CS 236 Winter 2023
Deep Generative Models
Variational Auto Encoders (VAE)

November 13, 2023

SUNet ID: jchan7
Name: Jason Chan

1 Motivation

1.1 Background

Latent models allow us to define complex models $p(x)$ in terms of simple building blocks $p(x|z)$, which is natural for unsupervised learning tasks (clustering, unsupervised representation learning etc). But there's no free lunch because it's much more difficult to learn than fully observable autoregressive models because $p(x)$ is hard to evaluate and optimise.

1.2 Latent Variables

The number of latent features z in a Variational Auto Encoder (VAE) is a design choice, which affects the model's capacity, the structure of the latent space, the quality of the generated samples and the training dynamics. When setting up a VAE, it's a balance between having sufficient dimensions to capture dasta complexity and not have so many that the model overfits or becomes intractable.

1.3 VAE Architecture

Autoencoders are compression algorithms. Autoencoders learn compressed representations to reconstruct the input data with minimal loss. They are deterministic. VAEs add a probabilistic spin, to learn the parameters of the probability distribution of the latent variables. The goal is not only to represent and reconstruct but to also generate new, similar data.

To generate new data from the VAE decoder, we sample specific values for z from the latent space distributions, which then generate x. This random sampling step leads to different reconstructions.

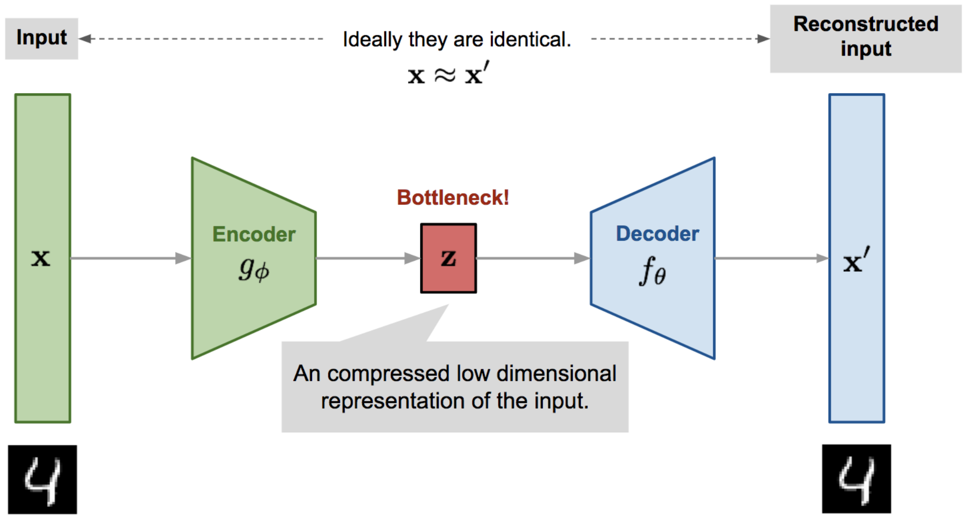

Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Figure 1: Image credits https://lilianweng.github.io/posts/2018-08-12-vae/

1.4 Marginal Probability in VAE Decoders

The encoder (stochastically) compresses all the information in a dataset x into latent variables, z. The decoder evaluates the marginal probability.

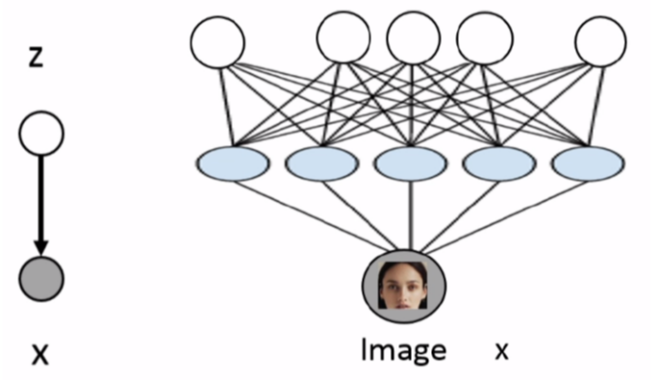
$$p_\theta(x) = \int p_\theta(x, z)dz \tag{1}$$

The term $p_\theta(x)$ represents the marginal likelihood of the data point x because it's computed by summing (or integrating) over all possible latent configurations that could produce x. This is where the integral comes in.

1.5 Intuition about the Marginal Probability

Why is the computation of the marginal probability integral vital in the VAE framework? At its core, the VAE tries to model our observed data x by considering all possible latent configurations z that might have produced x. This integral effectively sums over every conceivable scenario in the latent space that could explain our observed data. During training, our goal is to find the parameters of the VAE (both in the decoder and the encoder) that maximize the likelihood of our training dataset. To achieve this, we must understand how changes in these parameters influence the marginal probability. Thus, assessing this integral becomes essential at every optimization step for each data point x in our training set.

Imagine if our dataset were images of a faces. The dataset x would be a high dimensional vector, pixel intensities for each pixel. The latent space z represent features such as lighting angle, facial expression, tilt of head, lighting intensity. In this case, the number of latent variables is four. The VAE encoder tries to learn a way to convert each photo into a point on the latent space. The VAE decoder learns to do the reverse, for any four data points in the four dimensional latent space, generate a face photo. Now say you're given a new photo, x. The integral over z represents the sum of probabilities over all possible latent space configurations that could've produced this photo, which means considering every possible combination of lighting, intensity, facial expression and head tilt that could've produced the photo. Checking all these combinations is a daunting task.



1.6 Challenge of Integrating over all Latent Variables

But the marginal probability $p_\theta(x) = \int p_\theta(x, z)dz$ is expensive to compute because we need to enumerate all the z that could have generated x. Recall that the number of latent variables is a design choice. If the number of latent variables is one then the integral term needs to sum over all values of that latent variable, if the number of z is two then it's a double integral, if three, then it's a triple integral and so on. Imagine if z were discrete with 10 binary dimensions then that's already $2^{10} = 1024$ possible configurations to sum over.

2 The VAE foundations

2.1 First principles

1. Express the joint probability via the chain rule $p(x,z) = p(x|z)p(z)$ and substitute into the marginal probability equation
$$p_\theta(x) = \int p_\theta(x,z)dz = \int p(x|z)p(z)dz$$
2. We know the components of the marginal probability because these are also design choices
   (a) Declare that $z \sim \mathcal{N}(0, I)$. This is called the parameter-free isotropic Gaussian. $p(z)$ could also be learned parameters, see GMVAE.
   (b) $p(x|z) = \mathcal{N}(\mu_\theta(z), \Sigma_\theta(z))$ where μ_θ and Σ_θ are neural networks.
3. But we have two problems the stops us from evaluating $p(x)$
   (a) The first is that even though $p(x|z)$ is simple, the marginal p(x) is still very complex because of the integral term.
   (b) Furthermore, the problem formulation is incomplete The latent variables z, which represents our features are no accounted for.

2.2 Intuition for $p(z)$

Recall that the number of latent variables, z is a design choice. Each z is assumed to be a univariate Gaussian, which collectively form a multivariate Gaussian.

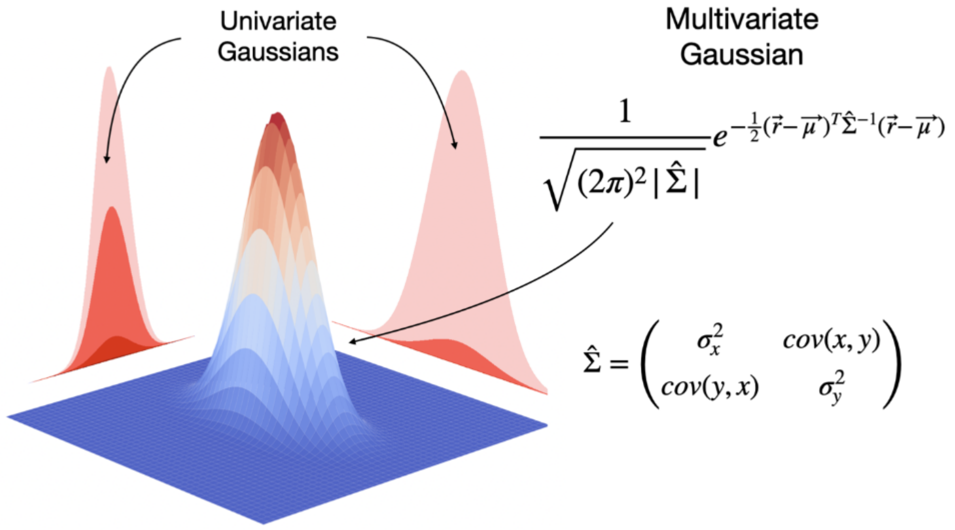

Univariate Gaussians
Multivariate Gaussian
$\frac{1}{\sqrt{(2\pi)^2|\hat{\Sigma}|}} e^{-\frac{1}{2}(\vec{r}-\vec{\mu})^T\hat{\Sigma}^{-1}(\vec{r}-\vec{\mu})}$
$\hat{\Sigma} = \begin{pmatrix} \sigma_x^2 & cov(x,y) \\ cov(y,x) & \sigma_y^2 \end{pmatrix}$

Figure 2: Image credit: https://www.hashpi.com/visualizing-a-multivariate-gaussian

3 Formulating the VAE objective function

To account for the inferred latent variables from observed data, we introduce the true posterior $p_\theta(z|x)$. It's called the **posterior** because it represents our beliefs about latent variable z after seeing the data x. It's intractable because Bayes' theorem $p_\theta(z|x) = \frac{p_\theta(x|z)p(z)}{p_\theta(x)}$, shows us that the marginal probability $p_\theta(x)$ is in the denominator, which takes us back to square one. So instead we solve the approximation of $p_\theta(z|x)$ using $q_\phi(z|x)$ using ELBO. This is the **variational inference** in VAE.

3.1 Modelling $q_\phi(z|x)$ as Multivariate Gaussian

We design $q_\phi(z|x)$ as Multivariate Gaussian. The VAE encoder is a neural network whose inputs are data x and output are the parameters of the Multivariate Gaussian: mean vector μ and log variance vector $log\sigma^2$, which both have the same dimension as our latent space.

3.2 Introduce $q_\phi(z|x)$ to the Marginal Distribution

1. To introduce our approximated posterior $q_\theta(z|x)$ we return to the marginal probability of x formula and multiply and divide by $q_\phi(z|x)$ and rearrange the numerator.

$$\begin{aligned} p_\theta(x) &= \int p_\theta(x|z)p(z)\frac{q_\phi(z|x)}{q_\phi(z|x)}dz \\ &= \int q_\phi(z|x)\frac{p_\theta(x|z)p(z)}{q_\phi(z|x)}dz \\ log(p_\theta(x)) &= log\Big(\int q_\phi(z|x)\frac{p_\theta(x|z)p(z)}{q_\phi(z|x)}dz\Big) \end{aligned}$$

   We recognise that the inner term is an expectation with respect to $q_\theta(z|x)$, which is simply saying the average over all possible values of the latent variable z's, as denoted by the integral with respect to dz.

$$log(p_\theta(x)) = log\Big(\mathbb{E}_{q_\phi(z|x)}\Big[\frac{p_\theta(x|z)p(z)}{q_\phi(z|x)}\Big]\Big)$$

   The formulation into an expectation is critical because it allows us to apply **Jensen's inequality**, which states that for a concave function like log (recall that log in machine learning is actually natural log, ln)

$$\begin{aligned}
log(\mathbb{E}[f(X)] &\geq \mathbb{E}[log f(X)] \\
log(\mathbb{E}_{q_\phi(z|x)}[p_\theta(x)]) &= log(p_\theta(x) \int q_\phi(z|x)dz = p_\theta(x) \cdot 1 = p_\theta(x)) \\
&\geq \mathbb{E}_{q_\phi(z|x)}\Big[log\Big(\frac{p_\theta(x|z)p(z)}{q_\phi(z|x)}\Big)\Big] \\
&= \mathbb{E}_{q_\phi(z|x)}\Big[log(p_\theta(x|z)) + log(p(z)) - log(q_\phi(z|x))\Big]
\end{aligned}$$

We know that KL divergence is given by $D_{KL}(q||p) = \mathbb{E}_q[log(q) - log(p)]$, if we flip the sign we get $-D_{KL}(q||p) = \mathbb{E}_q[log(p) - log(q)]$.

$$log(p_\theta(x)) \geq \mathbb{E}_{q_\phi(z|x)}\Big[log(p_\theta(x|z))\Big] - D_{KL}(q_\phi(z|x)||p(z))$$

3.2.1 Intuition for the VAE objective function

We want to maximise the probability of p(x), therefore we maximise ELBO. In PyTorch, objective functions are formulated as minimisation, so maximising ELBO is the same and minimising negative ELBO.

1. The first term is called the **reconstruction loss** but it's not a *loss* in the conventional sense of a before and after. It asks: for the latent variables that are likely given our observed data, how probable is it that our generative model produces the observed data? In VAE implementations when x is binary , $p_\theta(x|z)$ is Bernoulli and the negative of the reconstruction term is the equivalent of the binary cross-entropy loss between the data and its reconstruction, which is a more traditional measure of reconstruction quality.

2. The second term is considered the **regularisation**, which is the divergence between the estimated posterior and $p(z)$. Remember that KL=0 means the distributions are identical, so in the VAE objective formula a negative KL means we want the lowest KL possible to maximise the $log(p_\theta(x))$. We need KL to find $q_\phi(z|x)$ that fully describes the entire dataset x, not just a specific data point x, which is effectively prevents overfitting. This KL term penalizes the approximate posterior from deviating too much from the prior, so as the VAE learns, it's encouraged to keep the distributions of the latent variables z given the data close to a standard normal distribution.

4 VAE Variations

1. β-VAE. This adds a constant parameter to the DL term. If β is large (greater than 1) the the KL penalty is higher, which encourages the estimated posterior distribution to be more similar to p(z).

2. Gaussian Mixture VAE (GMVAE). In the GMVAE ELBO, an analytical solution from KL cannot be easily computed because $q_\phi(z|x)$ is a multivariate Gaussian, while $p_\theta(z)$ is a mixture of Gaussian. Recall that the KL term resides in an expectation, which means integrating over all z values. Instead, KL is estimated from an *unbiased estimator via monte carlo sampling.* The term unbiased means there's no bias in the sample, which is the case for monte carlo. In the implementation, only one sample is drawn via monte carlo because it's a student homework.

3. Importance Weighted Auto Encoder (IWAE). The key idea behind IWAE is to use $m > 1$ samples from the approximate posterior $q_\phi(z|x))$ to obtain the IWAE bound. The purpose is to improve ELBO in the event $q_\phi(z|x)$ is a poor approximation of the true posterior $p_\theta(z|x)$, by interrogating the unnormalized density ratio.

$$\frac{p_\theta(x,z)}{q_\phi(z|x)} = \frac{p_\theta(x|z)p_\theta(x)}{q_\phi(z|x)} \tag{2}$$

   Since the density ratio is un-normalized, we can obtain a tighter bound by averaging multiple un-normalized density ratios.

$$l_m(x;\theta,\phi) = \mathbb{E}_{z^{(1)},..,z^{(m)}\sim q_\phi(z|x)}\Big[log\frac{1}{m}\sum_{i=1}^{m}\frac{p_\theta(x,z^{(i)})}{q_\phi(z^{(i)}|x)}\Big] \tag{3}$$

   Notice for the special case when m=1, the IWAE objective reduces to the standard ELBO $\mathbb{E}_{q_\phi(z|x)}\big[log\frac{p_\theta(x,z)}{q_\phi(z|x)}\big]$

4. Self Supervised VAE (SSVAE), where a small percentage 100/60000 of dataset is labelled. We want to build a classifier that predicts the label y give sample x. We build a VAE where the labels are partially observed and z are always unobserved. This lets us partially incorporate unlabeled data into the MLE objective function by marginalizing y when it's unobserved.

   To train a model on datasets $X_{labelled}$ and $X_{unlabelled}$, MLE suggests we find the model p_θ that **maximises the likelihood over both datasets**. Assuming the samples from $X_labelled$ and $X_unlabelled$ are drawn i.i.d. then our objective is

$$\max_\theta \sum_{x\in X_{unlabelled}} logp_\theta(x) + \sum_{x,y\in X_{labelled}} logp_\theta(x,y) \tag{4}$$

   Where

$$p_\theta(x) = \sum_{y\in\mathcal{Y}}\int p_\theta(x,y,z)dz \tag{5}$$

$$p_\theta(x, y) = \int p_\theta(x, y, z) dz \tag{6}$$

The first formula gives you the probability of x over all possible y and z. The second formula gives you the probability of x for a specific y, but over all possible z. To overcome intractability of exact marginalization of the latent variables, z, we instead maximise their respective ELBOs.

$$\max_{\theta,\phi} = \sum_{x \in \mathcal{X}_{unlabelled}} ELBO(x; \theta, \phi) + \sum_{x,y \in \mathcal{X}_{labelled}} ELBO(x, y; \theta, \phi) \tag{7}$$

We introduce some amortized inference model $q_\phi(y, z|x) = q_\phi(y|x)q_\phi(z|x, y)$ Specifically

$$q_\phi(y|x) = Categorical(y|f_\phi(x)) \tag{8}$$

$$q_\phi(z|x, y) = \mathcal{N}(z|\mu_\phi(x, y), diag(\sigma^2_\phi(x, y))) \tag{9}$$

w

5 Implementing VAE in code

This is a summary of Problem Set 2 of Stanford's CS236 2023 course. There are four VAE implementations.

1. Problem 1 A VAE was implemented to learn a probabilistic model of the MNIST dataset. $p(z)$ is a parameter free, isotropic Gaussian, $\mathcal{N}(z|0, I)$. The inputs are a sequence of binary pixels with k=10 latent variables. The latent variables are sampled from a unit Gaussian, and passed to the decoder neural network to obtain the logits for the d Bernoulli random variables, which models the pixels in each image.
2. Problem 2 implements a Gaussian Mixture VAE (GMVAE) for $p(z)$ and evaluates NELBO for GMVAE, which is different to the NELBO for VAE in the regularization term because it compares a multivariate Gaussian vs. a Gaussian mixture.
3. Problem 3 implements the Importance Weighted Auto Encoder (IWAE) for the previous VAE and GMVAE questions.
4. Problem 4 implements a Self Supervised VAE (SSVAE) for MNIST

$$p(z) = \mathcal{N}(z|0, I) \tag{10}$$

$$p(y) = Categorial(y|\pi) = \frac{1}{10} \tag{11}$$

$$p_\theta(x|y, z) = Bern(x|f_\theta(y, z)) \tag{12}$$

   where π is a fixed uniform prior over the 10 possible labels and each sequence of pixels x is modelled by a Bernoulli random variable

5.1 Reparameterisation Trick For Latent Variable Sampling

z is a random variable, which doesn't work in neural network backpropagation so

```python
def sample_gaussian(m, v):
    """
    Element-wise application reparameterization trick to sample from Gaussian

    Args:
        m: tensor: (batch, ...): Mean
        v: tensor: (batch, ...): Variance

    Return:
        z: tensor: (batch, dim): Samples
    """
    std = torch.sqrt(v)
    eps = torch.randn_like(std)
    z = m + std * eps

    return z
```

5.2 NELBO

```python
def negative_elbo_bound(self, x):
        """
        Computes the Evidence Lower Bound, KL and, Reconstruction costs

        Args:
            x: tensor: (batch, dim): Observations

        Returns:
            nelbo: tensor: (): Negative evidence lower bound
            kl: tensor: (): ELBO KL divergence to prior
            rec: tensor: (): ELBO Reconstruction term
        """
        m, v = self.enc(x)
        z = ut.sample_gaussian(m, v)
        logits = self.dec(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
```

5.3 GMVAE NELBO

Recall $KL(q||p) = \mathbb{E}_q[log(q) - log(p)]$, which requires the log probability of a multivariate gaussian minus the log probability of a mixture of gaussians.

```python
def negative_elbo_bound(self, x):
        """
        Computes the Evidence Lower Bound, KL and, Reconstruction costs

        Args:
            x: tensor: (batch, dim): Observations

        Returns:
            nelbo: tensor: (): Negative evidence lower bound
            kl: tensor: (): ELBO KL divergence to prior
            rec: tensor: (): ELBO Reconstruction term
        """
        # Get prior p(z) params
        prior = ut.gaussian_parameters(self.z_pre, dim=1)
        m_prior, v_prior = prior # unpack

        # Get posterior p(z|x) params
        m_post, v_post = self.enc(x)

        # Reconstruction loss
        z = ut.sample_gaussian(m_post, v_post)
        logits = self.dec(z)
        rec = -ut.log_bernoulli_with_logits(x, logits)

        # Regularization KL(q(z|x)||p(z))
        kl = ut.log_normal(z, m_post, v_post) - ut.log_normal_mixture(z, m_prior, v_prior)

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

5.4 Log probability $q(z|x)$ for Multivariate Gaussian

Used by GMVAE NELBO. For a multivariate Gaussian distribution with mean m and diagonal covariance matrix Σ, the formula for log probability $\log(q(z|x))$ is

$$\log q(z|x; m, v) = -\frac{1}{2}\sum_{i=1}^{D}\left(\log(2\pi v_i) + \frac{(z_i - m_i)^2}{v_i}\right) \tag{13}$$

Note: the implementation has input parameter, x. In our case, we are evaluating p(z—x), so x_i corresponds to sampled z_i.

```python
def log_normal(x, m, v):
    """
    Computes the elem-wise log probability of a Gaussian and then sum over the
    last dim. Basically we're assuming all dims are batch dims except for the
    last dim.

    Args:
        x: tensor: (batch_1, batch_2, ..., batch_k, dim): Observation
        m: tensor: (batch_1, batch_2, ..., batch_k, dim): Mean
        v: tensor: (batch_1, batch_2, ..., batch_k, dim): Variance

    Return:
        log_prob: tensor: (batch_1, batch_2, ..., batch_k): log probability of
            each sample. Note that the summation dimension is not kept
    """
    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
```

5.5 Log probability $q(z)$ for Mixture of Gaussian

For mixture of Gaussians, the formula for the log probability $\log(p(x))$ the below where K is the number of Gaussian, and π is the weight ratio.

$$q(z) = \sum_{j=1}^{K} \pi_j \cdot p(x|m_j, v_j)$$

$$\log q(z) = \log\left(\sum_{j=1}^{K} \pi_j \cdot e^{\log p(x|m_j, v_j)}\right)$$

```python

def log_normal_mixture(z, m, v):
    """
    Computes log probability of Gaussian mixture.

    Args:
        z: tensor: (batch, dim): Observations
        m: tensor: (batch, mix, dim): Mixture means
        v: tensor: (batch, mix, dim): Mixture variances

    Return:
        log_prob: tensor: (batch,): log probability of each sample
    """
    z = z.unsqueeze(1)
    log_prob = log_normal(z, m, v)
    log_prob = log_mean_exp(log_prob , dim=1)
    return log_prob
```

5.6 Negative IWAE Bound VAE

```python
    def negative_iwae_bound(self, x, iw):
        """
        Computes the Importance Weighted Autoencoder Bound
        Additionally, we also compute the ELBO KL and reconstruction terms

        Args:
            x: tensor: (batch, dim): Observations
            iw: int: (): Number of importance weighted samples

        Returns:
            niwae: tensor: (): Negative IWAE bound
            kl: tensor: (): ELBO KL divergence to prior
            rec: tensor: (): ELBO Reconstruction term
        """
        ################################################################################
        # TODO: Modify/complete the code here
        # Compute niwae (negative IWAE) with iw importance samples, and the KL
        # and Rec decomposition of the Evidence Lower Bound
        #
        # Outputs should all be scalar
        ################################################################################
        m, v = self.enc(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(z)
        kl = ut.log_normal(z, m, v) - ut.log_normal(z, 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()
        ################################################################################
        # End of code modification
        ################################################################################
        return niwae, kl, rec
```

5.7 NELBO for SSVAE

```python
        def negative_elbo_bound(self, x):
        """
        Computes the Evidence Lower Bound, KL and, Reconstruction costs

        Args:
            x: tensor: (batch, dim): Observations

        Returns:
            nelbo: tensor: (): Negative evidence lower bound
            kl: tensor: (): ELBO KL divergence to prior
            rec: tensor: (): ELBO Reconstruction term
        """
        y_logits = self.cls(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
        # This enumerates all possible combination of x with labels (0, 1, ..., 9)
        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(x, y)
        z = ut. sample_gaussian(m, v)
        x_logits = self.dec(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])
        rec = -ut. log_bernoulli_with_logits(x, x_logits)

        rec = (y_prob.t() * rec.reshape(self.y_dim, -1)).sum(0)
        kl_z = (y_prob.t() * kl_z.reshape(self.y_dim, -1)).sum(0)

        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

    def kl_cat(q, log_q, log_p):
    """
    Computes the KL divergence between two categorical distributions

    Args:
        q: tensor: (batch, dim): Categorical distribution parameters
        log_q: tensor: (batch, dim): Log of q
        log_p: tensor: (batch, dim): Log of p

    Return:
        kl: tensor: (batch,) kl between each sample
    """
    element_wise = q * (log_q - log_p)
    kl = element_wise.sum(-1)
    return kl
```

5.8 Encoder

Take a specific x^i, which we compress into $\hat{z}$ using $\hat{z} = q_\phi(z|x^i)$.

1. Start with simple prior $z \sim \mathcal{N}(0, 1) = p(z)$
2. Transform via $q_\phi(z|x) = \mathcal{N}(\mu_\theta(z), \Sigma_\theta(z))$

5.9 Decoder

The decoder tries to reconstruct the data x, given z.

$$p(x|z) = \mathcal{N}(\mu_\theta(z), \Sigma_\theta(z))$$