CS 236 Homework 2 Solutions

Instructors: Stefano Ermon and Aditya Grover

{ermon,adityag}@cs.stanford.edu

Available: 10/15/2018; Due: 23:59 PST, 10/29/2018

Problem 1: Implementing the Variational Autoencoder (VAE) (25 points)

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 $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z}) d\mathbf{z}$. Specifically, the VAE is defined by the following generative process:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, I)$$

$$p_\theta(\mathbf{x}|\mathbf{z}) = \text{Bern}(\mathbf{x}|f_\theta(\mathbf{z}))$$

In other words, we assume that the latent variables $\mathbf{z}$ are sampled from a unit Gaussian distribution $\mathcal{N}(\mathbf{z}|0, I)$. The latent $\mathbf{z}$ are then passed through a neural network decoder $f_\theta(\cdot)$ to obtain the parameters 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 $p_\theta(\mathbf{x}) = \int p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z}) d\mathbf{z}$ 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:

$$q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\mu_\phi(\mathbf{x}), \text{diag}(\sigma^2_\phi(\mathbf{x})))$$

Specifically, we pass each image $\mathbf{x}$ through a neural network which outputs the mean μ_ϕ and diagonal covariance $\text{diag}(\sigma^2_\phi(\mathbf{x}))$ 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):

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

Notice that the ELBO as shown on the right hand side of the above expression decomposes into two terms: (1) **the reconstruction loss**: $-\mathbb{E}_{q_\phi(\mathbf{z})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$, and (2) **the Kullback-Leibler (KL) term**: $D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})))$.

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

1. **[5 points]** 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 $q_\phi(\mathbf{z}|\mathbf{x})$ and return a sample $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$.

Solution:
Code:

```
def sample_gaussian(m, v):
    eps = torch.randn_like(v)
    z = m + torch.sqrt(v) * eps
    return z
```

2. **[5 points]** 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 $\left\{\mathbf{x}^{(i)}\right\}_{i=1}^{n}$, 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

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

where $\mathbf{z}^{(1)} \sim q_\phi(\mathbf{z}|\mathbf{x})$ 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 KL divergence.

Solution:
Code:

```
def negative_elbo_bound(self, x):
    m, v = self.enc.encode(x)
    z = ut.sample_gaussian(m, v)
    logits = self.dec.decode(z)

    kl = ut.kl_normal(m, v, self.z_prior[0], self.z_prior[1])
    rec = -ut.log_bernoulli_with_logits(x, logits)
    nelbo = kl + rec

    nelbo, kl, rec = nelbo.mean(), kl.mean(), rec.mean()
    return nelbo, kl, rec
```

3. **[10 points]** To test your implementation, run `python run_vae.py` to train the VAE. Once the run is complete (20000 iterations), it will output (assuming your implementation is correct): the average (1) negative ELBO, (2) KL term, and (3) reconstruction loss as evaluated on a test subset that we have selected. Report the three numbers you obtain as part of the write-up. 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. (Hint: the negative ELBO on the test subset should be somewhere around 100.)

Solution:
Standard deviations are provided (based on 50 runs). We provide numbers computed on both the CPU and GPU. CPU numbers are slightly better since the test subset is slightly different.

GPU Numbers:

(a) VAE negative ELBO: 101.21 ± 0.61

(b) VAE KL: 19.34 ± 0.19

(c) VAE Rec: 81.86 ± 0.66

CPU Numbers:

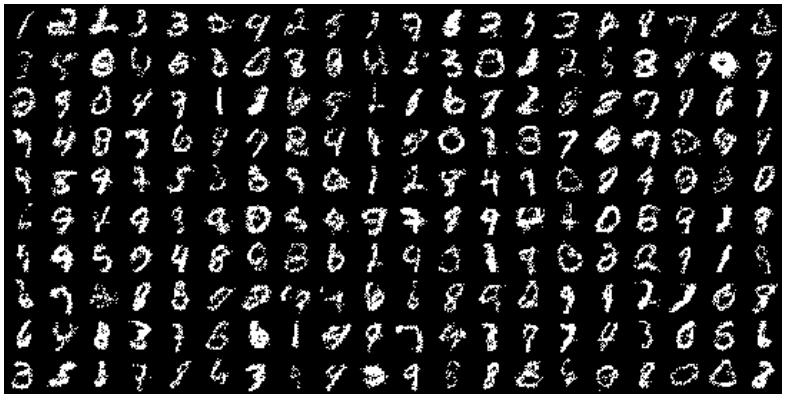
(a) VAE negative ELBO: 99.20 ± 0.56

(b) VAE KL: 19.35 ± 0.18

(c) VAE Rec: 79.84 ± 0.61

4. **[5 points]** Visualize 200 digits (generate a single image tiled in a grid of 10×20 digits) sampled from $p_\theta(\mathbf{x})$.

Solution:
Visualization of VAE samples



Problem 2: Implementing the Mixture of Gaussians VAE (GMVAE) (30 points)

Recall that in Problem 1, the VAE's prior distribution was a parameter-free isotropic Gaussian $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, I)$. 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:

$$p_\theta(\mathbf{z}) = \sum_{i=1}^{k} \frac{1}{k} \mathcal{N}(\mathbf{z}|\mu_i, \text{diag}(\sigma_i^2))$$

where $i \in \{1, \ldots, k\}$ denotes the i^{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 θ. 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:

$$q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\mu_\phi(\mathbf{x}), \text{diag}(\sigma_\phi^2(\mathbf{x}))$$
$$p_\theta(\mathbf{x}|\mathbf{z}) = \text{Bern}(\mathbf{x}|f_\theta(\mathbf{z}))$$

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

$$\begin{aligned} D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z})) &\approx \log q_\phi(\mathbf{z}^{(1)}|\mathbf{x}) - \log p_\theta(\mathbf{z}^{(1)}) \\ &= \log \mathcal{N}(\mathbf{z}^{(1)}|\mu_\phi(\mathbf{x}), \text{diag}(\sigma_\phi^2(\mathbf{x})) - \log \sum_{i=1}^{k} \frac{1}{k} \mathcal{N}(\mathbf{z}^{(1)}|\mu_i, \text{diag}(\sigma_i^2)), \end{aligned}$$

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

1. **[15 points]** 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.

Solution:

Code:

```
def log_normal(x, m, v):
    element_wise = -0.5 * (torch.log(v) + (x - m).pow(2) / v + np.log(2 * np.pi))
    log_prob = element_wise.sum(-1)
    return log_prob
```

```
def log_normal_mixture(z, m, v):
    # (batch, dim) -> (batch, 1, dim)
    z = z.unsqueeze(1)
    # (batch, 1, dim) -> (batch, mix, dim) -> (batch, mix)
    log_prob = log_normal(z, m, v)
    # (batch, mix) -> (batch,)
    log_prob = log_mean_exp(log_prob, dim=1)
    return log_prob
```

```python
def negative_elbo_bound(self, x):
    # Prior
    prior = ut.gaussian_parameters(self.z_pre, dim=1)

    m, v = self.enc.encode(x)
    z = ut.sample_gaussian(m, v)
    logits = self.dec.decode(z)

    kl = ut.log_normal(z, m, v) - ut.log_normal_mixture(z, *prior)
    rec = -ut.log_bernoulli_with_logits(x, logits)
    nelbo = kl + rec

    nelbo, kl, rec = nelbo.mean(), kl.mean(), rec.mean()
    return nelbo, kl, rec
```

2. **[10 points]** To test your implementation, run `python run_gmvae.py` to train the GMVAE. 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. Report the three numbers you obtain as part of the write-up. 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.

Solution:

Standard deviations are provided (based on 50 runs). We provide numbers computed on both the CPU and GPU. CPU numbers are slightly better since the test subset is slightly different.

GPU Numbers:

(a) GMVAE negative ELBO: 98.58 ± 0.46

(b) GMVAE KL: 17.82 ± 0.18

(c) GMVAE Rec: 80.77 ± 0.51

CPU Numbers:

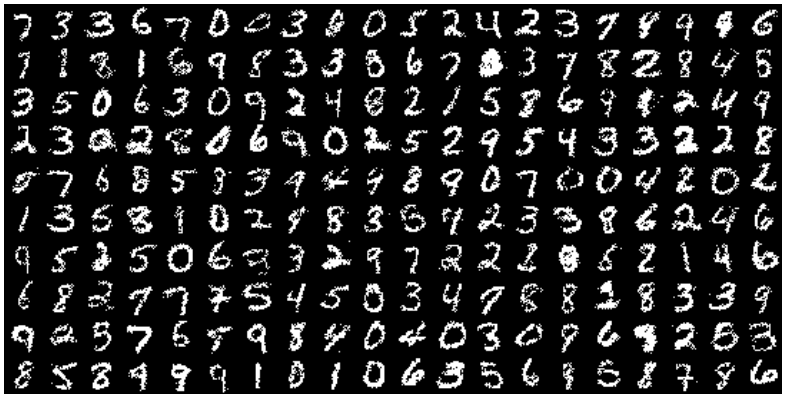
(a) GMVAE negative ELBO: 96.75 ± 0.56

(b) GMVAE KL: 17.78 ± 0.20

(c) GMVAE Rec: 78.96 ± 0.59

3. **[5 points]** Visualize 200 digits (generate a single image tiled in a grid of 10×20 digits) sampled from $p_\theta(\mathbf{x})$.

Solution:
Visualization of GMVAE samples



Problem 3: Implementing the Importance Weighted Autoencoder (IWAE) (25 points)

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

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

where $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$. As can be seen from the RHS, the density ratio is *unnormalized* since the density ratio is 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 $q_\phi(\mathbf{z}|\mathbf{x})$ to obtain the following IWAE bound:

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

Notice that for the special case of $m = 1$, the IWAE objective reduces to the standard ELBO.

1. **[5 points]** Prove that IWAE is a valid lower bound of the log-likelihood, and that the ELBO lower bounds IWAE

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

   for any $m \geq 1$. [Hint: consider Jensen's Inequality]

   Solution:

   A step-by-step proof. The crucial steps are (3) and (4), where we apply Jensen's Inequality. Identifying these two steps is sufficient for full-credit.

$$\begin{aligned}
\log p_\theta(x) &= \log \left(\frac{1}{m} \sum_{i=1}^{m} \mathbb{E}_{\mathbf{z}^{(i)} \sim q_\phi(\mathbf{z}|\mathbf{x})} \frac{p_\theta(\mathbf{x}, \mathbf{z}^{(i)})}{q_\phi(\mathbf{z}^{(i)} \mid \mathbf{x})} \right) && (1) \\
&= \log \left(\mathbb{E}_{\mathbf{z}^{(1)} \ldots \mathbf{z}^{(m)} \overset{\text{i.i.d.}}{\sim} q_\phi(\mathbf{z}|\mathbf{x})} \frac{1}{m} \sum_{i=1}^{m} \frac{p_\theta(\mathbf{x}, \mathbf{z}^{(i)})}{q_\phi(\mathbf{z}^{(i)} \mid \mathbf{x})} \right) && (2) \\
&\geq \mathbb{E}_{\mathbf{z}^{(1)} \ldots \mathbf{z}^{(m)} \overset{\text{i.i.d.}}{\sim} q_\phi(\mathbf{z}|\mathbf{x})} \left(\log \frac{1}{m} \sum_{i=1}^{m} \frac{p_\theta(\mathbf{x}, \mathbf{z}^{(i)})}{q_\phi(\mathbf{z}^{(i)} \mid \mathbf{x})} \right) = \mathcal{L}_m(\mathbf{x}) && (3) \\
&\geq \mathbb{E}_{\mathbf{z}^{(1)} \ldots \mathbf{z}^{(m)} \overset{\text{i.i.d.}}{\sim} q_\phi(\mathbf{z}|\mathbf{x})} \left(\frac{1}{m} \sum_{i=1}^{m} \log \frac{p_\theta(\mathbf{x}, \mathbf{z}^{(i)})}{q_\phi(\mathbf{z}^{(i)} \mid \mathbf{x})} \right) && (4) \\
&= \frac{1}{m} \sum_{i=1}^{m} \mathbb{E}_{\mathbf{z}^{(i)} \sim q_\phi(\mathbf{z}|\mathbf{x})} \left(\log \frac{p_\theta(\mathbf{x}, \mathbf{z}^{(i)})}{q_\phi(\mathbf{z}^{(i)} \mid \mathbf{x})} \right) && (5) \\
&= \frac{1}{m} \sum_{i=1}^{m} \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \left(\log \frac{p_\theta(\mathbf{x}, \mathbf{z})}{q_\phi(\mathbf{z} \mid \mathbf{x})} \right) && (6) \\
&= \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \left(\log \frac{p_\theta(\mathbf{x}, \mathbf{z})}{q_\phi(\mathbf{z} \mid \mathbf{x})} \right) = \mathcal{L}_1(\mathbf{x}). && (7)
\end{aligned}$$

2. **[5 points]** 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.

   Solution:

   Code:

```python
def negative_iwae_bound(self, x, iw):
    m, v = self.enc.encode(x)

    # Duplicate
    m = ut.duplicate(m, iw)
    v = ut.duplicate(v, iw)
    x = ut.duplicate(x, iw)
    z = ut.sample_gaussian(m, v)
    logits = self.dec.decode(z)

    kl = ut.log_normal(z, m, v) - ut.log_normal(z, self.z_prior[0], self.z_prior[1])
    # Important: it is technically incorrect to calculate KL analytically
    # IWAE bound requires that we calculate the importance sample weight
    # So do not do: kl = ut.kl_normal(m, v, self.z_prior[0], self.z_prior[1])
    rec = -ut.log_bernoulli_with_logits(x, logits)
    nelbo = kl + rec
    niwae = -ut.log_mean_exp(-nelbo.reshape(iw, -1), dim=0)

    niwae, kl, rec = niwae.mean(), kl.mean(), rec.mean()
    return niwae, kl, rec
```

3. **[10 points]** Run `python run_vae.py --train 0` to evaluate your implementation against the test subset. 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. Report all four IWAE bounds for this write-up.

Solution:

Standard deviations are provided (based on 50 runs). We provide numbers computed on both the CPU and GPU. CPU numbers are slightly better since the test subset is slightly different.

GPU Numbers:

(a) VAE negative IWAE-1: 101.21 ± 0.61

(b) VAE negative IWAE-10: 98.50 ± 0.51

(c) VAE negative IWAE-100: 97.46 ± 0.50

(d) VAE negative IWAE-1000: 96.93 ± 0.45

CPU Numbers:

(a) VAE negative IWAE-1: 99.20 ± 0.56

(b) VAE negative IWAE-10: 96.52 ± 0.46

(c) VAE negative IWAE-100: 95.54 ± 0.41

(d) VAE negative IWAE-1000: 95.07 ± 0.40

4. [**5 points**] 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`. Compare and contrast IWAE bounds for GMVAE and VAE.

Solution:
Code:

```
def negative_iwae_bound(self, x, iw):
    # Prior
    prior = ut.gaussian_parameters(self.z_pre, dim=1)

    m, v = self.enc.encode(x)

    # Duplicate
    m = ut.duplicate(m, iw)
    v = ut.duplicate(v, iw)
    x = ut.duplicate(x, iw)
    z = ut.sample_gaussian(m, v)
    logits = self.dec.decode(z)

    kl = ut.log_normal(z, m, v) - ut.log_normal_mixture(z, *prior)
    rec = -ut.log_bernoulli_with_logits(x, logits)
    nelbo = kl + rec
    niwae = -ut.log_mean_exp(-nelbo.reshape(iw, -1), dim=0)

    niwae, kl, rec = niwae.mean(), kl.mean(), rec.mean()
    return niwae, kl, rec
```

Standard deviations are provided (based on 50 runs). We provide numbers computed on both the CPU and GPU. CPU numbers are slightly better since the test subset is slightly different.

GPU Numbers:

(a) GMVAE negative IWAE-1: 98.58 ± 0.46
(b) GMVAE negative IWAE-10: 96.12 ± 0.42
(c) GMVAE negative IWAE-100: 95.16 ± 0.42
(d) GMVAE negative IWAE-1000: 94.71 ± 0.39

CPU Numbers:

(a) GMVAE negative IWAE-1: 96.75 ± 0.56
(b) GMVAE negative IWAE-10: 94.30 ± 0.48
(c) GMVAE negative IWAE-100: 93.43 ± 0.46
(d) GMVAE negative IWAE-1000: 92.98 ± 0.41

Problem 4: Implementing the Semi-Supervised VAE (SSVAE) (20 points)

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:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, I)$$

$$p(y) = \text{Categorical}(y|\pi) = \frac{1}{10}$$

$$p_\theta(\mathbf{x}|y, \mathbf{z}) = \text{Bern}(\mathbf{x}|f_\theta(y, \mathbf{z}))$$

where $\pi = (1/10, \ldots, 1/10)$ 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 $f_\theta(\cdot)$.

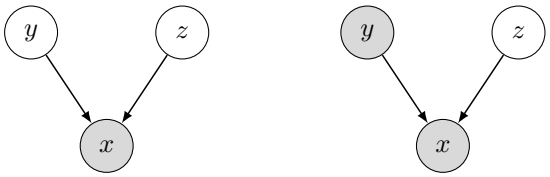


Figure 1: Graphical model for SSVAE. Gray nodes denote observed variables; unshaded nodes denote latent variables. **Left:** SSVAE for the setting where the labels y are unobserved; **Right:** SSVAE where some data points (x, y) have observed labels.

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_θ 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

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

where

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

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

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

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

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

$$q_\phi(y|\mathbf{x}) = \text{Categorical}(y|f_\phi(\mathbf{x}))$$

$$q_\phi(\mathbf{z}|\mathbf{x}, y) = \mathcal{N}(\mathbf{z}|\mu_\phi(\mathbf{x}, y), \text{diag}(\sigma_\phi^2(\mathbf{x}, y)))$$

where the parameters of the Gaussian distribution are obtained through a forward pass of the encoder. We note that $q_\phi(y|\mathbf{x}) = \text{Categorical}(y|f_\phi(\mathbf{x}))$ 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.

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

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

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

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,

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

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 $q_\phi(y|\mathbf{x})$ (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]

1. **[1 point]** Run `python run_ssvae.py --gw 0`. 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. Report your classification accuracy on the test set after the run completes (30000 iterations).

   Solution:
   Classifier accuracy using only the limited labeled data: 73.5%

2. **[10 points]** 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,

$$\begin{aligned}-\text{ELBO}(\mathbf{x};\theta,\phi) &= -\mathbb{E}_{q_\phi(y|\mathbf{x})} \log \frac{p(y)}{q_\phi(y|\mathbf{x})} - \mathbb{E}_{q_\phi(y|\mathbf{x})}\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x},y)}\left(\log \frac{p(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x},y)} + \log p_\theta(\mathbf{x}|\mathbf{z},y)\right) \\ &= \underbrace{D_{\text{KL}}(q_\phi(y|\mathbf{x})\|p(y))}_{\text{KL}_y} + \underbrace{\mathbb{E}_{q_\phi(y|\mathbf{x})} D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x},y)\|p(\mathbf{z}))}_{\text{KL}_z} + \underbrace{\mathbb{E}_{q_\phi(y,\mathbf{z}|\mathbf{x})}\left[-\log p_\theta(\mathbf{x}|\mathbf{z},y)\right]}_{\text{Reconstruction}}.\end{aligned}$$

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

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

   where $\mathbf{z}^{(y)} \sim q_\phi(\mathbf{z}|\mathbf{x},y)$ 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.

[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.

Solution:
Code:

```
def negative_elbo_bound(self, x):
    y_logits = self.cls.classify(x)
    y_logprob = F.log_softmax(y_logits, dim=1)
    y_prob = torch.softmax(y_logprob, dim=1) # (batch, y_dim)

    # Duplicate y based on x's batch size. Then duplicate x
    y = np.repeat(np.arange(self.y_dim), x.size(0))
    y = x.new(np.eye(self.y_dim)[y])
    x = ut.duplicate(x, self.y_dim)

    m, v = self.enc.encode(x, y)
    z = ut.sample_gaussian(m, v)
    x_logits = self.dec.decode(z, y)

    kl_y = ut.kl_cat(y_prob, y_logprob, np.log(1.0 / self.y_dim))
    kl_z = ut.kl_normal(m, v, self.z_prior[0], self.z_prior[1])       # (y_dim * batch)
    rec = -ut.log_bernoulli_with_logits(x, x_logits)                  # (y_dim * batch)

    # Compute the expected reconstruction and kl (under the distribution q(y | x))
    rec = (y_prob.t() * rec.reshape(self.y_dim, -1)).sum(0)           # (batch,)
    kl_z = (y_prob.t() * kl_z.reshape(self.y_dim, -1)).sum(0)         # (batch,)

    # Reduce to means
    kl_y, kl_z, rec = kl_y.mean(), kl_z.mean(), rec.mean()
    nelbo = rec + kl_z + kl_y
    return nelbo, kl_z, kl_y, rec
```

3. **[9 points]** Run `python run_ssvae.py`. This will run the SSVAE with the ELBO(x) term included, and thus perform semi-supervised learning. Report your classification accuracy on the test set after the run completes.

   SSVAE Accuracy: $93.52 \pm 1.09\%$

Bonus: Style and Content Disentanglement in SVHN (10 points)

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.

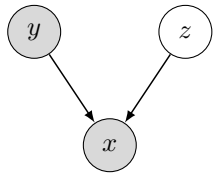


Figure 2: Graphical model for FSVAE. Gray nodes denote observed variables.

1. **[3 points]** Since fully-supervised VAE (FSVAE) always conditions on an observed y in order to generate the sample $\mathbf{x}$, it is a special case of the *conditional* variational autoencoder. Derive the Evidence Lower Bound $\text{ELBO}(\mathbf{x}; \theta, \phi, y)$ of the conditional log probability $\log p_\theta(\mathbf{x}|y)$. You are allowed to introduce the amortized inference model $q_\phi(\mathbf{z}|\mathbf{x}, y)$.

   Solution:

   The only difference from the vanilla VAE is that we condition on the additional information y. The prior $p(\mathbf{z})$ is not conditioned on y since y does not point to $\mathbf{z}$ in the generative model.

$$\log p_\theta(\mathbf{x} \mid y) \geq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x},y)} \left(\log \frac{p(\mathbf{z}) p_\theta(\mathbf{x} \mid \mathbf{z}, y)}{q_\phi(\mathbf{z} \mid \mathbf{x}, y)} \right) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x},y)} \left[\log p_\theta(\mathbf{x} \mid \mathbf{z}, y) \right] - D_{\mathrm{KL}}(q_\phi(\mathbf{z} \mid \mathbf{x}, y) \| p(\mathbf{z})). \quad (8)$$

2. **[7 points]** Implement the `negative_elbo_bound` function in `fsvae.py`. In contrast to the MNIST dataset, the SVHN dataset has a *continuous* observation space

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, I)$$

$$p_\theta(\mathbf{x}|y, \mathbf{z}) = \mathcal{N}(\mathbf{x}|\mu_\theta(y, \mathbf{z}), \text{diag}(\sigma^2_\theta(y, \mathbf{z}))).$$

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

$$\text{diag}(\sigma^2_\theta(y, \mathbf{z})) = \frac{1}{10} I,$$

   and only train the decoder mean function μ_θ. Once you have implemented `negative_elbo_bound`, run `python run_fsvae.py`. The default settings will use a max iteration of 1 million. We suggest checking the image quality of $\text{clip}(\mu_\theta(\text{y}, \text{z}))$—where $\text{clip}(\cdot)$ 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, generate twenty latent variables $\mathbf{z}^{(0)}, \ldots, \mathbf{z}^{(19)} \overset{\text{i.i.d.}}{\sim} p(\mathbf{z})$. Then, generate 200 SVHN digits (a single image tiled in a grid of 10×20 digits) where the digit in the i^{th} row, j^{th} column (assuming zero-indexing) is the image $\text{clip}(\mu_\theta(\text{y} = i, \text{z} = j))$.

   Solution:

   Code:

[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 $x \sim p_\theta(\mathbf{x}|y)$, a common heuristic is to simply sample $\text{clip}(\mu_\theta(\text{y}, \text{z}))$ instead.

```
def negative_elbo_bound(self, x, y):
    m, v = self.enc.encode(x, y)
    z = ut.sample_gaussian(m, v)
    x_mean = self.dec.decode(z, y)

    kl_z = ut.kl_normal(m, v, self.z_prior[0], self.z_prior[1]).mean()
    rec = -ut.log_normal(x, x_mean, 0.1 * torch.ones_like(x_mean)).mean()
    nelbo = kl_z + rec
    return nelbo, kl_z, rec
```

Visualization of FSVAE samples