Variational autoencoders

- Unsupervised learning
- Autoencoders
- Extensions of autoencoders
- Variational autoencoders
- Formulation of encoder and decoder distributions
- Derivation of evidence lower-bound

Unsupervised learning

There may be several scenarios where we wish to do *unsupervised learning*. Unsupervised learning has to do with learning key features or structure in the data without labels.

This involves learning key features or structure in the data. There are simple linear techniques to do this that you may be familiar with (e.g., PCA, FA) as well as some nonlinear techniques (e.g., LLE, tSNE). Here, we'll talk about autoencoders and variational autoencoders, which use neural networks.

Unsupervised learning

Both autoencoders and variational autoencoders (VAEs) produce a lower-dimensional representation of the input data. If the input is $\mathbf{x} \in \mathbb{R}^n$, an autoencoder will produce a $\mathbf{h} \in \mathbb{R}^d$ where $d < n$, which is designed to contain most of the important features of $\mathbf{x}$ to reconstruct it.

The key difference between an autoencoder and a VAE is that the latter is probabilistic. In this manner, the VAE is also a *generative model*. Because the VAE is probabilistic, it is possible to obtain samples $\mathbf{x} \in \mathbb{R}^n$ by sampling from its distribution.

Generative models

VAEs are in a class of models called *generative models*. These models do exactly what their name suggests: they can be used to generate examples of input data by learning their statistics.

Concretely, if you trained a VAE to learn features from some data $\mathbf{x} \in \mathbb{R}^n$ (e.g., the $\mathbf{x}$ could comprise images from CIFAR-10) then the VAE would be able to generate new samples of images that share similar statistics to CIFAR-10.

Autoencoders

You got exposure to a basic autoencoder in HW #3. The basic autoencoder is set up as follows:

- **Encoder**: Perform a dimensionality reduction step on the data, $\mathbf{x} \in \mathbb{R}^n$ to obtain features $\mathbf{h} \in \mathbb{R}^d$.
- **Decoder**: Map the features $\mathbf{h} \in \mathbb{R}^d$ to closely reproduce the input, $\hat{\mathbf{x}} \in \mathbb{R}^n$.

Thus, the autoencoder implements the following problem:

Let $\mathbf{x} \in \mathbb{R}^n$, $f(\cdot) : \mathbb{R}^n \to \mathbb{R}^d$ and $g(\cdot) : \mathbb{R}^d \to \mathbb{R}^n$. Let

$$\hat{\mathbf{x}} = g(f(\mathbf{x}))$$

Define a loss function, $\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}})$, and minimize $\mathcal{L}$ with respect to the parameters of $f(\cdot)$ and $g(\cdot)$.

There are different loss functions that you could consider, but a common one is the squared loss:

$$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) = \|\mathbf{x} - \hat{\mathbf{x}}\|^2$$

PCA as an autoencoder

PCA can be implemented as an autoencoder when $\mathbf{h} = f(\mathbf{x}) = \mathbf{W}^T\mathbf{x}$ and $g(\mathbf{h}) = \mathbf{W}\mathbf{h}$, with $\mathbf{W} \in \mathbb{R}^{n \times d}$. Then, the loss function is

$$\mathcal{L} = \left\| \mathbf{x} - \mathbf{W}\mathbf{W}^T\mathbf{x} \right\|^2$$

In this case, one will learn a $\mathbf{W}$ that has the following key property:

If $\mathbf{W} = \mathbf{U}\Sigma\mathbf{V}^T$ is the singular value decomposition of $\mathbf{W}$, then $\mathbf{U}$ will span the same subspace as the top d eigenvectors of $\mathbf{cov}(\mathbf{x})$.

- Proof of this result is covered in ECE C143A/C243A when we talk about probabilistic PCA.
- Hence, $\mathbf{W}$ spans the same subspace as that found via performing PCA on $\mathbf{x}$.

Nonlinear dimensionality reduction

In general, however, one may implement $f(\mathbf{x})$ and $g(\mathbf{x})$ as neural networks. In their simplest form, for some activation function (e.g., a $\tanh$), one could implement $f(\mathbf{x}) = \tanh(\mathbf{W}^T\mathbf{x})$ and $g(\mathbf{h}) = \tanh(\mathbf{W}\mathbf{h})$. Hence, both $f()$ and $g()$ are single layer neural networks.

In a more general form, $f()$ and $g()$ could be deep neural networks, learning potentially more nonlinear and expressive features $\mathbf{h}$.

In these scenarios, it is straightforward to train with stochastic gradient descent; all gradients can be calculated straightforwardly via backpropagation.

What to do after learning $f()$?

After learning $f()$, there are several things one can do:

- It is possible to visualize the lower-dimensional features to infer structure about the underlying data.
- It is possible to use $f(\mathbf{h})$, rather than $\mathbf{x}$ as the input features to a softmax or svm classifier.

The autoencoder may be extended in several ways. Here, we outline a few.

- Sparse autoencoders
- Denoising autoencoders
- Contractive autoencoders
- Variational autoencoders (majority of these notes)

Sparse autoencoders

A sparse autoencoder is one that is regularized to not only minimize the loss, but to also incorporate sparse features. If $\mathbf{h} = f(\mathbf{x})$ and $\hat{\mathbf{x}} = \mathbf{g}(\mathbf{h})$, then the sparse encoder has the following loss:

$$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) + \lambda \sum_i |h_i|$$

where h_i is the ith element of $\mathbf{h}$. This is intuitive, as we know L1-regularization introduces sparsity.

Denoising autoencoders

Say you wanted to obtain an autoencoder that was robust to noise. One could generate noise, ε, and add it to the input $\mathbf{x}$, so that $\tilde{\mathbf{x}} = \mathbf{x} + \varepsilon$. Then, the loss function of the autoencoder would have loss:

$$\mathcal{L}(\mathbf{x}, g(f(\tilde{\mathbf{x}})))$$

and it would learn to denoise $\tilde{\mathbf{x}}$ to reproduce $\mathbf{x}$. This can cause your autoencoder to be robust to certain types of noise. Alain and Bengio also show that this type of training causes f and g to learn structure of $p(\mathbf{x})$.

Contractive autoencoders

Another consideration in autoencoder design is that the features be *robust*, that is, that a small change in $\mathbf{x}$ produces only a small (and not a large) change in $\mathbf{h}$. One way to incorporate this idea is to regularize the Frobenius norm of the Jacobian matrix, $\nabla_{\mathbf{x}}\mathbf{h}$. Then, the loss function becomes:

$$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) + \lambda \left\| \nabla_{\mathbf{x}}\mathbf{h} \right\|_F^2$$

This autoencoder is called *contractive* because the features can be seen as a contraction of the inputs performed by $f(\mathbf{x})$ in the neighborhood of examples from the data-generating distributions $p(\mathbf{x})$.

Variational autoencoders

Variational autoencoders (VAEs) were introduced by Kingma and Welling (2014). A fairly good tutorial is provided by Doersch (2016).

VAEs, instead of learning $f()$ and $g()$, learn distributions of the features given the input and the input given the activations, i.e., probabilistic versions of $f()$ and $g()$. Concretely, the VAE will learn:

- $p(\mathbf{h}|\mathbf{x})$: the distribution of the features given the input.
- $p(\mathbf{x}|\mathbf{h})$: the distribution of the input given the features.

These distributions are parametrized by neural networks, meaning that they can express nonlinear transformations, and be trained via stochastic gradient descent.

For the rest of these notes, to stay consistent with typical VAE papers, we'll let $\mathbf{z}$ denote the features, i.e., $\mathbf{z} = \mathbf{h}$.

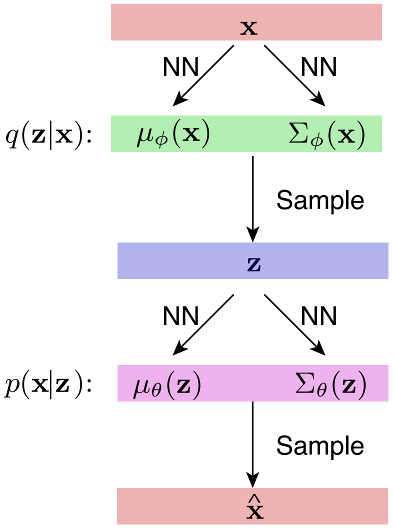
Why learn distributions?

Why might it be good to learn distributions?

- Often times the data is noisy, and a model of the distribution of the data is more useful for a given application.
- Further, often times the relationship between the observed variables and the latent variables is *nonlinear*, in which case the VAE provides a way to do inference.
- The VAE is a generative model; by learning $p(\mathbf{x}|\mathbf{z})$, it is possible to sample $\mathbf{z}$ and then sample $\mathbf{x}$. This enables the generation of data that has similar statistics to the input.

Conceptual diagram of the VAE

Below is a conceptual diagram of the VAE that we will arrive at. However, we put it here to give you a high-level intuition to start:



Conceptual diagram of the VAE (cont.)

This conceptual block diagram of the VAE looks very similar to an autoencoder, with one major change: instead of inferring $\mathbf{z}$ directly from $\mathbf{x}$ and $\hat{\mathbf{x}}$ directly from $\mathbf{z}$, we instead infer distributions.

- Hence, the inference changes from learning $\mathbf{z} = f(\mathbf{x})$ to learning $q(\mathbf{z}|\mathbf{x})$; and from learning $\hat{\mathbf{x}} = g(\mathbf{z})$ to learning $p(\mathbf{x}|\mathbf{z})$.
- Because we learn distributions, to obtain values for $\mathbf{z}$ and $\hat{\mathbf{x}}$, we instead *sample* from these learned distributions.
- In this manner, the VAE is a probabilistic version of the autoencoder.

Formulating the VAE

We'll approach the VAE from the context of a generative model. Say we want to generate samples from the data distribution, $p(\mathbf{x})$. Instead of inferring $p(\mathbf{x})$ directly, we can use what are called *latent variable models*.

Latent variable models model the data, $\mathbf{x}$, as arising from an unobserved (and hence latent) variable, $\mathbf{z}$. It may not be easy to model $p(\mathbf{x})$ directly, but it may be easier to choose some distribution $p(\mathbf{z})$ and instead model $p(\mathbf{x}|\mathbf{z})$.

Intuitively, this means that we model how $\mathbf{x}$ arises from some latent variables, $\mathbf{z}$, that themselves have their own variability. Concretely, $p(\mathbf{x}|\mathbf{z})$ can be defined by some mapping $\mathbf{x} = g(\mathbf{z})$. The prior distribution, $p(\mathbf{z})$ and the function $g()$ then define the distribution $p(\mathbf{x}|\mathbf{z})$.

Generating samples from $p(\mathbf{x})$

If we know $p(\mathbf{z})$ and $p(\mathbf{x}|\mathbf{z})$ then it is straightforward to generate samples from $p(\mathbf{x})$. In particular, we do the following repeatedly:

1. Generate a sample randomly from $\mathbf{z}_s \sim p(\mathbf{z})$. Call this sample $\mathbf{z}_s$.
2. Then generate a sample $\mathbf{x}_s \sim p(\mathbf{x}|\mathbf{z} = \mathbf{z}_s)$. This sample of $\mathbf{x}_s$ is from $p(\mathbf{x})$.

This is true because the sample from step 2 has probability:

$$p(\mathbf{x}, \mathbf{z}_s)$$

If we repeatedly do this, it is drawing samples repeatedly for all different sampled $\mathbf{z}_s$'s. If we call the ith sample $\mathbf{z}_s^{(i)}$ then the resulting density of the sampled $\mathbf{x}_s$'s is:

$$p(\mathbf{x}) \approx \frac{1}{n} \sum_{i=1}^{n} p(\mathbf{x}, \mathbf{z}_s^{(i)})$$

which approximates the distribution of $\mathbf{x}$.

Example: modeling a distribution with a latent variable

Say that we want to model

$$\mathbf{x} \sim \mathcal{N}(\mu, \Sigma)$$

where the parameters are $\mu \in \mathbb{R}^n$ and $\Sigma \in \mathbb{R}^{n \times n}$. We have data samples $\mathbf{x}^{(1)}, \mathbf{x}^{2}, \ldots, \mathbf{x}^{(m)}$. Say we want to learn these parameters and then be able to draw samples $\mathbf{x}$ from this distribution.

- To set this up via a latent variable model, we can define some latent variable $\mathbf{z}$.
- What should the distribution of $\mathbf{z}$ be?
- To answer this question, we can use the fact that if $\mathbf{z} \sim \mathcal{N}(\nu, \Lambda)$, then $\mathbf{x} = \mathbf{A}\mathbf{z} + \mathbf{b}$ has distribution:

  $$\mathbf{x} \sim \mathcal{N}(\mathbf{A}\nu + \mathbf{b}, \mathbf{A}\Lambda\mathbf{A}^T)$$

  (You showed that these are the means and covariances on HW #1.)
- Since $\mathbf{x}$ is normal, we can always set $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$ and define:

  $$\mathbf{x} = \mu + \Sigma^{1/2}\mathbf{z}$$

  to obtain samples $\mathbf{x} \sim \mathcal{N}(\mu, \Sigma)$. Here, $\Sigma^{1/2}$ corresponds to the Cholesky decomposition of Σ.

Designing the prior distribution, $\mathbf{p}(\mathbf{z})$

Our first design choice is to decide what $p(\mathbf{z})$ will be. In almost all cases, we will set $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$. Why can we do this? i.e., isn't this quite a stringent constraint on the distribution of the latent variables?

No, intuitively, we have two knobs here: we can set $p(\mathbf{z})$ and $p(\mathbf{x}|\mathbf{z})$. The point is that even if $\mathbf{z}$ has a simple distribution, if $\mathbf{x}|\mathbf{z}$ is expressive enough (e.g., if it incorporates nonlinear transformations) then we can put all of our work into optimizing $\mathbf{x}|\mathbf{z}$ to generate the distribution $p(\mathbf{x})$.

Said differently, we can generate nonlinear transformations that map a $\mathcal{N}(0, \mathbf{I})$ distribution to arbitrarily complex distributions. So instead, we let the expressivity of the nonlinear transform $\mathbf{x} = g(\mathbf{z})$ fit the data $p(\mathbf{x})$.

Modeling the distribution of $\mathbf{x}$

In a VAE, we will want to model the distribution of $\mathbf{x}$, but do so through a latent variable so that modeling these complex distributions is reformulated as nonlinear transformations on a simpler latent variable $\mathbf{z}$.

For example, imagine if $\mathbf{x}$ given $\mathbf{z}$ was normally distributed:

$$\mathbf{x}|\mathbf{z} \sim \mathcal{N}(\mu_\theta(\mathbf{z}), \Sigma_\theta(\mathbf{z}))$$

Hence, if we define $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$, we can draw a sample $\mathbf{x}_s$ by first drawing a sample $\mathbf{z}_s \sim p(\mathbf{z})$ and then applying then sampling $\mathbf{x}_s \sim p(\mathbf{x}|\mathbf{z}_s)$.

Here, $\mu_\theta(\mathbf{z})$ is a neural network that takes $\mathbf{z}$ and outputs a vector in $\mathbb{R}^n$ that represents the mean of $\mathbf{x}|\mathbf{z}$. Similarly, $\Sigma_\theta(\mathbf{z})$ is a neural network that takes $\mathbf{z}$ and outputs a matrix in $\mathbb{R}^{n\times n}$ that represents the covariance of $\mathbf{x}|\mathbf{z}$.

In this example, $\mathbf{x}|\mathbf{z}$ is normal, $\mathbf{z}$ is normal, but $\mathbf{x}$ is not normal. This is because $\mathbf{x}$ is not a linear transformation of $\mathbf{z}$ and in general, we have no guarantees on what the distribution of a a nonlinear transformation of a random variable is.

Finding parameters θ

How do we find the parameters θ? As in this class, we will follow our method of:

- Defining a loss function to optimize
- Calculating gradients of the loss function with respect to the parameters θ.
- Apply a variant of stochastic gradient descent.

The most natural loss function to optimize is to maximize the likelihood of observing the data, i.e., to maximize:

$$\prod_{i=1}^{m} p_\theta(\mathbf{x}^{(i)})$$

Maximizing the likelihood

Consider one example, $\mathbf{x}$. Then to maximize the likelihood, we have to calculate:

$$\begin{aligned} p_\theta(\mathbf{x}) &= \int p_\theta(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\ &= \int p_\theta(\mathbf{x}|\mathbf{z}) p(\mathbf{z}) d\mathbf{z} \end{aligned}$$

There are a few cases when we know how to do this integral.

- E.g., we know how to do this integral if $\mathbf{x}|\mathbf{z}$ and $\mathbf{z}$ are normal, and further if $\mathbf{x}$ is a linear function of $\mathbf{z}$. In this case, we can use other approaches (e.g., the expectation maximization algorithm) to optimize parameters.

In general, if $\mathbf{x}$ is a nonlinear function of $\mathbf{z}$ then the distribution, $p_\theta(\mathbf{x}, \mathbf{z})$ isn't something we can easily write analytically.

Thus, in general, calculating $p_\theta(\mathbf{x})$ is intractable and we thus cannot directly maximize the likelihood.

Monte Carlo samples of $p(\mathbf{x})$

One might wonder: even if we can't calculate $p(\mathbf{x})$ explicitly, can't I sample from it using the scheme we previously discussed?

Yes, we could sample this distribution by sampling $\mathbf{z}$ and then calculating $\mathbf{x}|\mathbf{z}$. However, this is really only tractable when $\mathbf{x}$ is relatively low-dimensional. Often times $\mathbf{x}$ is high-dimensional (e.g., it may be an image with $32 \times 32 \times 3$ values to represent). In these scenarios, we run into the curse of dimensionality, where we need to grab *many* samples to get an accurate view of $\mathbf{x}$.

So what now?

We need a loss function, so we know how to change our parameters to optimize it. However, we've run into a large problem at this point:

- We can't calculate $p(\mathbf{x})$.
- Hence, we can't write the maximum-likelihood objective.

As a result, we need a different approach / new objective function.

This leads to the VAE objective function, which is typically called the *evidence lower bound* or ELBO. The idea is as follows:

- If we can't write $\prod_i p(\mathbf{x}^{(i)})$, let's instead derive a lower bound on it.
- If the lower bound is tractable, then we can optimize the parameters with respect to the lower bound.
- If we are making the lower bound larger, we are making the likelihood larger.
- If you are familiar with the expectation maximization (EM) algorithm, this is a similar idea.

Arriving at the ELBO: defining a posterior

So far, we have the following formula to work with:

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}$$

If this is the only equation we have, we're in bad shape, since we don't know how to do this integral.

Hence, as always in machine learning, we also make use of our chain rule for probability:

$$p(\mathbf{z}|\mathbf{x})p(\mathbf{x}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$$

By using this formula, we have another way to represent $p(\mathbf{x})$, i.e., as:

$$p(\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})}$$

Arriving at the ELBO: defining a posterior

However, now we run into another problem: $p(\mathbf{z}|\mathbf{x})$ is also intractable. To get around this, we're going to make an approximation of this distribution, $q(\mathbf{z}|\mathbf{x})$, and incorporate into the loss function that this approximation should be close to $p(\mathbf{z}|\mathbf{x})$.

What form should $q(\mathbf{z}|\mathbf{x})$ take, so that it is expressive but also simple to work with? We take a similar approach to when we defined the likelihood $(\mathbf{x}|\mathbf{z}))$ and say that $\mathbf{z}|\mathbf{x}$ should be a nonlinear function (i.e., a neural network) that has normally distributed statistics (so we can still do algebra with it), i.e.:

$$q(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu_\phi(\mathbf{x}), \Sigma_\phi(\mathbf{x}))$$

KL divergence

The Kullback-Liebler divergence of two distributions, $q(\mathbf{z})$ and $p(\mathbf{z})$, is defined as:

$$\mathrm{KL}(q||p) = -\sum_{\mathbf{z}} q(\mathbf{z}) \log \frac{p(\mathbf{z})}{q(\mathbf{z})}$$

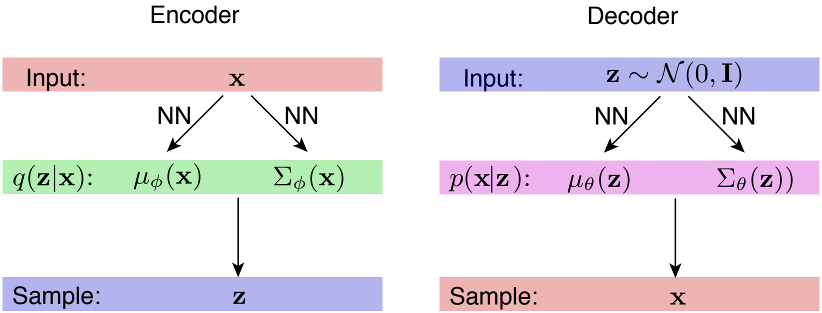
and is a measure of how similar the distributions $q(\mathbf{z})$ and $p(\mathbf{z})$ are. It has the following properties:

1. In general, KL divergence is not commutative, i.e., $\mathrm{KL}(q||p) \neq \mathrm{KL}(p||q)$.
2. $\mathrm{KL}(q||p) \geq 0$ for any q, p.
3. $\mathrm{KL}(q||p) = 0$ if and only if $q(\mathbf{z}) = p(\mathbf{z})$ for all $\mathbf{z}$, i.e., q and p are the same distribution.

Proof of these results is covered in ECE C143A/C243A.

Intuition of the VAE

At this point, our VAE has the following high-level construction (note: we haven't defined the ELBO yet, but will shortly).


Encoder
Input: $\mathbf{x}$
NN
NN
$q(\mathbf{z}|\mathbf{x})$: $\mu_\phi(\mathbf{x})$ $\Sigma_\phi(\mathbf{x})$
Sample: $\mathbf{z}$
Decoder
Input: $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$
NN
NN
$p(\mathbf{x}|\mathbf{z})$: $\mu_\theta(\mathbf{z})$ $\Sigma_\theta(\mathbf{z}))$
Sample: $\mathbf{x}$

- To sample a latent variable, we use the encoder network: take an input, $\mathbf{x}$, and calculate $q(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu_\phi(\mathbf{x}), \Sigma_\phi(\mathbf{x}))$. Then, sample $\mathbf{z} \sim q(\mathbf{z}|\mathbf{x})$.
- To sample an input, we draw our latent variable $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$. We then calculate $p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mu_\theta(\mathbf{x}), \Sigma_\phi(\mathbf{x}))$. Then, we sample $\mathbf{x} \sim p(\mathbf{x}|\mathbf{z})$.

Deriving the ELBO

With this intuition, we're prepared to derive the ELBO. For the sake of simplicity, we'll assume here $\mathbf{x}$ is a sample, $\mathbf{x}^{(i)}$. (i.e., for these slides, we'll calculate the ELBO for one example $\mathbf{x}^{(i)}$, but I will drop the superscript (i) for the sake of brevity.)

First, we note that:

$$\log p_\theta(\mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x})$$

We can do this because $\log p_\theta(\mathbf{x})$ has no dependence on $\mathbf{z}$; and so by the linearity of the expectation, $\mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}) = \log p_\theta(\mathbf{x}) \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} 1$, which is $\log p_\theta(\mathbf{x})$. For shorthand, we'll let $\mathbb{E}_{\mathbf{z}}$ denote $\mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}$.

Deriving the ELBO (cont.)

We continue from here:

$$\begin{aligned}
\log p_\theta(\mathbf{x}) &= \mathbb{E}_{\mathbf{z}} \log p_\theta(\mathbf{x}) \\
&\overset{(a)}{=} \mathbb{E}_{\mathbf{z}} \log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})} \\
&\overset{(b)}{=} \mathbb{E}_{\mathbf{z}} \log \left(\frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})} \frac{q(\mathbf{z}|\mathbf{x})}{q(\mathbf{z}|\mathbf{x})} \right) \\
&\overset{(c)}{=} \mathbb{E}_{\mathbf{z}} \log p(\mathbf{x}|\mathbf{z}) - \mathbb{E}_{\mathbf{z}} \log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} + \mathbb{E}_{\mathbf{z}} \log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})} \\
&\overset{(d)}{=} \mathbb{E}_{\mathbf{z}} \log p(\mathbf{x}|\mathbf{z}) - \mathrm{KL}(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) + \mathrm{KL}(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z}|\mathbf{x})) \\
&\geq \mathbb{E}_{\mathbf{z}} \log p(\mathbf{x}|\mathbf{z}) - \mathrm{KL}(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))
\end{aligned}$$

where we used the following facts: (a) is the chain rule for probability, (b) is multiplication by 1 expressed as $q(\mathbf{z}|\mathbf{x})/q(\mathbf{z}|\mathbf{x})$, (c) is expanding the logarithm and using the linearity of the expectation, (d) is using the definition of KL divergence. The final inequality comes from the fact that

$$\mathrm{KL}(q(\mathbf{z}|x)||p(\mathbf{z}|x) \geq 0$$

Notice that this term encapsulates our penalty in estimating the log-likelihood of the data from using $q(\mathbf{z}|\mathbf{x})$ to approximate $p(\mathbf{z}|\mathbf{x})$.

Deriving the ELBO (cont.)

This is then our lower bound on the log-likelihood:

$$\begin{aligned} \log p_\theta(\mathbf{x}) &\geq \mathbb{E}_{\mathbf{z}} \log p(\mathbf{x}|\mathbf{z}) - \mathrm{KL}(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) \\ &= \mathcal{L}_{\text{vae}}(\mathbf{x}) \end{aligned}$$

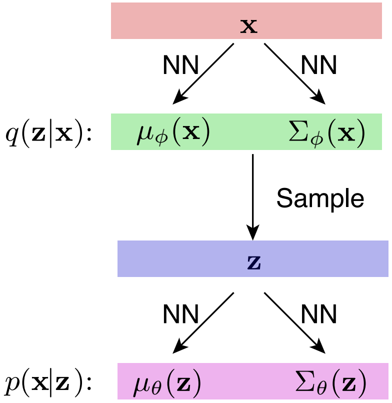
This is great news, because we now have a loss function (the lower bound, $\mathcal{L}_{\text{vae}}(\mathbf{x})$) that is tractable in the following manner:

- The term $\mathbb{E}_{\mathbf{z}} \log p(\mathbf{x}|\mathbf{z})$ can be approximated from data using a minibatch, almost exactly like we do for the softmax loss. We take some minibatch of m examples $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}$, and for each $\mathbf{x}^{(i)}$, we calculate $q(\mathbf{z}|\mathbf{x}^{(i)})$, sample $\mathbf{z}$ from it, and then calculate $\log p(\mathbf{x}^{(i)}|\mathbf{z})$.
- The KL divergence has a closed form when $q()$ and $p()$ are Gaussian distributions. That is,

$$\mathrm{KL}(\mathcal{N}(\mu_0, \Sigma_0), \mathcal{N}(\mu_1, \Sigma_1)) = \frac{1}{2}\left[\mathrm{tr}(\Sigma_1^{-1}\Sigma_0) + (\mu_1 - \mu_0)^T \Sigma_1^{-1}(\mu_1 - \mu_0) - d + \log \frac{\det \Sigma_1}{\det \Sigma_0}\right]$$

Intuition to calculate the ELBO

Thus, we can calculate our ELBO. The KL term in the ELBO is straightforward to calculate, as it merely requires calculating an analytic expression with our parameters $\mu_\phi, \mu_\theta, \Sigma_\phi, \Sigma_\theta$. The computational graph below shows how to calculate $\log p(\mathbf{x}|\mathbf{z})$.


$\mathbf{x}$
NN
NN
$q(\mathbf{z}|\mathbf{x})$:
$\mu_\phi(\mathbf{x})$
$\Sigma_\phi(\mathbf{x})$
Sample
$\mathbf{z}$
NN
NN
$p(\mathbf{x}|\mathbf{z})$:
$\mu_\theta(\mathbf{z})$
$\Sigma_\theta(\mathbf{z})$

The reparameterization trick

There's one last detail, which is that the term $\mathbb{E}_{\mathbf{z}} \log p(\mathbf{x}|\mathbf{z})$ involves a sampling step. This isn't an operation that we know how to backpropagate through. To get around this, we use something called the reparameterization trick. In the sampling operation, we want to sample:

$$\mathbf{z} \sim q(\mathbf{z}|\mathbf{x}^{(i)})$$

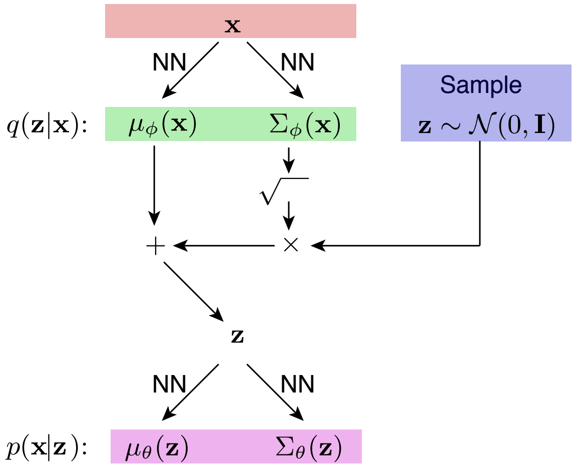
To do so, we sample $\varepsilon \sim \mathcal{N}(0, \mathbf{I})$ and calculate:

$$\mathbf{z} = \mu_\phi(\mathbf{x}^{(i)}) + \Sigma_\phi^{1/2}(\mathbf{x}^{(i)})\varepsilon$$

Then, $\mathbf{z}$ will be a sample from $q(\mathbf{z}|\mathbf{x}^{(i)})$ as its a linear transformation of ε with mean $\mu_\phi(\mathbf{x}^{(i)})$ and covariance $\Sigma_\phi(\mathbf{x}^{(i)})$. The sampling operation now occurs only for ε, which we don't need to backpropagate through.

Updated computational graph to calculate the ELBO

With the reparametrization trick, we can now backpropagate to calculate the gradient with respect to all parameters.


$\mathbf{x}$
NN
NN
Sample
$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$
$q(\mathbf{z}|\mathbf{x})$:
$\mu_\phi(\mathbf{x})$
$\Sigma_\phi(\mathbf{x})$
$\sqrt{}$
$+$
$\times$
$\mathbf{z}$
NN
NN
$p(\mathbf{x}|\mathbf{z})$:
$\mu_\theta(\mathbf{z})$
$\Sigma_\theta(\mathbf{z})$

Optimization of the ELBO

Now that we can calculate the ELBO, and know how to backpropagate through all the parameters, we can obtain two gradients:

$$\nabla_\theta \mathcal{L}_{\text{vae}} \qquad \& \qquad \nabla_\phi \mathcal{L}_{\text{vae}}$$

With these two gradients, we can now perform SGD to optimize the parameters of the encoder and decoder networks, ϕ and θ.

Caveats about the VAE

There are some caveats about the VAE that we need to be aware of.

- Inference used a lower bound rather than the likelihood of the data. Empirically, it has been observed that when the bound is maximized, the gap between the bound and the likelihood is not large. But there might still be a large gap between the maximum-likelihood and the bound.
- There currently remains no proof that VAEs are asymptotically consistent.
- Subjectively, VAEs generate images but other generative models do better.

We'll talk about one such model in the next lecture, the generative adversarial network (GAN).