Assignment 3: Generative Models

Jose Gallego

December 16, 2017

1 Introduction

Suppose we have a collection of N D-Dimensional points: $\{x_1, ..., x_N\}$. Each x_n might represent a vector of pixels in an image, or a bag of words in a document. In generative modeling, we imagine that these points are samples from a D-dimensional probability distribution. The distribution represents whatever real-world process was used to generate that data. Our objective is to learn the parameters of this distribution. This allows us to do things like:

1. Generate *new* data samples, given our estimate of the data distribution.
2. Estimate the probability that some new piece of data was generated by your model.
3. Fill in missing information in your data. e.g. If half of an image is cut off, can I guess what it should look like based on the other half?
4. Infer the "latent variables" which *caused* the data. For example, we can think of an image of a cat really being a nonlinear projection from some low-dimensional space, where instead of pixel-brightness, variables represent abstract things like the presence of a cat, and background, lighting, and camera position variables. Many tasks in machine learning become easier when working on with abstract variables than with raw pixel values directly.

In training, we want to maximize the probability of our data, given the model. Intuitively, we can think of our model as having a fixed amount of probability (in total 1) that it can distribute among all the possible data points (in the case of a D-dimensional binary data: $X \in \{0, 1\}^D$, there are 2^D possible data points). In training, our model learns to put high probability on the observed (training)

data points. Our real goal, however, is to learn a model that generalizes well, i.e. it does not just memorize the training points, but learns their underlying distribution, so that it would also assign high-probability to held-out test data that it did not have access to during training. We can then say that the model has learned the *distribution of the data*.

The aim of this assignment is give you insight into what makes this task hard, and some methods that have been used to overcome this hardness. Deep Generative models are a very active topic of research today.

For this assignment we will use the MNIST dataset - a collection of 60000 handwritten digits which serves as the most common benchmark for new machine learning algorithms.

Figure 1: Some digits from the MNIST Dataset

2 Goals of this assignment

By the end of this assignment, you should be able to:

- Build and train a simple generative model.
- Understand how the problem of *intractability* arises in generative modelling.
- Have a rough intuition as to how *Variational Inference* deals with the problem of intractability.
- Train a Variational Autoencoder: A generative model that uses a deep neural network.

This assignment doubles as a tutorial. To work on it, first open the assignment on Overleaf at https://www.overleaf.com/read/fbjbbtxxnkxg, then click "Clone Document" to start your own copy. Fill your answers in the "Answer" sections provided. Once finished, download the assignment as a PDF and submit it in a zip file on Blackboard along with all code required to reproduce your results. It's best to view this document online (as opposed to a printout) because there are several useful links throughout.

Many of the questions in this assignment test understanding. The answers will be graded on how well they indicate understanding, so be sure to be as specific as possible in your answers. Vague answers (statements may be true but are not specific enough to really answer the question) will lose marks. Short and specific answers are best.

3 Latent Variable Models

Latent Variable models are models where we assume that each data-point X is generated by some *Latent Variable* Z, which is transformed to X. For example, we may imagine that the pixels in an image (the visible data) are generated by some *Latent Variable* corresponding to the real-world objects in the image, along with the position of the camera, etc.

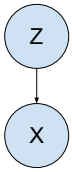


Figure 2: A general latent variable model. Each data point X is considered to be generated by a "latent" variable Z.

4 The simplest generative model we can imagine

Before we get into fancy deep generative networks, it is useful to implement the simplest model we can imagine. Here we will learn a model of the MNIST digits using a Naive Bayes model with a categorical latent variable.

In this assignment we will use the model in Figure 3 to represent the distribution of our MNIST digits. Here we create random variables Z and X_i. Each pixel (in the 28x28 digit images) will be represented by an element of X, so we have

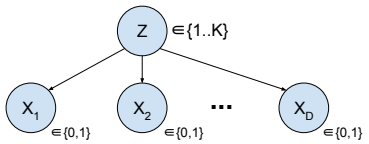


Figure 3: A Naive Bayes model with a Categorical Latent Variable Z (with K categories) and Bernoulli Visible variables X_i. Note that we have redrawn the graph to reveal our *Conditional Independence* (i.e. Naive) assumption: X_i is independent of all other elements of X ($X_{\setminus i}$) given Z.

$X_1 ... X_D$, where $D = 784$ is the dimensionality of the data. A white pixel corresponds to a 1 while a black pixel corresponds to a 0. Because we don't directly observe Z (we just *assume* that it exists), we say that Z is a *latent variable*. We will assume that each digit is generated from one of K categories (corresponding to the K values that Z can take).

4.1 Defining our Model

We can parameterize our model as follows:

$$p(Z) = \text{Cat}(Z; \text{softmax}(b)) \tag{1}$$
$$p(X_d|Z = k) = \text{Bern}(X_d; \text{sigm}(w_{k,d} + c_d)) \tag{2}$$

Where:
$X_d \in \{0, 1\}$ and $Z \in [1..K]$ are random variables.
$p(Z)$ is the probability distribution over latent categories.
$\text{Cat}(Z; softmax(b))$ is a Categorical distribution where the probability of the k'th category is $\text{softmax}(b)_k$
$p(X_d|Z = k)$ is the probability distribution over the value of the d'th pixel given that it is generated from the $k'th$ latent category.
$\text{Bern}(X; \mu)$ is a Bernoulli distribution over x with mean μ (ie $p(X = 1) = \mu$)
sigm is the sigmoid function: $\text{sigm}(x) := \frac{1}{1+e^{-x}} \in [0, 1]$
$w \in \mathbb{R}^{K \times D}$, $b \in \mathbb{R}^K$ and $c \in \mathbb{R}^D$ are trainable parameters

4.2 Sampling from our Model

To sample from this model, we first draw a sample for Z, then use this sample to generate an X. i.e. The sampling procedure is:

1. Sample k from $\text{Cat}(Z; \text{softmax}(b))$
2. Compute the mean data activation per dimension of x given k: $\mu_d := \text{sigm}(w_{k,d} + c_d)$
3. Sample each pixel value $x_d \in \{0, 1\}$ from $\text{Bern}(X_d; \mu_d)$

4.3 Training our Model

We want to train our model parameters (w, b) so that our model generates samples that are like our data. To do this, we will try to *maximize the probability of the data, given the model.*

Suppose we have a batch of binary data $x \in \{0, 1\}^{N \times D}$ (where N is the number of samples). $x_n \in \mathbb{R}^D$ is a particular data example, and $x_{n,d}$ is the $d'th$ pixel of example n. We can expand $p(x)$ so that it is written in terms of Equations 1 and 2.

$$p(X = x_n) = \sum_{k=1}^{K} p(X = x_n, Z = k) \qquad \text{Marginalization of Z} \tag{3}$$

$$= \sum_{k=1}^{K} p(Z = k) p(X = x_n | Z = k) \qquad \text{Decompose joint prob} \tag{4}$$

$$= \sum_{k=1}^{K} p(Z = k) \prod_{d=1}^{D} p(X_d = x_{n,d} | Z = k) \qquad \text{Naive Assumption} \tag{5}$$

Typically we do not maximize the probability directly, but the log-probability. Note that since the $\log p$ is *monotonic* in p, this is equivalent to maximizing the probability. But for both numerical stability and ease of optimization, we choose to optimize $\log p$ (see this discussion for why). For the model described above, we want to find optimal parameters:

$$w^*, b^* = \underset{w,b}{\text{argmax}} \log p(X = x) \tag{6}$$

In other words, our loss is the average *negative log likelihood* of the data.

$$\mathcal{L} := -\frac{1}{N} \sum_{n=1}^{N} \log p(x) \tag{7}$$

Problem 1 (6 Points)

Give a 1-3 sentence explanation for each of the 3 steps for expanding $p(X = x_n)$ from Equation 3 to 5. If the step involves an assumption, say what that assumption means. Show that you understand why those steps are taken.

Answer 1

We want to write the probability of one example in terms of those distributions we have assumed previously: a prior over the latent variable $p(Z = k)$, and the conditional distribution of a pixel given the latent variable $p(X_d = x_{n,d}|Z = k)$.

(3) We need to introduce the latent variable, that we can do by marginalizing over Z: $p(x) = \int p(x, y)dy$.

(4) By applying the definition of conditional probability, we can get a product of the prior (which we know) and a conditional distribution over the whole image vector.

(5) We exploit the assumption that the X_ds are conditionally independent given the latent variable (d-separation criterion) and thus can factorize the joint distribution.

Problem 2 (8 Points)

As previously noted, we optimize $\log p(x)$ instead of of $p(x)$. One of the reasons we work with log-probabilities instead of probabilities is numerical stability. Suppose (forget for a moment about the rest of the problem) that we have a vector of probabilities $[p_1...p_N] : p_n \in [0, 1]$, and we want to calculate $\log p_{total} = \log\left(\prod_{n=1}^{N} p_n\right)$. Show mathematically that there are two ways to calculate this, and show with a small (<10 lines) Python script that (1) these two ways produce the same result when N is small, but (2) the non-numerically stable method fails when N becomes large.

Answer 2

$$\log \prod_n p_n = \sum_n \log p_n$$

```python
import numpy as np
for n in [1, 10, 100, 500, 1000, 100000000]:
    p = np.random.rand(n)
    naive_logp = np.log(np.prod(p))
    stable_logp = np.sum(np.log(p))
    print(n, naive_logp, stable_logp)
```

N	Naive	Stable
1	-0.274245275897	-0.274245275897
10	-9.00537241442	-9.00537241442
100	-123.963356076	-123.963356076
500	-473.031232113	-473.031232113
1000	$-\infty$	-1020.38764215
100000000	$-\infty$	-100011926.534

Table 1: Numerical stability comparison

Problem 3 (5 Points)

Now that you understand the issues of numerical stability, modify Equation 5 to calculate $\log p(x)$ so that computations are numerically stable.

Answer 3

$$\begin{aligned}
\log p(X = x_n) &= \log \sum_{k=1}^{K} p(Z = k) \prod_{d=1}^{D} p(X_d = x_{n,d} | Z = k) \\
&= \log \sum_{k=1}^{K} \exp \left[\log \left[p(Z = k) \prod_{d=1}^{D} p(X_d = x_{n,d} | Z = k) \right] \right] \\
&= \log \sum_{k=1}^{K} \exp \left[\log p(Z = k) + \sum_{d=1}^{D} \log p(X_d = x_{n,d} | Z = k) \right]
\end{aligned}$$

It is also possible to use the log-sum-exp trick to further improve the stability of the computation.

Problem 4 (5 Points)

Using your last result, write equation for $\log p(x)$ in terms of the parameters of your model: w and b.
Hint: Your can write your result in terms of a Bernoulli Variable X parameterized by $\theta \in [0, 1]$: $\text{Bern}(x; \theta) := x\theta + (1-x)(1-\theta)$ for $x \in \{0, 1\}$.

Answer 4

$$\log p(Z = k) = \log \frac{e^{b_k}}{\sum_{k'} e^{b_{k'}}} = b_k - \log \sum_{k'} e^{b_{k'}}$$

Let $\alpha_{k,d} = w_{k,d} + c_d$.

$$\begin{aligned} \log p(X_d = x_{n,d} | Z = k) &= \log Ber\left(x_{n,d}, \sigma(\alpha_{k,d})\right) \\ &= \log \left(\sigma(\alpha_{k,d})^{x_{n,d}} (1 - \sigma(\alpha_{k,d})^{1-x_{n,d}}\right) \\ &= x_{n,d} \log \sigma(\alpha_{k,d}) + (1 - x_{n,d}) \log \sigma(-\alpha_{k,d}) \end{aligned}$$

Putting it all together,

$$\begin{aligned} \log p(X = x_n) &= \log \sum_{k=1}^{K} \exp \Bigg(b_k - \log \sum_{k'} e^{b_{k'}} \\ &\qquad + \sum_{d=1}^{D} x_{n,d} \log \sigma(\alpha_{k,d}) + (1 - x_{n,d}) \log \sigma(-\alpha_{k,d}) \Bigg) \end{aligned}$$

4.4 Coding our Model

Now, we will train this model on MNIST.

Problem 5 (20 Points)

Start by using the template in a3_simple_template.py. If the code is implemented correctly, you should be able to train the network with the default settings in that file.
Train the model and submit the code for training. If your answers to problems 7, 8, 9 indicate that you have trained the model successfully you get full points here.

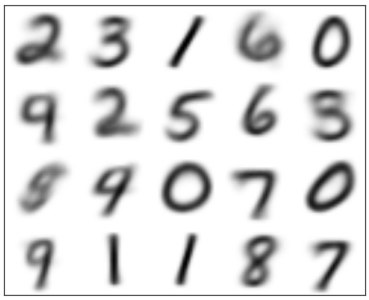
Answer 5

See attached code. We have chosen white backgrounds for the dataset to improve readability.

Problem 6 (5 Points)

After training, plot K images, where image k shows an image of the expected pixel values given that the latent variable $Z = k$. ie. The $i'th$ pixel in the (flattened) image K will be $p(X_i = 1|Z = k)$

Answer 6



Problem 7 (5 Points)

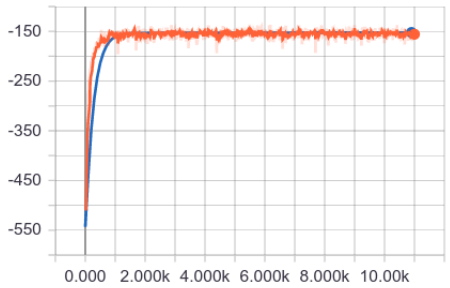
Show 16 images samples from your trained model.

Answer 7

Problem 8 (5 Points)

Plot learning curves of your training and test log-probability. For speed, you can just use the first 1000 samples of the training/test sets to compute this.

Answer 8



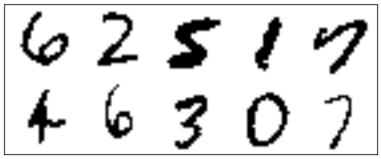
The plot shows the average log-likelihood for training (orange) and testing (blue) examples as training evolves.

Problem 9 (10 Points)

Create "Frankenstein" digits by splicing together the left and right half of randomly paired MNIST digits of different classes, and compute the log-probability of these digits under your model (which you already trained in the Problems 7 and 8). Plot 10 of these Frankenstein digits alongside 10 natural (unmodified) digits, and show the log-probabilities for each. Describe how you might use your model to reliably verify whether a batch of samples are Frankenstein digits or not.

Answer 9

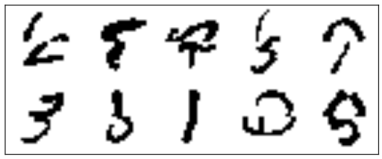
Original Digits



[-164.14 -146.53 -190.32 -110.96 -247.65
-147.36 -151.64 -148.78 -163.02 -119.99]

$\mu = -159.045 \qquad \sigma = 36.278$

Frankenstein Digits



[-204.23 -179.98 -231.98 -189.70 -177.80
-141.14 -145.33 -86.12 -248.64 -240.10]

$\mu = -184.507 \qquad \sigma = 47.993$

A possible way to detect whether a batch is made of Frankenstein digits is to do a statistical t-test to compare the mean log-likelihood between the given batch and a control batch which we are certain has no Frankenstein digits. We ran such a test with the samples shown above. The p-value was 0.22, from which we can not conclude that the two populations (original and Frankenstein digits) have a statistically significantly different mean. Note however that the log-likelihood for the above samples is better in the original case. The fact that the test is inconclusive might be due to the small size.

5 Understanding Intractability

We will now modify our previous model to have a *distributed latent representation*. It now becomes a 1-layer sigmoid belief net. Although it's not required to watch, a short and possibly useful video lecture by Geoff Hinton on the topic of sigmoid belief nets and how they relate to intractability is available here.

Our previous modelling assumption - that every sample in the data belongs to just 1 of K categories - was quite restrictive. A more interesting model might have a distributed latent representation. In other words, our data is caused

not by one latent variable, but by the interaction of all the elements of an M-dimensional latent variable. That is: $Z \in \{0,1\}^M$.

Let us now define a model with a *distributed latent representation:*. Note that this time we will now write out the full conditional probability for X: $p(X|Z)$, instead of the probability for a single element of X: $p(X_d|Z)$

$$p(Z) = \prod_m \text{Bern}(Z; \text{sigm}(b_m)) \qquad \text{Assume marginal independence} \tag{8}$$

$$p(X|Z=z) = \prod_d \text{Bern}(X_d; \text{sigm}(z \cdot w_{\cdot,d} + c_d)) \qquad \text{We keep the Naive assumption} \tag{9}$$

Where
$X \in \{0,1\}^D, Z \in \{0,1\}^M$ are random variables.
$w \in \mathbb{R}^{M,D}$, $b \in \mathbb{R}^M$, $c \in \mathbb{R}^D$ are the model parameters.
$w_{\cdot,d}$ is the d'th column of weight matrix w

This model would be called a 1-layer Sigmoid Belief network. We can modify Equation 5 to show how we would calculate the probability of data under this model:

$$p(X = x_n) = \sum_{z \in Z} p(Z = z) \prod_{d=1}^{D} p(X_d = x_{n,d} | Z = z) \tag{10}$$

Where
$\sum_{z \in Z}$ indicates a sum over all possible values of Z.

Problem 10 (9 Points)

(A) How many possible values of Z are there (ie, how many terms are in the sum $\sum_{z \in Z}$) ?
(B) Using big-O notation, write down the time-complexity of computation for each step of training in terms of N, M and D.
(C) Compare it to the time-complexity of the previous model (in terms of N, K and D).

Answer 10

(A) There are 2^M possible configurations of the latent variable and thus 2^M summands.

Since the questions concern the time and not the memory complexity of the algorithms, the proposed pseudocode might not be optimal in terms of memory, i.e. could imply caching large tensors.

(B) *Sigmoid Belief Network*

for $z \in Z$ **do** ▷ $\mathcal{O}(2^M)$
  Compute $p(z)$ ▷ $\mathcal{O}(M)$
end for
for $X_n \in \{X_n\}_{n=1}^N$ **do** ▷ $\mathcal{O}(N)$
  for $z \in Z$ **do** ▷ $\mathcal{O}(2^M)$
    Compute $p(X_n|z)$ ▷ $\mathcal{O}(D)$
    Update accumulator for Eq (10) ▷ $\mathcal{O}(1)$
  end for
end for
Compute $\nabla_w \mathcal{L}$ ▷ $\mathcal{O}(NMD)$
Compute $\nabla_b \mathcal{L}$ ▷ $\mathcal{O}(NM)$
Compute $\nabla_c \mathcal{L}$ ▷ $\mathcal{O}(ND)$
Apply update $\nabla_w \mathcal{L}$ ▷ $\mathcal{O}(MD)$
Apply update $\nabla_b \mathcal{L}$ ▷ $\mathcal{O}(M)$
Apply update $\nabla_c \mathcal{L}$ ▷ $\mathcal{O}(D)$

Train Step Complexity: $\mathcal{O}((M + ND)2^M)$

(C) *Naive Bayes*

for $X_n \in \{X_n\}_{n=1}^N$ **do** ▷ $\mathcal{O}(N)$
  for $z \in Z$ **do** ▷ $\mathcal{O}(K)$
    Retrieve $p(z)$ ▷ $\mathcal{O}(1)$
    Compute $p(X_n|z)$ ▷ $\mathcal{O}(D)$
    Update accumulator for Eq (5) ▷ $\mathcal{O}(1)$
  end for
end for
Compute $\nabla_w \mathcal{L}$ ▷ $\mathcal{O}(NKD)$
Compute $\nabla_b \mathcal{L}$ ▷ $\mathcal{O}(NK)$
Compute $\nabla_c \mathcal{L}$ ▷ $\mathcal{O}(ND)$
Apply update $\nabla_w \mathcal{L}$ ▷ $\mathcal{O}(KD)$
Apply update $\nabla_b \mathcal{L}$ ▷ $\mathcal{O}(K)$
Apply update $\nabla_c \mathcal{L}$ ▷ $\mathcal{O}(D)$

Train Step Complexity: $\mathcal{O}(NKD)$

Both models have linear complexity in the batch size and the dimension of the observed variable. However, Naive Bayes is cheaper since it is linear in the dimension of the latent variables (imagine each class a as K-dim one hot vector), while SBNs are (worse than) exponential in M.

6 A Variational Autoencoder

By now you should have a grasp of the concept of intractability, and why it makes it hard to learn even a shallow generative model: the number of possible values of latent variable Z grows exponentially with the dimensionality of the latent space. For continuous latent variables, it's even worse: There are infinite possible values of Z, so there is no way to compute an exact data likelihood. One approach that has been taken to tackle the problem of intractability is Variational Inference. The approach taken in Variational Inference to give up on directly optimizing the log-probability of the data, and instead optimize a *lower bound* to the log-probability.

This section will introduce you to a type of generative model that was discovered right here in Science Park 904 by Kingma & Welling, 2013: The Variational Autoencoder (VAE). VAEs use a pair of neural networks to generate data from latent variables, and to generate latent variables from data. An overall schematic of a variational autoencoder can be found in Figure 4.

Two excellent tutorials on VAEs are this one by Brian Keng and this one by Carl Doersch. We will walk through VAEs in this assignment, but if you become confused we recommend looking through those sources.

6.1 The Generative part (decoder)

We will begin by defining a generative model that uses neural networks. This will later be referred to as the *decoding* part of a variational autoencoder.

$$p(Z) = \mathcal{N}(0,1)^{D_Z} \tag{11}$$

$$p_\theta(X|Z=z) = \prod_d^{D_X} \text{Bern}(X_d; \mu_{X;\theta}(z)_d) \tag{12}$$

Where:
D_Z is the dimensionality of the latent space (e.g. 5)
D_X is the dimensionality of the data (e.g. 784)
$N(0,1)^{D_Z}$ is a standard Normal distribution in D_Z dimensions
$\text{Bern}(X, \mu)$ is a bernoulli distribution parametrized by $\mu \in [0,1]^{D_X}$
$\mu_{X;\theta} : D_Z \times |\theta| \rightarrow D_X$ is a neural network with parameters θ that takes a sample z and outputs the the mean of the Bernoulli distributions for each pixel in X
θ represents all the parameters for the neural network $\mu_{X;\theta}$.

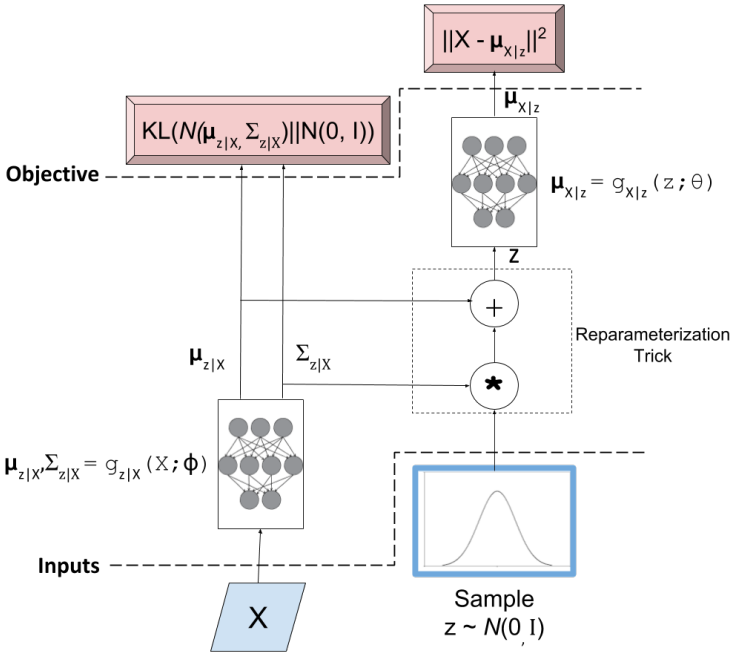


Figure 4: A schematic of a VAE (taken from Brian Keng's website). Note that in their model, X is considered a Gaussian-distributed Random Variable, whereas in ours we consider it to be Bernoulli-Distributed (which is more appropriate the the MNIST data).

Problem 11 (5 Points)

Describe how you would *sample* from such a model, in the same way that we described sampling from the naive model in Section 4.2

Answer 11

1. Sample z from $N(0,1)^{Dz}$
2. Do a forward pass of z through the network to obtain $\mu_{X;\theta}(z)_d$
3. Sample each pixel value $x_d \in \{0,1\}$ from $\text{Bern}(x_d; \mu_{X;\theta}(Z)_d)$

Problem 12 (5 Points)

This model makes a very simplistic assumption about $p(Z)$. i.e. It assumes our latent variables follow a standard-normal distribution. Note that there is no trainable parameter in $p(Z)$. Describe why, due to the nature of Equation 12, this is not such a restrictive assumption in practice. **Hint** See Figure 2 and the accompanying explanation in Carl Doersch's tutorial.

Answer 12

Even though we use a simple distribution for $p(Z)$ (from which we can easily sample), the fact that we use a expressive enough neural network to map the latent representation to the parameters of a distribution from which we sample the observed variables, makes the initial assumption much less restrictive.

Now, lets write out an expression for the log-probability of this the data under this model (analogous to Equation 5 but with a continuous latent variable this time).

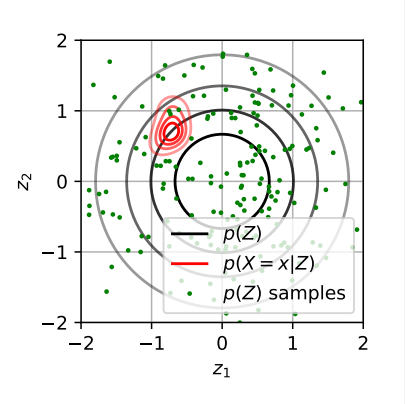
$$\log p(X) = \log \int_z p(X,z)dz \tag{13}$$

$$= \log \int_z p(z)p(X|z)dz \tag{14}$$

$$= \log \mathbb{E}_{z\sim p(Z)}[p(X|z)] \tag{15}$$

Problem 13 (10 Points)

(A) Evaluating $\log p(x)$ involves an intractable integral. But Equation 15 hints at how we could approximate this probability. Write down an expression for approximating $\log p(x)$ by sampling. **Hint**: You can use sampling to approximate an expectation. e.g. The $\mathbb{E}_{x\sim p(x)}[f(x)] := \int p(x)f(x)dx \approx \frac{1}{M}\sum_m^M f(x_m)$ where x_m are sampled from $p(x)$.
(B) This approach is *not* used for training this type of model, because it is inefficient. In a few sentences, describe why it is inefficient. How does this efficiency scale with the dimensionality of Z?
Hint: You may use the following figure in your explanation:



Answer 13

(A) $\log p(X) = \log \mathbb{E}_{z\sim p(Z)}[p(X|z)] \approx \log\left(\frac{1}{M}\sum_m^M p(X|z_m)\right)$, where z_m are sampled from $p(Z)$.

(B) Fix the observed variable X. As shown in the picture, for most z, $p(X|z)$ will be nearly zero, and hence contribute almost nothing to our estimate of $p(X)$. We would like to sample values of z that are likely to have produced X, and estimate $p(X)$ just from those. However, the size of the regions with high density reduces exponentially as the dimension of the latent space increases.

6.2 The encoder: q(Z|X)

The intuition we should have gained by now is that we only want to sample latent variables Z that are likely to have caused our data. i.e. we want to sample

z's for which $p(X = x|z)$ is not close to zero.

One approach to solving this is to instead sample from a *variational distribution* $q(Z|X)$, which we use to approximate our (intractable) posterior $p(Z|X)$. One way to see if two distributions are close is to use the KL-divergence:

$$\mathcal{D}(q\|p) := \mathbb{E}_{x\sim q(X)}\left[\log\left(\frac{q(x)}{p(x)}\right)\right] = \int q(x)\log\left(\frac{q(x)}{p(x)}\right)dx \tag{16}$$

Where: q, p are probability distributions in the space of some random variable X.

Problem 14 (5 Points)

Assume that q and p in Equation 16, are univariate gaussians: $q = \mathcal{N}(\mu_q, \sigma_q^2)$ and $p = \mathcal{N}(0, 1)$.
(A) Give two examples of (μ_q, σ_q^2): one of which results in a very small, and one of which has a very large, KL divergence: $\mathcal{D}(\mathcal{N}(\mu_q, \sigma_q^2)\|\mathcal{N}(0, 1))$
(B) Find the formula for $\mathcal{D}(\mathcal{N}(\mu_q, \sigma_q^2)\|\mathcal{N}(0, 1))$. (You do not need to derive it, you can just search for it, but feel free to derive it if that brings you joy). You will need to use this answer later.
Hint: This will be helpful.

Answer 14

(B) Consider the more general case

$$\mathcal{D}(\mathcal{N}(\mu_1, \sigma_1^2)\|N(\mu_2, \sigma_2^2)) = \log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}.$$

Clearly, $\mathcal{D}(\mathcal{N}(\mu_q, \sigma_q^2)\|\mathcal{N}(0, 1)) = \frac{\sigma_q^2+\mu_q^2-1}{2} - \log\sigma_q$.

(A) It is easy to see that the divergence vanishes when the two distributions coincide, so for a small KL divergence take $(\mu_q, \sigma_q^2) = (0,1)$. On the other hand, the divergence is quadratic in the difference of the means, so for a large KL divergence take $(\mu_q, \sigma_q^2) = (\mu,1)$, for some μ with large magnitude.

Now, if we write out the expression for the KL divergence between our proposal $q(Z|X)$ and our posterior $p(X|Z)$, we can derive an expression for the probability of our data under our model:

$$\mathcal{D}(q(z|X)\|p(z|X)) \triangleq \mathbb{E}_{z\sim q(Z|X)}\left[\log\left(\frac{q(z|X)}{p(z|X)}\right)\right] \tag{17}$$

$$= \mathbb{E}_{z\sim q(Z|X)}\left[\log\left(\frac{q(z|X)p(X)}{p(X|z)p(z)}\right)\right] \tag{18}$$

$$= \mathbb{E}_{z\sim q(Z|X)}\left[\log\left(\frac{q(z|X)}{p(z)}\right) - \log p(X|Z) + \log p(X)\right] \tag{19}$$

$$= \mathcal{D}(q(z|X)\|p(z)) - \mathbb{E}_{z\sim q(Z|X)}\left[\log p(X|z)\right] + \log p(X) \tag{20}$$

$$\log p(X) - \mathcal{D}(q(z|X)\|p(z|X)) = \mathbb{E}_{z\sim q(Z|X)}\left[\log p(X|z)\right] - \mathcal{D}(q(z|X)\|p(z)) \tag{21}$$

We've arranged the equation above so that directly-computable quantities are on the right-hand side. The right side of the equation is referred to as the *lower bound* on the log-probability of the data. This is what will optimize. So, just as we previously defined our loss to be the mean negative log likelihood over samples, we now define our loss as the mean negative lower bound:

$$\mathcal{L} = -\frac{1}{N}\sum_n \left(\mathbb{E}_{z\sim q(Z|X)}\left[\log p(X|z)\right] - \mathcal{D}(q(z|X)\|p(z))\right) \tag{22}$$

Problem 15 (5 Points)

Why is the right-hand-side of Equation 21 called the *lower bound* on the log-probability?

Answer 15

If $x - y = z$ and y is non-negative, then $x \geq z$. The fact that the KL divergence is non-negative applied to Equation (21) gives the desired conclusion.

Problem 16 (5 Points)

Looking at Equation 21, why must we optimize the lower-bound, instead of optimizing the log-probability directly?

Answer 16

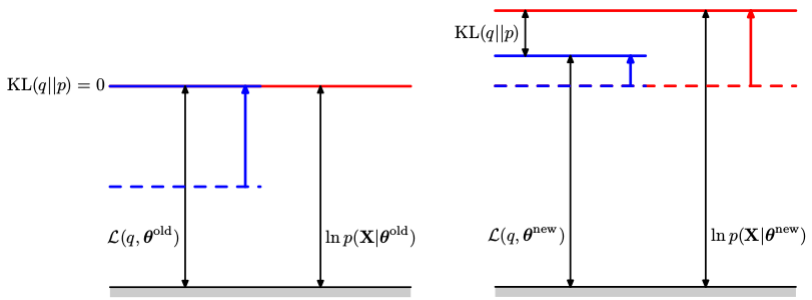
Because to optimize the log-probability directly, we would have to compute $\mathcal{D}(q(z|X)\|p(z|X))$, which involves the intractable posterior $p(z|X)$.

Problem 17 (6 Points)

Now, looking at the two terms on left-hand side of 21: Two things can happen when the lower bound is pushed up. Can you describe what these two things are?

Answer 17

When we optimize wrt the variational parameters ϕ (or q itself), the divergence $\mathcal{D}(q(z|X)\|p(z|X))$ is reduced (left image). When we optimize wrt the parameters of the generative model θ, this causes the log-likelihood $\log p(X|\theta)$ to increase by at least as much as the lower bound does (right image).



Lower bound optimization wrt ϕ (left) and wrt θ (right). Note that if we restrict q to be a certain model with parameters ϕ it might not be possible to achieve equality with the posterior in the left plot. Image taken from (Bishop, 2006).

6.3 Specifying the encoder $q_\phi(Z|X)$

In Variation autoencoders, we define $q_\phi(Z|X = x_n)$ to be a normally distributed random variable, whose distribution depends on X. Like the decoder, q will be defined by a neural network.

$$q_\phi(Z|X = x_n) = \mathcal{N}(Z; \mu_{Z;\phi}(x_n), \Sigma_{Z;\phi}(x_n)) \tag{23}$$

Where:
$x_n \in \mathbb{R}^{Dx}$ is the a particular data sample
$\mu_{Z;\phi}(X) \in \mathbb{R}^{Dz}$ is the mean of the gaussian q(Z) for a given X
$\Sigma_{Z;\phi}(X) \in \mathbb{R}^{Dz}_{>0}$ is a Diagonal covariance matrix.
$\mu_{Z;\phi}, \Sigma_{Z;\phi} = g_{Z|X;\phi}(X)$ ie. the parameters of our approximate posterior distribution are generated by a single neural network with two output layers (as in Figure 4). To ensure that Σ's are positive, they are computed with an exponential nonlinearity: $f_\Sigma(x) = exp(x) \in \mathbb{R}^{Dz}$ at the last layer.

Problem 18 (6 Points)

The loss in Equation 22:

$$\mathcal{L} = -\frac{1}{N}\sum_n \left(\mathbb{E}_{z\sim q(Z|X)}\left[\log p(X|z)\right] - \mathcal{D}(q(z|X)\|p(z))\right)$$

can be rewritten in terms of per-sample losses:

$$\mathcal{L} = -\frac{1}{N}\sum_n^N (\mathcal{L}_{recon,n} - \mathcal{L}_{reg,n})$$

Where:
$\mathcal{L}_{recon,n} := \mathbb{E}_{z\sim q(Z|X=x_n)}\left[\log p(X = x_n|z)\right]$ can be seen as a reconstruction loss
$\mathcal{L}_{reg,n} := \mathcal{D}(q(z|X = x_n)\|p(z))$ can be seen as a regularization term.

Explain why the terms "reconstruction" and "regularization" are appropriate for these two losses.
Hint: Suppose (as is common practice in VAEs) we use just 1 sample to approximate the expectation $E_{z\sim q(Z|X=x_n)}[\cdot]$

Answer 18

$\mathcal{L}_{recon,n}$ represents how likely it is that we obtain x_n if we decode the latent representation z. In a sense, how good was z as a code for x_n.
$\mathcal{L}_{reg,n}$ forces the variational distribution $q(z|X = x_n)$ to be close to the prior $p(z)$, and thus can be seen as a regularization term.

Problem 19 (10 Points)

Now, putting together your model definition (Equations 11, 12), your variational distribution definition (Equation 23), write down an expressions for $\mathcal{L}_{recon,n}$, $\mathcal{L}_{reg,n}$ (as defined in the Problem 18) that you can actually optimize.
Hint: For $\mathcal{L}_{recon,n}$ you'll need to use sampling to approximate the expectation.
Hint: For $\mathcal{L}_{reg,n}$, You'll need to use your answer from problem 14

Answer 19

$$\mathcal{L}_{recon,n} = \mathbb{E}_{z\sim q(Z|X=x_n)}\left[\log p(X=x_n|z)\right] \approx \frac{1}{L}\sum_{l=1}^{L} \log p(X=x_n|z_l),$$

where $z_l \sim q(z|X=x_n) = \mathcal{N}(z; \mu_{Z;\phi}(x_n), \Sigma_{Z;\phi}(x_n))$.

$$\mathcal{L}_{reg,n} := \mathcal{D}(q(z|X=x_n)\|p(z)) = \frac{1}{2}\sum_{j=1}^{D_Z}\left[\mu_j^2 + \sigma_j^2 - \log\sigma_j^2 - 1\right]$$

where $\mu_j = \mu_{Z;\phi}(x_n)_j$ and $\sigma_j^2 = \Sigma_{Z;\phi}(x_n)_{jj}$ (cf. Appendix B - Kingma and Welling, 2013).

6.4 The Reparametrization Trick

Note that we're not quite done yet. We use sampling to approximate the term $E_{q_\phi(Z|X)}\left[\log p(X|Z)\right]$, which is in our loss. Yet we need to pass the derivative through these samples if we want to compute the gradient of the encoder parameters: $\frac{\partial\mathcal{L}}{\partial\phi}$.

Problem 20 (9 Points)

Read and understand Figure 4 from the tutorial by Carl Doersch.
In a few sentences each, explain why:
(A) we need $\frac{\partial\mathcal{L}}{\partial\phi}$
(B) the act of sampling prevents us from computing $\frac{\partial\mathcal{L}}{\partial\phi}$
(C) What the *reparametrization trick* is, and how it solves this problem.

Answer 20

(A) We are using a parametric model $q_\phi(z|x)$ to approximate the intractable posterior $p_\theta(z|x)$. However, we do not a priori know the parameters ϕ that would make q a *good* approximation. Thus, since we take q to be a NN, we would like to optimize its parameters using gradient descent, and this gradient is precisely $\frac{\partial \mathcal{L}}{\partial \phi}$.

(B) Our computation of the loss implies sampling from $\mathcal{N}(z; \mu_{Z;\phi}(x_n), \Sigma_{Z;\phi}(x_n))$ which is a non-differentiable operation on ϕ, and thus we can not compute $\frac{\partial \mathcal{L}}{\partial \phi}$ directly.

(C) As mentioned above, if we want to do gradient descent on the loss $\mathcal{L}$ wrt ϕ, we need that any operation in our computational graph that involves ϕ is almost everywhere differentiable wrt ϕ. The reparametrization trick effectively modifies the sampling module such that all we can fulfill the mentioned requirement. This is done by moving the sampling to a separate stochastic component ϵ (through which we do not need to back-propagate) and transform ϵ in a way that is differentiable wrt ϕ.

7 Building a VAE

Problem 21 (30 Points)

Build a Variational Autoencoder in tensorflow, and train it on the MNIST data. Start with the template in: a3_vae_template.py. Submit your code along with this assignment.
For simplicity, you may assume that the number of samples used to approximate your reconstruction loss is 1 (this is common practice anyway).
Hint: You may find it covenient to build your encoder and decoder networks with Keras (see tf.keras.models.Sequential, tf.keras.layers.Dense, tf.keras.layers.Activation)
You will get full points on this Problem if your answers to the following problems indicate that you successfully trained the VAE.

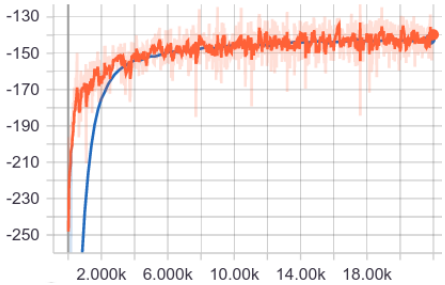
Answer 21

See attached code.

Problem 22 (10 Points)

Plot your the estimated lower-bounds of your training and test set as training progresses.
Hint: With the default parameters, the mean over samples of the lower bound on the training set should start around -544 and rise above -174 by epoch 1.

Answer 22

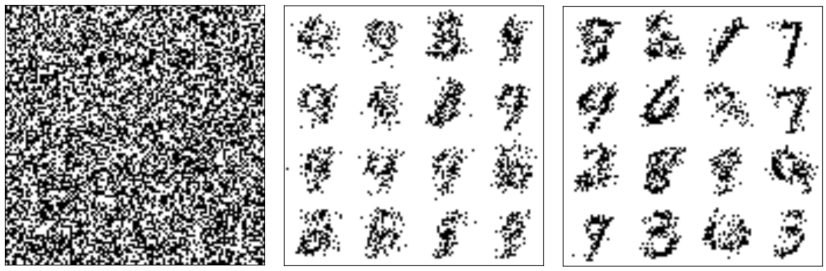


The plot shows the average lower bound for training (orange) and testing (blue) examples as training evolves.

Problem 23 (10 Points)

Plot samples from your model at 3 points throughout training (first, before any training, next: part way through, finally: after you finish training). You should observe an improvement in the quality of samples.

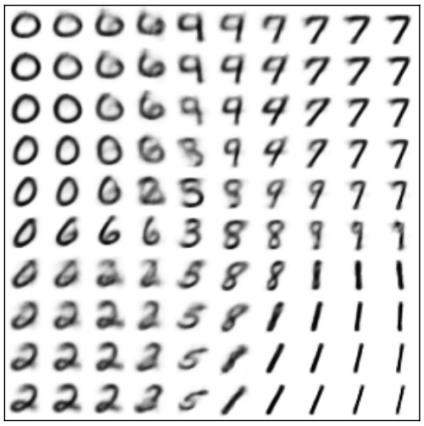
Answer 23



Problem 24 (10 Points)

After training has completed, plot a slice of the learned **manifold** - as is done in Figure 4b of Auto-Encoding Variational Bayes. This is done by taking a 2-D grid of points in Z-space (you can just use the first 2-dimensions of Z), and plotting the mean-probability of X: $\mu_{X|Z=z}$, for each of these points.
Hint: Use 'np.meshgrid'

Answer 24



Total Points: 204