

# Assignment 3: Generative Models

## Dana Kianfar 11391014

January 6, 2018

### 1 Introduction

Suppose we have a collection of  $N$   $D$ -Dimensional points:  $\{x_1, \dots, x_N\}$ . 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.

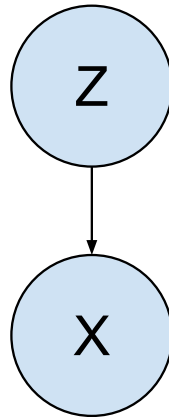
This assignment doubles as a tutorial. To work on it, first open the assignment on Overleaf at <https://www.overleaf.com/read/fbjbtxnxkxg>, then click “Clone Document” to start your own copy. Fill your answers in the “Answer” sections provided. Once finished,

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.

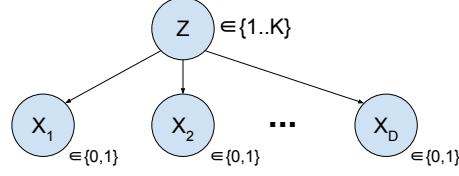


**Figure 2:** A general latent variable model. Each data point  $X$  is considered to be generated by a “latent” variable  $Z$ .

### 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  $X_1 \dots X_D$ , where  $D = 784$  is the



**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  $Z$ .

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:

$$p(Z) = \text{Cat}(Z; \text{softmax}(b)) \quad (1)$$

$$p(X_d|Z = k) = \text{Bern}(X_d; \text{sigm}(w_{k,d} + c_d)) \quad (2)$$

Where:

- $X_d \in \{0, 1\}$  and  $Z \in [1..K]$  are random variables.
- $p(Z)$  is the probability distribution over latent categories.
- $\text{Cat}(Z; \text{softmax}(b))$  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  $\mu$  (ie  $p(X = 1) = \mu$ )
- $\text{sigm}(\cdot)$  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  $k$ :  $\mu_d := \text{sgm}(w_{k,d} + c_d)$
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.

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

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

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

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:

$$w^*, b^* = \arg \max_{w, b} \log p(X = x) \quad (6)$$

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

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

### 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

**Equation 3** Using the marginalization property, we rewrite  $p(X = x_n)$  to include the latent variable  $Z$ , allowing us to account for the generative process ( $Z$  generates  $X$ ) using the joint probability term.

**Equation 4** We decompose the joint probability which further specifies the generative process with more granularity. We first sample an outcome  $k$  from the distribution over the random variable  $Z$ , and condition on this outcome to generate (conditionally sample from)  $X$ . This is the generative assumption, namely that the latent variables constitute the observed variables.

**Equation 5** This equation highlights the Naive Bayes assumption that is illustrated in Figure 3, namely that the dimensions (i.e. pixels) of any observation  $x_n$  are conditionally independent given knowledge of  $Z$ . This is a very strong assumption, especially in our example of learning generative models of images. Any two neighboring pixels are most likely not independent given the latent class, i.e. presence of a cat, or in general.

### Problem 2 (8 Points)

As previously noted, we optimize  $\log p(x)$  instead 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  $[p_1 \dots p_N] : p_n \in [0, 1]$ , 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

The two ways to compute the desired quantity is as follows.

**Numerically unstable method** This method is subject to numerical underflow, as the product of  $(p_i)_{i=1}^N$  decreases exponentially with  $N$ .

$$\log p_{total} = \log \prod_i p_i$$

**Numerically stable method** This method is more robust towards numerical underflow, assuming that each individual probability value is not too small.

$$\log p_{total} = \sum_i \log p_i$$

**Example with  $N=10, 100, 500, 1000$**  We present an example of the two implementations below.

```
In [1]: import numpy as np

# get random (unnormalized) probability values
In [2]: def get_p(N=10):
...:     return np.random.rand(N)

# Naive
In [6]: def naive_logp(p):
...:     return np.log(p.prod())

# Robust
In [8]: def logp(p):
...:     return np.log(p).sum()

In [14]: p_vecs = [get_p(N) for N in [10,100,500,1000]]

# Naive Implementation
In [27]: [naive_logp(p) for p in p_vecs]
RuntimeWarning: divide by zero encountered in log
Out[27]: [-13.0489, -95.6393, -504.0457, -inf]

# Robust implementation
In [28]: [logp(p) for p in p_vecs]
Out[28]: [-13.0489, -95.6393, -504.0457, -1000.9396]
```

## 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**

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

**Problem 4 (5 Points)**

Using your last result, write equation for  $\log p(x)$  in terms of the parameters of your model:  $w$ ,  $b$  and  $c$ .

**Hint:** You 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**

$$\begin{aligned} \log p(Z = k) &= \log \left[ \prod_{i=1}^K p_i^{\delta_i^k} \right] = \sum_i^K \delta_i^k \log p_i = \log \text{softmax}(b)_k \\ &= b_k - \log \sum_{i=1}^K \exp b_i \end{aligned}$$

Defining  $\theta = \text{sigm}(w_{kd} + c_d)$ , we have

$$\begin{aligned} \log p(x_d = 1 | \theta) &= \log \left[ \theta^{x_d} \cdot (1 - \theta)^{1-x_d} \right] = x_d \log \theta + (1 - x_d) \log(1 - \theta) \\ \log p(X = x_n) &= \log \sum_{k=1}^K \exp \left[ \log p(Z = k) + \sum_{d=1}^D \log p(X_d = x_{n,d} | Z = k) \right] \\ &= \log \sum_{k=1}^K \exp \left[ b_k - \log \sum_{i=1}^K \exp b_i \right. \\ &\quad \left. + \sum_{d=1}^D x_{nd} \log \text{sigm}(w_{kd} + c_d) + (1 - x_{nd}) \log(1 - \text{sigm}(w_{kd} + c_d)) \right] \end{aligned}$$

**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.

#### Hints:

- For further numerical stability, use functions [tf.reduce\\_logsumexp](#), and [tf.log\\_sigmoid](#) where possible.
- It is possible (and cleaner) to write the code without any loops (so you can avoid the scan function). For this, you can use [array broadcasting](#): ie first build a  $N \times K \times D$  array and then sum the elements appropriately to get a log probability. [tf.gather](#) may be useful.
- Before training, the mean (over samples) of the log probability of the training data under the model should start at around -540. With the default hyperparameters, it should go above -180 after 1 epoch of training.
- Using [tf.distributions.Bernoulli](#) can save you some manual coding.

### Answer 5

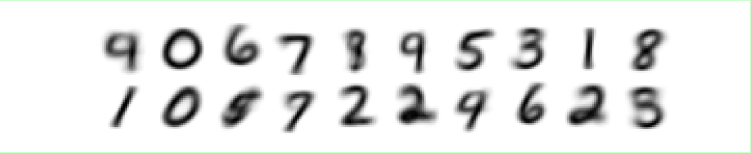
Please refer to code.

### 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  $p(X_i = 1|Z = k)$

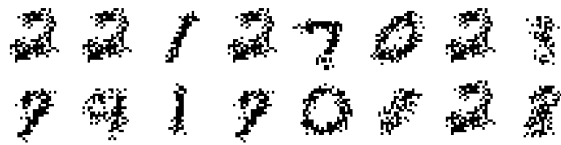
### Answer 6

$$P(x = 1|Z = k) \text{ for } k \in \{1, 2, \dots, 20\}$$



### Problem 7 (5 Points)

Show 16 images samples from your trained model.

**Answer 7**

2 2 1 2 7 0 2 1  
7 9 1 7 0 7 2 1

**Problem 8 (5 Points)**

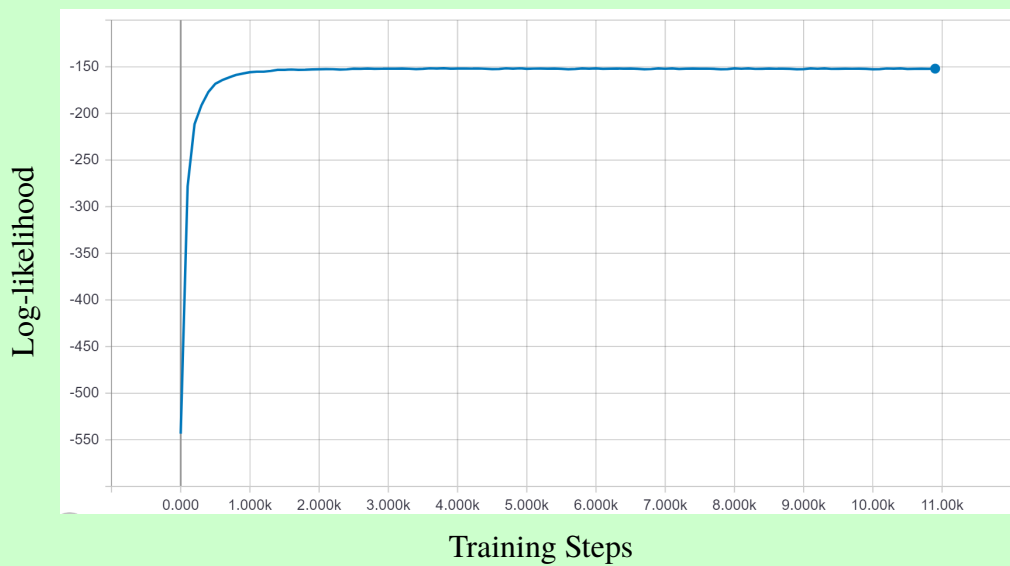
Plot learning curves of your training and test log-probability. For speed, you can just use the first 1000 samples of the training/test sets to compute this.

## Answer 8

Training log-likelihood across training steps



Test log-likelihood across training steps (for a fixed subset of 1000 test points)

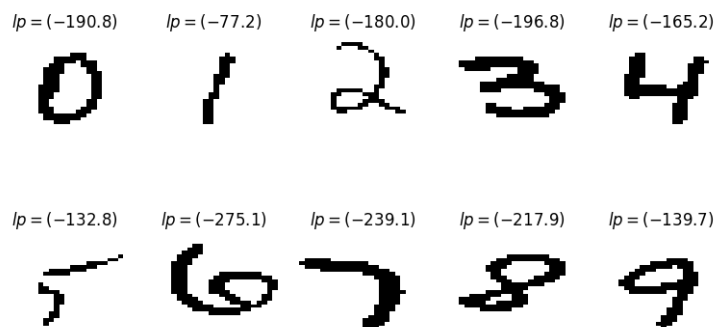


### Problem 9 (10 Points)

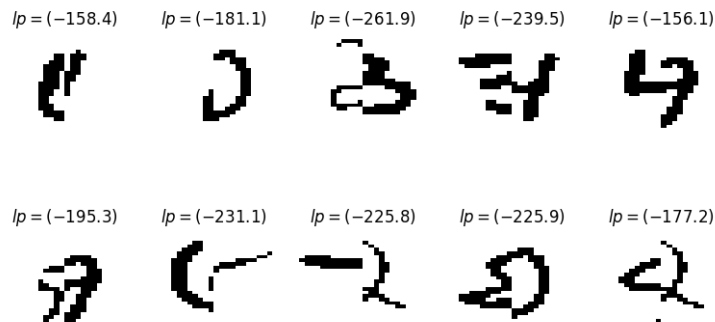
Create “Frankenstein” digits by splicing together the left and right half of randomly paired MNIST digits of different classes, and compute the log-probability of these digits under your model (which you already trained in the Problems 7 and 8). Plot 10 of these Frankenstein digits alongside 10 natural (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

Normal images with their log-probabilities



Frankenstein images with their log-probabilities



One way to distinguish between frankenstein digits and normal digits is by their log-probability. Naturally we expect normal images to be more likely (higher log-probability) as the model is trained to assign more probability mass in the latent space to these examples. However, if a "normal" image is in fact an outlier, it is likely to have a low probability as well. It is interesting to note that some frankenstein images in our case look similar to normal images, and therefore have a higher log probability. A more conclusive way to tell them apart would be to conduct statistical tests on some statistic of frankenstein and normal images (for example their log probability). However designing the appropriate statistical test may be difficult.

## 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  $X$ :  $p(X_d|Z)$

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

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

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:

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

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: Number of terms in  $|Z|$**  Given the dimensionality of the space  $Z = \{0, 1\}^M$ , there are  $2^M$  terms in the sum (possible configurations for any vector  $z \in Z$ ).

**B: Complexity of Sigmoid Belief Nets** For the inference procedure (forward pass), we outline the algorithm and the respective complexity of each step below. Due to the distributed representation, the complexity of the inference procedure is rather high and exponential in the dimensionality of the latent space, namely  $O(MND2^M)$ . The time complexity of computing the gradients and applying the updates are linear in the dimensionality of the parameters, and thus are dominated by the inference complexity.

```
procedure SBN INFERENCE STEP( $w, b, c$ )  
  for  $i$  in  $1, \dots, 2^M$  do  $\triangleright O(2^M)$   
    cache  $\leftarrow p(z_i) = \prod_{j=1}^M \text{Bern}(z_j^{(i)} | \text{sigm}(b_j))$   $\triangleright O(M)$   
    for  $x^{(n)} \in X$  do  $\triangleright O(N)$   
      for  $i$  in  $1, \dots, 2^M$  do  $\triangleright O(2^M)$   
        Retrieve from cache  $z_i, p(z_i)$   $\triangleright O(1)$   
         $\theta \leftarrow z_i \cdot w_{.,d} + c_d$   $\triangleright O(M)$   
        Add cache for  $x_n \leftarrow p(z_i)p(x^{(n)} | \text{sigm}(\theta))$   $\triangleright O(D)$ 
```

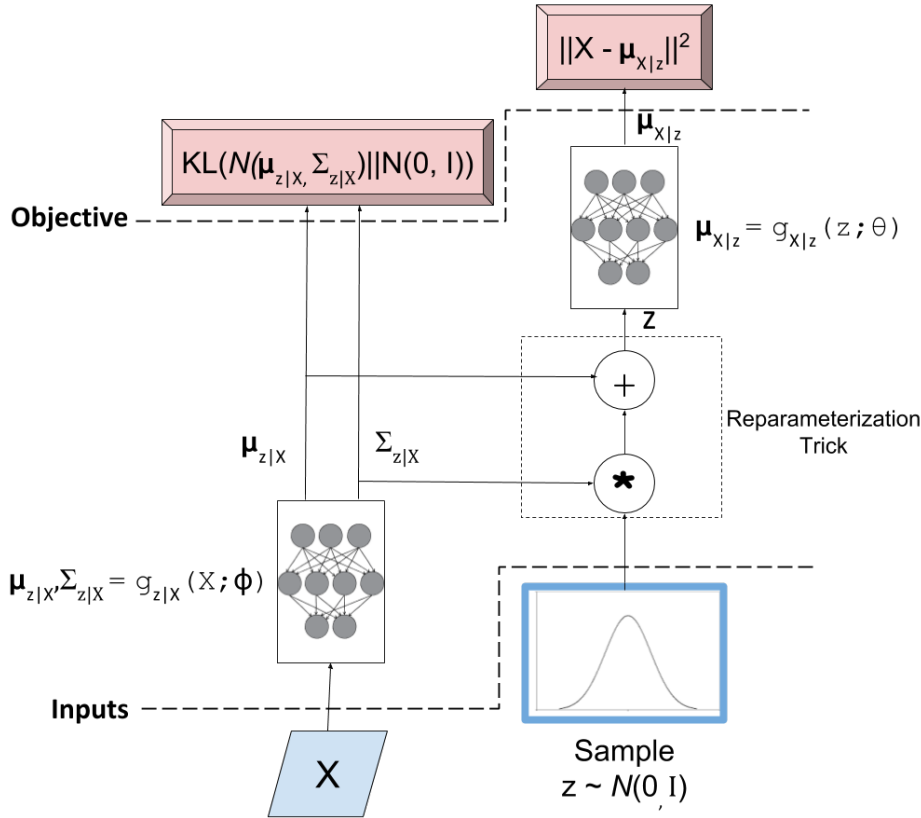
**C: Complexity of Naive Bayes** As we assume that any datapoint  $x \in X$  is generated by one category of  $z \in Z$ , the complexity of the naive bayes model is  $O(NKD)$  which is considerably lower than the sigmoid belief nets.

```
procedure NB INFERENCE STEP( $w, b, c$ )  
   $\omega \leftarrow \text{softmax}(b)$   $\triangleright O(K)$   
  for  $x^{(n)} \in X$  do  $\triangleright O(N)$   
    for  $k$  in  $1, \dots, K$  do  $\triangleright O(K)$   
      Evaluate  $p(z = k | \omega)$   $\triangleright O(1)$   
       $\theta \leftarrow w_{k,d} + c_d$   $\triangleright O(1)$   
      Add cache for  $x_n \leftarrow p(z = k | \omega)p(x^{(n)} | \text{sigm}(\theta))$   $\triangleright O(D)$ 
```

Clearly, the Naive Bayes inference procedure is considerably cheaper than the Sigmoid Belief Nets.

## 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



**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 for the MNIST data).

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 is 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.

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

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

Where:

- $D_Z$  is the dimensionality of the latent space (e.g. 5)
- $D_X$  is the dimensionality of the data (e.g. 784)
- $\mathcal{N}(0, 1)^{D_Z}$  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  $\theta$  that takes a sample  $z$  and outputs the the mean of the Bernoulli distributions for each pixel in X
- $\theta$  represents all the parameters for the neural network  $\mu_{X;\theta}$ .

### 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  $z \sim \mathcal{N}(0, 1)^{D_Z}$
2. For each dimension  $d$  of  $x$ , obtain the parameter  $\mu_{X;\theta}(z)_d$  by doing a forward pass of  $z$  through the decoder network
3. Sample each binary pixel  $x_d$  using the obtained parameter  $x_d \sim \text{Bern}(x_d|\mu_{X;\theta}(z)_d)$

**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**

The choice of the standard normal distribution over the latent variables allows us to easily sample from it (and furthermore the choice of a Gaussian in general as the variational posterior makes the optimization problem much simpler). As long as we are able to use a transformation which maps our latent variable  $Z$  to any other latent variable that has a more appropriate distribution for generating our data, the choice of initial distribution does not matter for the most part. Such transformations, albeit complex, can be learned by sufficiently expressive neural networks.

Now, let's 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).

$$\log p(X) = \log \int_z p(X, z) dz \quad (13)$$

$$= \log \int_z p(z) p(X|z) dz \quad (14)$$

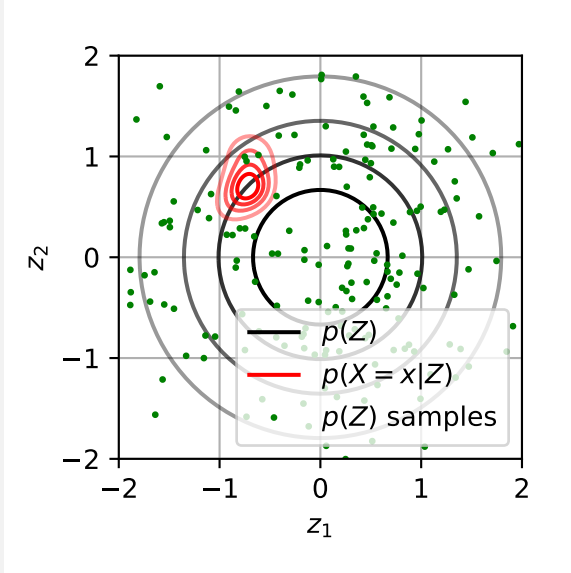
$$= \log \mathbb{E}_{z \sim p(Z)} [p(X|z)] \quad (15)$$

**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** We draw  $M$  samples from  $Z$  and denote them by  $z_m$  and use MC integration to approximate the expectation.

$$\log p(x) = \log \mathbb{E}_{z \sim p(Z)}[p(X|z)] \approx \log\left(\frac{1}{M} \sum_m^M p(x|z_m)\right)$$

**B** The regions in the latent space that are likely to generate desirable  $X$ 's are concentrated and compact, as shown in the picture. This means that most  $z$ 's will not be useful for generating good samples of  $X$ . The "volume" of these useful regions reduce exponentially relative to the dimensionality of the latent space. This makes random sampling from  $Z$  high inefficient as most samples  $z$  will produce a very low probability for  $p(X=x|z)$  (for any observed input  $x$ ) and thus will fail to generate good samples of  $X$ . Ideally, we wish to only focus on these ideal regions and sample  $z$  from those.

## 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:

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

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  $(\mu_q, \sigma_q^2)$ : 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

**A and B** We have the following result for the KL-divergence of two Gaussians. Given that the support of both distributions match and is  $\mathbb{R}$ , this quantity is well-defined (i.e. never goes to infinity when  $p(x) = 0$ ).

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

We observe that for a small divergence,  $\mu_1$  must be very close to  $\mu_2$  and similarly for  $\sigma_1^2$  and  $\sigma_2^2$ . For a large divergence, having different mean parameters is sufficient (due to the quadratic term).

For the particular case where  $\mu_2 = 0$  and  $\sigma_2^2 = 1$ , taking  $\mu_1 = 0$  and  $\sigma_1^2 = 1$  gives the a divergence of 0. Setting  $\mu_1 \gg 0$  and  $0 < \sigma_1^2 < 1$  results in a large divergence.

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

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:

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

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

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

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

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

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:

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

#### Problem 15 (5 Points)

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

#### Answer 15

Because the KL divergence is non-negative, we have that  $\log p(X) \geq \mathbb{E}_{z \sim q(Z|X)} [\log p(X|z)] - \mathcal{D}(q(z|X) \| p(z))$ .

#### 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 we do not know the true posterior distribution  $p(z|x)$ , and cannot compute it due to intractability issues. Therefore, we cannot evaluate  $\mathcal{D}(q(z|X) \| p(z|X))$ .

#### 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

Analogous to the EM algorithm, when we minimize the complexity cost  $\mathcal{D}(q(z|X) \| p(z))$  with respect to the variational parameters (E-step), our lower bound increases since the KL divergence is always non-negative. When we minimize the reconstruction error  $\mathbb{E}_{z \sim q(Z|X)} [\log p(X|z)]$  (M-step) we further increase the lower bound. This also has an effect on  $\log p(X)$  as it depends on our network, i.e. in minimizing the reconstruction error we are able to better model the training data and therefore  $\log p(X)$  also increases. However, as we may never recover the true posterior distribution given our modelling choices, the term  $\mathcal{D}(q(z|X) \| p(z|X))$  is greater than zero, and so the right hand side remains a lower bound.

### 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.

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

Where:

- $x_n \in \mathbb{R}^{D_x}$  is the a particular data sample
- $\mu_{Z;\phi}(X) \in \mathbb{R}^{D_z}$  is the mean of the gaussian  $q(Z)$  for a given  $X$
- $\Sigma_{Z;\phi}(X) \in \mathbb{R}_{>0}^{D_z}$  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  $\Sigma$ 's are positive, they are computed with an exponential nonlinearity:  $f_\Sigma(x) = \exp(x) \in \mathbb{R}^{D_z}$  at the last layer.

#### Problem 18 (6 Points)

The loss in Equation 22:

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

can be rewritten in terms of per-sample losses:

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

Where:

$\mathcal{L}_{recon,n} := \mathbb{E}_{z \sim q(Z|X=x_n)} [\log p(X = x_n|z)]$  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

The reconstruction error quantifies how well we are able to "reconstruct" the training data given the latent variables, i.e. it reflects on the quality of our learned latent variables. A good  $z$  gives a high probability  $p(X|z)$  for any observation  $X$ .

The regularization term quantifies how well our latent representation matches our prior beliefs about  $Z$ . Minimizing this term moves our variational posterior to be closer to the prior  $p(z)$ , which corresponds to regularization and is analogous to weight decay in MAP learning.

### 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

We sample  $z_l \sim q(Z|X = x_n)$  for  $l \in \{1, 2, \dots, L\}$ .

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

By using the encoder network, and noticing that the variational distribution has a diagonal covariance matrix, we obtain  $\mu_i = \mu_{Z;\phi}(x_n)_i$  and  $\sigma_i^2 = \Sigma_{Z;\phi}(x_n)_{ii}$ .

$$\mathcal{L}_{reg,n} := \mathcal{D}(q(z|X = x_n) || p(z)) = \sum_{i=1}^{D_z} -\log \sigma_i + \frac{\sigma_i^2 + \mu_i^2 - 1}{2}$$

## 6.4 The Reparametrization Trick

Note that we're not quite done yet. We use sampling to approximate the term  $E_{q_\phi(Z|X)} [\log p(X|Z)]$ , 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** Obtaining the gradient of the loss with respect to the variational parameters  $\phi = \{\mu, \sigma^2\}$  allows us to optimize these variables using gradient descent. This allows us to use variational inference to learn a variational distribution that approximates the true posterior. This optimization can be done with gradient descent which is why we need this gradient.

**B** Direct sampling from the variational distribution  $q_\phi(Z|X)$  is a non-differentiable operation, and does not allow us to take gradients of the lower bound (which involves expectations over  $q_\phi(Z|X)$ ) with respect to the variational parameters  $\phi$ .

**C** If we can find a transformation from samples of some random variable (which we do not wish to learn) to samples of our variational distribution (which we wish to learn), then we no longer need to sample directly from the variational distribution and this overcomes the non-differentiability problem.

We require this transformation to be invertible, and furthermore it must absorb the variational parameters, i.e. be parameterized by  $\phi$  such that we can take its gradients with respect to  $\phi$ . Therefore, by using this transformation, we can obtain samples from the variational posterior and take gradients with respect to the parameters that govern the variational distribution.

The reparameterization trick can be used on certain distributions for which there are known parameterizable transformations such as members of the location-scale family (more examples in the original VAE paper). The trick involves manipulations of the expectations in the lower bound such that we can use Leibniz's Integral Rule to take the gradient of terms inside an integral. Briefly put, this is performed by using the change of variable technique, followed by a change of density. Conveniently, the two Jacobians that arise are the Jacobians of the transformation and its inverse, which cancel out. Once we have performed the reparameterization trick, we can separate the non-differentiable components in our computational graph from the variables that we wish to learn.

## 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 convenient 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.

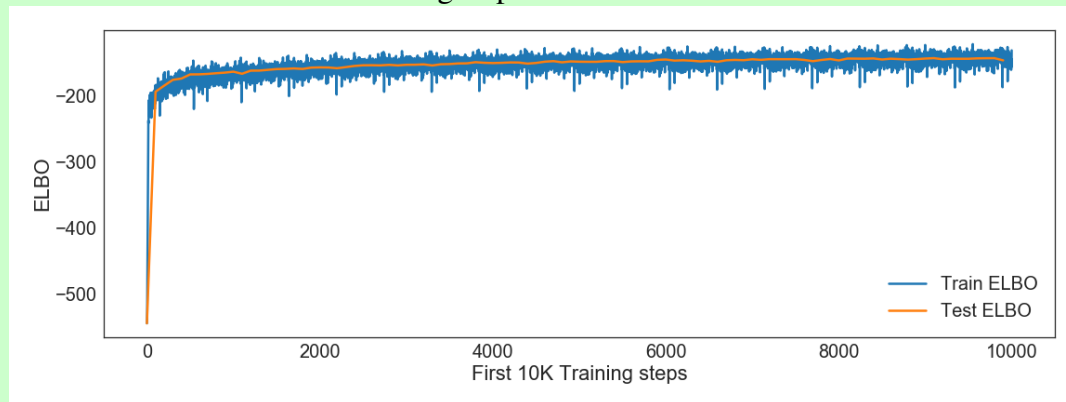
### Problem 22 (10 Points)

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

ELBO across the first 10K training steps.



### Problem 23 (10 Points)

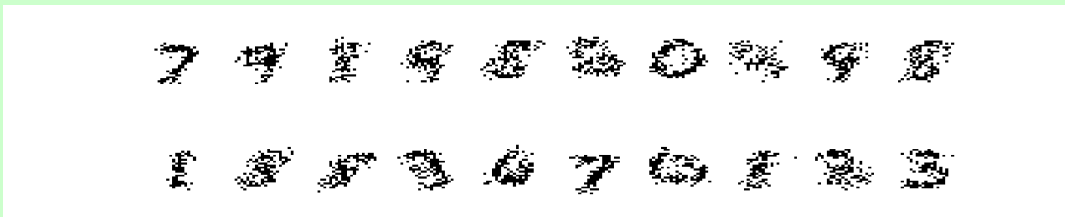
Plot samples from your model at 3 points throughout training (first, before any training, next: part way through, finally: after you finish training). You should observe an improvement in the quality of samples.

### Answer 23

At training step 100



At training step 4500



At training step 27400



### 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

Manifold

0	0	0	0	0	4	9	7	7	7
0	0	0	0	6	4	9	7	7	7
0	0	0	0	6	9	9	7	7	7
0	0	0	0	2	9	4	7	7	7
0	0	0	6	2	3	9	7	7	7
2	2	2	2	3	8	9	7	7	7
2	2	2	5	8	8	3	7	7	7
2	2	5	5	1	1	1	1	1	7
2	5	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Total Points: 204