implementation against the test subset:

```
python main.py --model vae --iwae
```

This will output IWAE bounds for $m = \{1, 10, 100, 1000\}$. Check that the IWAE-1 result is consistent with your reported ELBO for the VAE. All four IWAE bounds for VAE will be reported to:

(a) `submission/VAE_iwae_1.pkl`

(b) `submission/VAE_iwae_10.pkl`

(c) `submission/VAE_iwae_100.pkl`

(d) `submission/VAE_iwae_1000.pkl`

To check the accuracy of your results, run:

```
python grader.py 3c-0-basic
python grader.py 3c-1-basic
python grader.py 3c-2-basic
python grader.py 3c-3-basic
```

(d) **[13.50 points (Coding)]** As IWAE only requires the averaging of multiple unnormalized density ratios, the IWAE bound is also applicable to the GMVAE model. Repeat parts 2 and 3 for the GMVAE by implementing the `negative_iwae_bound` function in `gmvae.py`.

Run

```
python main.py --model gmvae --iwae
```

to evaluate your implementation against the test subset.

All four IWAE bounds for GMVAE will be reported to:

(a) `submission/GMVAE_iwae_1.pkl`

(b) `submission/GMVAE_iwae_10.pkl`

(c) `submission/GMVAE_iwae_100.pkl`

(d) `submission/GMVAE_iwae_1000.pkl`

To check the accuracy of your results, run:

```
python grader.py 3d-0-basic
python grader.py 3d-1-basic
python grader.py 3d-2-basic
python grader.py 3d-3-basic
```

# 4 Implementing the Semi-Supervised VAE (SSVAE)

So far we have dealt with generative models in the unsupervised setting. We now consider semi-supervised learning on the MNIST dataset, where we have a small number of labeled $\mathbf{x}_\ell = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{100}$ pairs in our training data and a large amount of unlabeled data $\mathbf{x}_u = \{\mathbf{x}^{(i)}\}_{i=101}^{60000}$. A label $y^{(i)}$ for an image is simply the number the image $\mathbf{x}^{(i)}$ represents. We are interested in building a classifier that predicts the label$y$ given the sample $\mathbf{x}$. One approach is to train a classifier using standard approaches using only the labeled data. However, we would like to leverage the large amount of unlabeled data that we have to improve our classifier's performance.

We will use a latent variable generative model (a VAE), where the labels $y$ are partially observed, and $\mathbf{z}$ are always unobserved. The benefit of a generative model is that it allows us to naturally incorporate unlabeled data into the maximum likelihood objective function simply by marginalizing $y$ when it is unobserved. We will implement the Semi-Supervised VAE (SSVAE) for this task, which follows the generative process specified below:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z} \mid 0, I) \tag{16}$$

$$p(y) = \text{Categorical}(y \mid \pi) = \frac{1}{10} \tag{17}$$

$$p_\theta(\mathbf{x} \mid y, \mathbf{z}) = \text{Bern}\left(\mathbf{x} \mid f_\theta\left(y, \mathbf{z}\right)\right) \tag{18}$$

where $\pi = (1/10, ..., 1/10)$ is a fixed uniform prior over the 10 possible labels and each sequence of pixels $\mathbf{x}$ is modeled by a Bernoulli random variable parameterized by the output of a neural network decoder $f_\theta(\cdot)$.

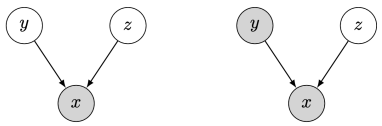



Figure 1: Graphical model for SSVAE. Gray nodes denote observed variables; unshaded nodes denote latent variables. **Left**: SSVAE for the setting where the labels $y$ are unobserved; **Right**: SSVAE where some data points $(x, y)$ have observed labels.

To train a model on the datasets $\mathbf{X}_\ell$ and $\mathbf{X}_u$, the principle of maximum likelihood suggests that we find the model $p_\theta$ which maximizes the likelihood over both datasets. Assuming the samples from $\mathbf{X}_\ell$ and $\mathbf{X}_u$ are drawn i.i.d., this translates to the following objective:

$$\max_\theta \sum_{x \in \mathbf{X}_u} \log p_\theta(\mathbf{x}) + \sum_{\mathbf{x}, y \in \mathbf{X}_\ell} \log p_\theta(\mathbf{x}, y) \tag{19}$$

where

$$p_\theta(\mathbf{x}) = \sum_{y \in \mathcal{Y}} \int p_\theta(\mathbf{x}, y, \mathbf{z}) d\mathbf{z} \tag{20}$$

$$p_\theta(\mathbf{x}, y) = \int p_\theta(\mathbf{x}, y, \mathbf{z}) d\mathbf{z} \tag{21}$$

To overcome the intractability of exact marginalization of the latent variables $\mathbf{z}$, we will instead maximize their respective evidence lower bounds,

$$\max_{\theta, \phi} \sum_{x \in \mathbf{X}_u} \text{ELBO}(\mathbf{x}; \theta, \phi) + \sum_{\mathbf{x}, y \in \mathbf{X}_\ell} \text{ELBO}(\mathbf{x}, y; \theta, \phi) \tag{22}$$

where we introduce some amortized inference model $q_\phi(y, \mathbf{z} \mid \mathbf{x}) = q_\phi(y \mid \mathbf{x}) q_\phi(\mathbf{z} \mid \mathbf{x}, y)$. Specifically,

$$q_\phi(y \mid \mathbf{x}) = \text{Categorical}\left(y \mid f_\phi\left(\mathbf{x}\right)\right) \tag{23}$$

$$q_\phi(\mathbf{z} \mid \mathbf{x}, y) = \mathcal{N}\left(\mathbf{z} \mid \mu_\phi\left(\mathbf{x}, y\right), \text{diag}\left(\sigma^2_\phi\left(\mathbf{x}, y\right)\right)\right) \tag{24}$$

where the parameters of the Gaussian distribution are obtained through a forward pass of the encoder. We note that $q_\phi(y \mid \mathbf{x}) = \text{Categorical}(y \mid f_\phi(\mathbf{x}))$ is actually an MLP classifier that is also a part of the inference model, and it predicts the probability of the label $y$ given the observed data $\mathbf{x}$.

We use this amortized inference model to construct the ELBOs.

$$\text{ELBO}(\mathbf{x}; \theta, \phi) = \mathbb{E}_{q_\phi(y, \mathbf{z} \mid \mathbf{x})}\left[\log \frac{p_\theta\left(\mathbf{x}, y, \mathbf{z}\right)}{q_\phi\left(y, \mathbf{z} \mid \mathbf{x}\right)}\right] \tag{25}$$

$$\text{ELBO}(\mathbf{x}, \mathbf{y}; \theta, \phi) = \mathbb{E}_{q_\phi(\mathbf{z} \mid \mathbf{x}, y)}\left[\log \frac{p_\theta\left(\mathbf{x}, y, \mathbf{z}\right)}{q_\phi\left(\mathbf{z} \mid \mathbf{x}, y\right)}\right] \tag{26}$$

However, Kingma et al. (2014)[1] observed that maximizing the lower bound of the log-likelihood is not sufficient to learn a good classifier. Instead, they proposed to introduce an additional training signal that directly trains the classifier on the labeled data

$$\max_{\theta,\phi} \sum_{x \in \mathbf{X}_u} \text{ELBO}(\mathbf{x}; \theta, \phi) + \sum_{\mathbf{x}, y \in \mathbf{X}_\ell} \text{ELBO}(\mathbf{x}, y; \theta, \phi) + \alpha \sum_{\mathbf{x}, y \in \mathbf{X}_\ell} \log q_\phi(y \mid \mathbf{x}) \tag{27}$$

where $\alpha \geq 0$ weights the importance of the classification accuracy. In this problem, we will consider a simpler variant of this objective that works just as well in practice,

$$\max_{\theta,\phi} \sum_{x \in \mathbf{X}} \text{ELBO}(\mathbf{x}; \theta, \phi) + \alpha \sum_{\mathbf{x}, y \in \mathbf{X}_l} \log q_\phi(y \mid \mathbf{x}) \tag{28}$$

where $\mathbf{X} = \{\mathbf{X}_u, \mathbf{X}_\ell\}$. It is worth noting that the introduction of the classification loss has a natural interpretation as maximizing the ELBO subject to the soft constraint that the classifier $q_\phi(y \mid \mathbf{x})$ (which is a component of the amortized inference model) achieves good performance on the labeled dataset. This approach of controlling the generative model by constraining its inference model is thus a form of amortized inference regularization[2].

(a) **[3 points (Coding)]** Run

```
python main.py --model ssvae --train --gw 0
```

To use GPU acceleration run the command below.

```
python main.py --model ssvae --train --gw 0 --device gpu
```

The `gw` flag denotes how much weight to put on the ELBO(x) term in the objective function; scaling the term by zero corresponds to a traditional supervised learning setting on the small labeled dataset only, where we ignore the unlabeled data. Your classification accuracy on the test set after the run completes (30000 iterations) will be reported to `submission/SSVAE_0.pkl`. To check your accuracy. run:

```
python grader.py 4a-0-basic
```

[1] Kingma, et al. Semi-Supervised Learning With Deep Generative Models. Neural Information Processing Systems, 2014

[2] Shu, et al. Amortized Inference Regularization. Neural Information Processing Systems, 2018.

(b) **[7 points (Coding)]** Implement the `negative_elbo_bound` function in `ssvae.py`. Note that the function expects as output the negative Evidence Lower Bound as well as its decomposition into the following terms,

$$-\text{ELBO}(\mathbf{x};\theta,\phi) = -\mathbb{E}_{q_\phi(y|\mathbf{x})} \log \frac{p(y)}{q_\phi(y \mid \mathbf{x})} - \mathbb{E}_{q_\phi(y|\mathbf{x})}\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x},y)} \left( \log \frac{p\left(\mathbf{z}\right)}{q_\phi\left(\mathbf{z} \mid \mathbf{x}, y\right)} + \log p_\theta\left(\mathbf{x} \mid \mathbf{z}, y\right) \right) \tag{29}$$

$$\underbrace{D_{\text{KL}}\left(q_\phi\left(y \mid \mathbf{x}\right) \mid\mid p\left(y\right)\right)}_{\text{KL}_y} + \underbrace{\mathbb{E}_{q_\phi(y|\mathbf{x})} D_{\text{KL}}\left(q_\phi\left(\mathbf{z} \mid \mathbf{x}, y\right) \mid\mid p\left(\mathbf{z}\right)\right)}_{\text{KL}_z} + \underbrace{\mathbb{E}_{q_\phi(y,\mathbf{z}|\mathbf{x})}[-\log p_\theta(\mathbf{x} \mid \mathbf{z}, y)]}_{\text{Reconstruction}} \tag{30}$$

Since there are only ten labels, we shall compute the expectations with respect to $q_\phi(y \mid \mathbf{x})$ exactly, while using a single Monte Carlo sample of the latent variables $\mathbf{z}$ sampled from each $q_\phi(\mathbf{z} \mid \mathbf{x}, y)$ when dealing with the reconstruction term. In other words, we approximate the negative ELBO with

$$D_{\text{KL}}\left(q_\phi\left(y \mid \mathbf{x}\right) \mid\mid p\left(y\right)\right) + \sum_{y \in \mathcal{Y}} q_\phi(y \mid \mathbf{x}) \left[ D_{\text{KL}}\left(q_\phi\left(\mathbf{z} \mid \mathbf{x}, y\right) \mid\mid p\left(\mathbf{z}\right)\right) - \log p_\theta\left(\mathbf{x} \mid \mathbf{z}^{(y)}, y\right) \right] \tag{31}$$

where $\mathbf{z}^{(y)} \sim q_\phi(\mathbf{z} \mid \mathbf{x}, y)$ denotes a sample from the inference distribution when conditioned on a possible $(\mathbf{x}, y)$ pairing. The functions `kl_normal` and `kl_cat` in `utils.py` will be useful.

(c) **[3 points (Coding)]** Run

```
python main.py --model ssvae --train
```

To use GPU acceleration run the command below.

```
python main.py --model ssvae --train --device gpu
```

This will run the SSVAE with the ELBO(x) term included, and thus perform semi-supervised learning. Your classification accuracy on the test set after the run completes will be saved to `submission/SSVAE_1.pkl`. Your accuracy on the test set should be $> 90\%$. To check your accuracy, run:

```
python grader.py 4c-0-basic
```

**Note 1:** When training the SSVAE model, if you experience an out-of-control growth on the $KL_z$ term, please check the inputs into `ut.kl_cat`. Try double checking if your inputs are correct (especially the third term).

**Note 2:** If you report a drop of accuracy when training it is most likely as a result of issue with reshaping or summing over the wrong dimension. Make sure you are doing those correctly.

# 5 Bonus: Style and Content Disentanglement in SVHN

A curious property of the SSVAE graphical model is that, in addition to the latent variables $y$ learning to encode the content (i.e. label) of the image, the latent variables $\mathbf{z}$ also learns to encode the *style* of the image. We shall demonstrate this phenomenon on the SVHN dataset. To make the problem simpler, we will only consider the *fully*-supervised scenario where $y$ is fully-observed. This yields the fully-supervised VAE, shown below in Figure 2.

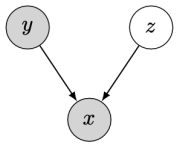



Figure 2: Graphical model for FSVAE. Gray nodes denote observed variables.

(a) **[5 points (Written, Extra Credit)]** Since fully-supervised VAE (FSVAE) always conditions on an observed $y$ in order to generate the sample $\mathbf{x}$, it is a conditional variational autoencoder. Derive the Evidence Lower Bound $\text{ELBO}(\mathbf{x}; \theta, \phi, y)$ of the conditional log probability $\log p_\theta(\mathbf{x} \mid y)$. You are allowed to introduce the amortized inference model $q_\phi(\mathbf{z} \mid \mathbf{x}, y)$.

(b) **[5 points (Coding, Extra Credit)]** Implement the `negative_elbo_bound` function in `fsvae.py`. In contrast to the MNIST dataset, the SVHN dataset has a *continuous* observation space

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z} \mid 0, I) \tag{32}$$

$$p_\theta(\mathbf{x} \mid y, \mathbf{z}) = \mathcal{N}\left(\mathbf{x} \mid \mu_\theta\left(y, \mathbf{z}\right), \text{diag}\left(\sigma_\theta^2\left(y, \mathbf{z}\right)\right)\right) \tag{33}$$

To simplify the problem more, we shall assume that the *variance is fixed* at

$$\text{diag}\left(\sigma_\theta^2\left(y, \mathbf{z}\right)\right) = \frac{1}{10} I \tag{34}$$

and only train the decoder mean function $\mu_\theta$. Once you have implemented `negative_elbo_bound`, run

```
python main.py --model fsvae --train
```

To use GPU acceleration run the command below.

```
python main.py --model fsvae --train --device gpu
```

The default settings will use a max iteration of 1 million. We suggest checking the image quality of clip $(\mu_\theta\,(\text{y,z}))$—where clip$(\cdot)$ performs element-wise clipping of outputs outside the range [0, 1]—every 10k iterations and stopping the training whenever the digit classes are recognizable[3].

Once you have learned a sufficiently good model, you can then generate twenty latent variables $\mathbf{z}^{(0)}, ..., \mathbf{z}^{(19)} \overset{\text{i.i.d.}}{\sim} p(\mathbf{z})$ by running

```
python main.py --model fsvae
```

This will work to generate 200 SVHN digits where the digit in the $i^{\text{th}}$ row, $j^{\text{th}}$ column (assuming zero-indexing) is the image clip$(\mu_\theta(\text{y} = i, \text{z} = j))$. The samples (at 50000 iterations) will look like the image below:

[3] An important note about sample quality inspection when modeling continuous images using VAEs with Gaussian observation decoders: modeling continuous image data distributions is quite challenging. Rather than truly sampling $x \sim p_\theta(\mathbf{x} \mid y)$, a common heuristic is to simply sample clip$(\mu_\theta(y, z))$ instead