Assignment 3: Generative Models

Chouliaras George
g.chouliaras@student.vu.nl

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.

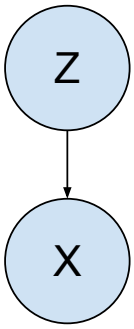


Figure 2: A general latent variable model. Each data point X is considered to be generated by a "latent" variable Z.

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.

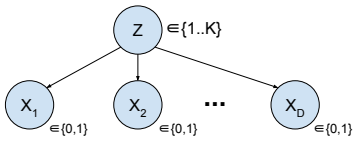


Figure 3: A Naive Bayes model with a Categorical Latent Variable Z (with K categories) and Bernoulli Visible variables X_i. Note that we have redrawn the graph to reveal our *Conditional Independence* (i.e. Naive) assumption: X_i is independent of all other elements of X ($X_{\setminus i}$) given Z.

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 $X_1 ... X_D$, 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:

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

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

Our goal is to maximize the probability of the data given the model ($p(X|Z)$). Since we have a batch of data, we use the marginal distribution of the data ($p(X = x_n)$) to 'access' $p(X|Z)$, so as to maximize it.
To achieve that, the 1st step is to write the marginal distribution as a summation of the joint distribution $p(X, Z)$. We do that using the sum rule of probabilities (marginalization):

$$p(x) = \sum_{y \in Y} p(x, y)$$

Now that we have expressed $p(X)$ as a summation of joint distributions we can use the product rule, to access $p(X|Z)$. The product rule is the following

$$p(x, y) = p(x|y)p(y)$$

In other words, we express the joint distribution as a product of the likelihood $p(x|y)$ and the prior $p(y)$. In our case $p(Z = k)$ is the prior belief on the latent variables and $p(X = x_n|Z = k)$ is the likelihood of the data, given the latent codes.
Since each data example is $D-$dimensional, a general distribution would correspond to a table of 2^D numbers for each category K, leading to a total of $(2^D - 1)K$ parameters. However, this grows exponentially with D, which in our case is large (784), so we need a more restricted representation. We achieve this, by using the Naive Bayes assumption which states that the data points x_n are treated as independent conditioned on the category K (conditional independence). This allows the likelihood to be written as a product

$$p(X|Z) = \prod_{d=1}^{D} p(X_d = x_{n,d}|Z = k)$$

which reduces the number of parameters from $(2^D - 1)K$ to DK.

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

There are two ways to calculate $\log p_{total} = \log\left(\prod_{n=1}^{N} p_n\right)$:

1. The first way is to compute the logarithm directly, by first evaluating the product and then the logarithm:

$$\log p_{total} = \log(p_1 \cdot p_2 \cdot \ldots p_N)$$

2. The second way is to use the logarithm property $log(ab) = loga + logb$ in order to turn the product into a summation

$$\log p_{total} = \sum_{n=1}^{N} log(p_n)$$

Although these two ways seem equivalent, the 1st one is prone to numerical underflow, when N is large. This is due to the fact that, when N is large, we multiply many small numbers (between 0 and 1) and this results to a number which is almost 0, something that leads to $log0 = -\infty$. This can be verified by making a Python script for the two aforementioned cases, as can be seen in Figure 4. In this figure, we see that when N is small ($N = 5$), the two methods lead to the same result. However, when N is large ($N = 1000$), the first way leads to $-\infty$ as explained earlier. On the other hand, we see that the second way is numerically stable. This is because we turned the logarithm of a product, to a summation of logarithms, something that prevents numerical underflow.

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

In [2]: #PROBLEM 1

        #SMALL N
        #Set the size of N
        N = 5
        #Create vector of probabilities of size N
        probs = np.random.uniform(low = 0.0, high = 1.0, size = N)

        case1 = np.log(np.prod(probs))
        case2 = np.sum(np.log(probs))

        print('Case 1 result for small N: {0:.2f}'.format(case1))
        print('Case 2 result for small N: {0:.2f}'.format(case2))

        Case 1 result for small N: -4.00
        Case 2 result for small N: -4.00

In [3]: N = 1000

        #LARGE N
        #Create vector of probabilities of size N
        probs = np.random.uniform(low = 0.0, high = 1.0, size = N)

        case1 = np.log(np.prod(probs))
        case2 = np.sum(np.log(probs))

        print('Case 1 result for large N: {0:.2f}'.format(case1))
        print('Case 2 result for large N: {0:.2f}'.format(case2))

        Case 1 result for large N: -inf
        Case 2 result for large N: -988.95
```

Figure 4: Python script for numerical stability of the logarithm

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

By taking the logarithm in Equation 5 we get

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

We then use the logarithmic property $x = \exp(\log(x))$ to get a logarithm inside and turn the product into summation

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

To further improve numerical stability we can deploy the Log-Sum-Exp trick which exploits the identity:

$$\log \sum_{n=1}^{N} \exp x_n = \alpha + \log \sum_{n=1}^{N} \exp(x_n - \alpha)$$

In our case, we use

$$\alpha = \min_k \left(\log p(Z=k) + \sum_{d=1}^{D} \log p(X_d = x_{n,d} | Z = k)) \right)$$

Problem 4 (5 Points)

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

$$\log p(X) = \log(\sum_{k=1}^{K} \exp(\log \text{Cat}(Z = k; \text{softmax}(b)_k) + \\ + \sum_{d=1}^{D} \log \text{Bern}(X_d = x_{n,d}; \text{sigm}(w_{k,d} + c_d))))$$

By using $\text{Bern}(x; \theta) := x\theta + (1 - x)(1 - \theta)$ for $x \in \{0, 1\}$ we can write the result as follows

$$\begin{aligned}
\log p(X) &= \log(\sum_{k=1}^{K} \exp(\log \text{Cat}(Z = k; \text{softmax}(b)_k) + \\
&\quad + \sum_{d=1}^{D} \log \left(X_d \text{sigm}(w_{k,d} + c_d) + (1 - X_d)(1 - \text{sigm}(w_{k,d} + c_d))\right)) \\
&= \log(\sum_{k=1}^{K} \exp(\log \text{Cat}(Z = k; \text{softmax}(b)_k) + \\
&\quad + \sum_{d=1}^{D} \log \left(X_d \frac{1}{1 + e^{-(w_{k,d} + c_d)}} + (1 - X_d)(1 - \frac{1}{1 + e^{-(w_{k,d} + c_d)}})\right)
\end{aligned}$$

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

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

After training the model for 20 epochs (11k training steps), we can plot images of the expected pixel values given the latent variable. For each of the $K = 20$ categories, we find $x_{k,i} = \text{sigm}(w_{k,i} + c_i)$, where $x_{k,i}$ is the $i'th$ pixel in image k. Figure 5 depicts the expected pixel values for all the categories of z. We see that the model has assigned categories not only to different digits, but also to different forms of digits. For example, categories 0, 9 and 10 represent the digit '1', but in different writing style. The same goes for categories 8 and 18 which represent different styles of the digit '6'. We observe that although we trained the model for 20 epochs, it was not able to capture a clear form for the digits '4' and '5'. This is probably due to the fact that these digits have more complicated structure than the rest.

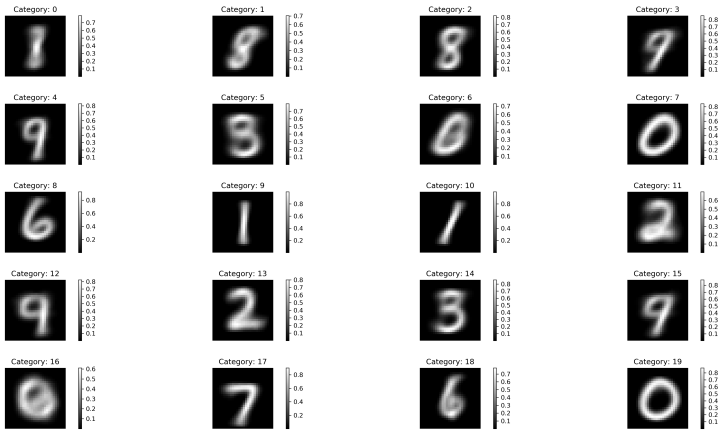


Figure 5: Expected pixel values for K=20 categories of z.

Problem 7 (5 Points)

Show 16 images samples from your trained model.

Answer 7

Except of the expected pixel values for given z, we can also plot images sampled from our model after training. To do so, we create images by sampling $x_{k,i} \in \{0, 1\}$ where $x_{k,i} \sim \text{Bern}(\text{sigm}(w_{k,i} + c_i))$ is the i_{th} pixel of an image of category k. Figure 6 depicts 16 digits sampled from the model after training, along with the corresponding categories. By comparing figures 5 and 6 we see that the sampled digits for a specific category are similar with their expected pixel values. For example, we see that the digit sampled from category 7, is indeed a 0 and has very similar form with the expected pixel values for $k = 7$ in Figure 5. We also note that we observed an improvement on the quality of the samples as the training was evolving, however after some point the quality did not improve further.

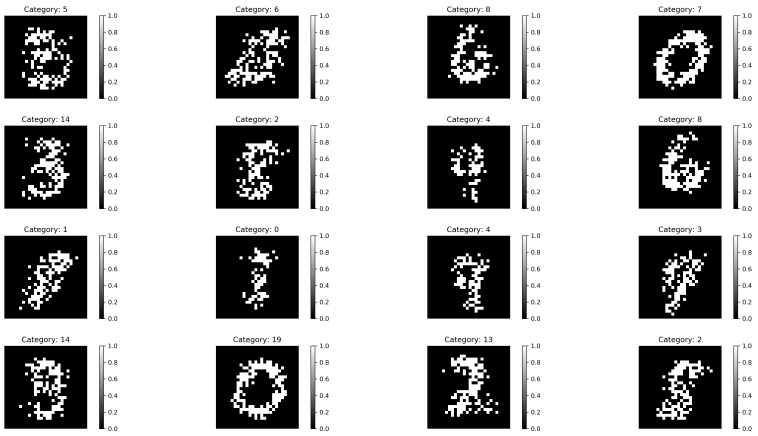


Figure 6: 16 sampled digits after training.

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

We can keep track of the model's performance by frequent evaluations of the log-probability on the first 1000 samples of the training and test sets. More specifically we compute and plot the mean over samples $\frac{1}{N}\sum_{n=1}^{N}\log p(X = x_n)$ every 100 training steps. Figure 7 shows the mean over the first 1000 samples both for train and test set. We should note that normally one would expect $\log p(x_{train}) > \log p(x_{test})$, however in Figure 7 we see that the log-probability of the test set is slightly higher than the one in the training set. The most likely reason is that the examples in the test set are slightly easier for the model than those in the training set. Moreover, we observe that there is not an indication of overfitting, as we do not see any pattern of high training log-probability and low test log-probability even after many epochs. This is due to the simplicity of the Naive Bayes model which makes it unable to specialize enough in the training examples.

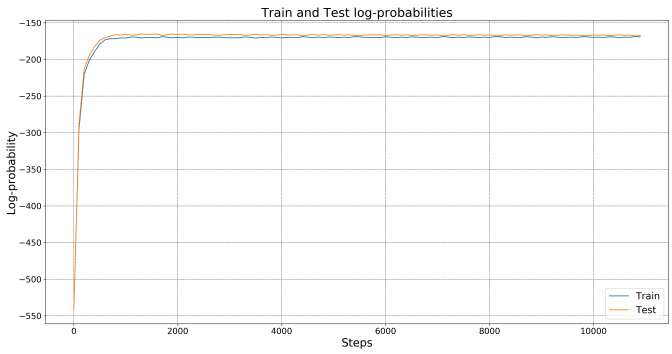


Figure 7: Learning curves of training and test log-probabilities.

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

In order to create 'Frankenstein' digits, we first sample 10 digits from our model and then we randomly merge their left and right parts to create new digits. Figure 8 shows 10 'Frankenstein' digits, created by merging parts of the unmodified digits shown in Figure 9. Moreover, both figures depict the log-probabilities assigned to the corresponding digits. The code which creates these figures can be found in the file `frankenstein.py`. By comparing the two figures, we see that the 'Frankenstein' digits have smaller log-probabilities than the unmodified digits. This is due to the fact that these modified digits have different forms from the digits which the model was trained on, and hence it assigns smaller log-probabilities as they are less likely to occur. As an example, let's look at the top left 'Frankenstein' digit. This digit was created by merging the left part of the $4th$ unmodified digit (resembles a '2') and the right part of the last unmodified digit (resembles a '9'). We observe that these two digits have log-likelihoods -183.77 and -138.81 respectively, however the top left 'Frankenstein' digit has a log-likelihood -233.69 which is smaller due to its modified form.
One way to use the model to reliably verify if a batch of digits are 'Frankenstein' digits is the following. Since the modified digits have generally lower log-probabilities, we could find the mean value of the log-probabilities of the digits for the batch that we want to check. If this mean value is smaller than -200, then we know that the batch contains digits with small log-probabilities which most likely are modified digits. On the other hand if the mean is relatively larger than -200 (around -150 or larger) then the batch contains unmodified digits. In this way, one can reliably verify if a batch consists of 'Frankenstein' digits or not.

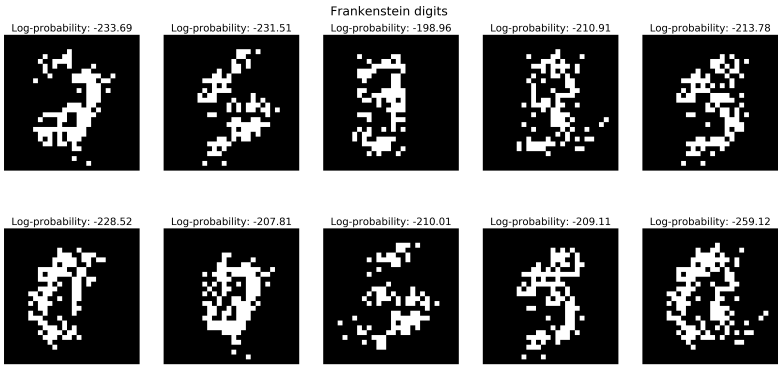


Figure 8: Frankenstein digits along with log-probabilities for each digit.

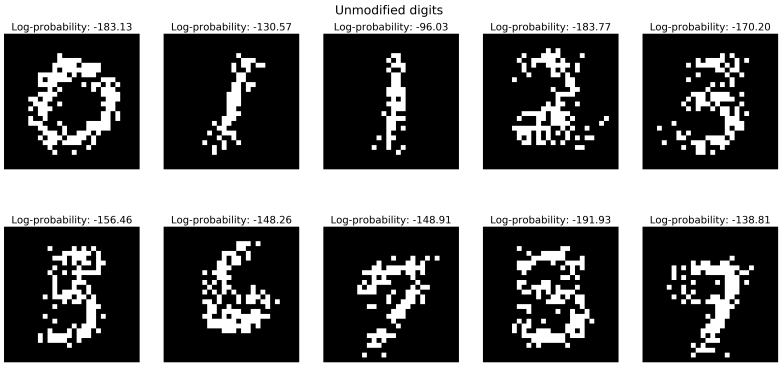


Figure 9: Unmodified digits along with log-probabilities for each digit.

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) The latent variable Z is M-dimensional, $Z \in \{0, 1\}^M$. This means that we have 2^M possible values for the latent code z. We note that this number grows exponentially with the dimensionality of the latent space.
(B) In each training step and for one sample, we have 2^M computations for Z and D computations for $p(X|Z)$. This means that the time complexity for one sample is $O(2^M D)$. Since we have N samples, the total time complexity for each training steps is $O(2^M DN)$.
(C) In the Naive Bayes model, the total complexity is $O(KDN)$, which is smaller than $O(2^M DN)$, because usually $K < 2^M$.

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

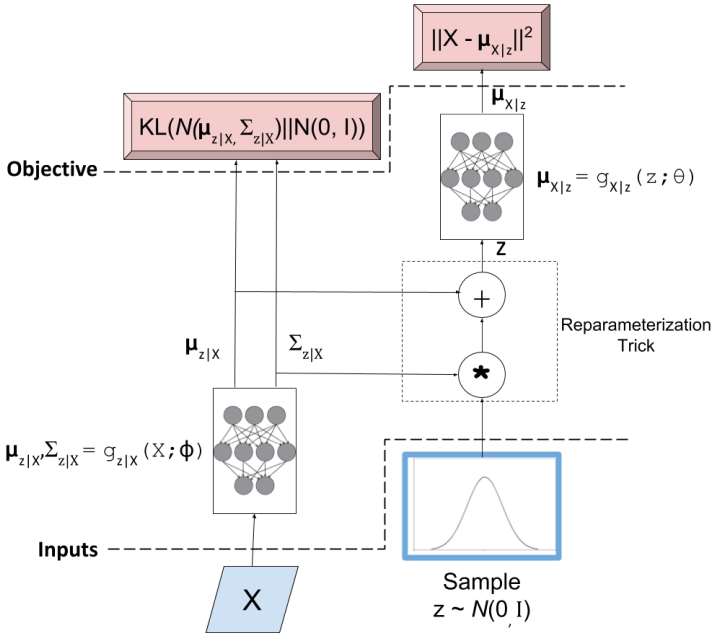


Figure 10: A schematic of a VAE (taken from Brian Keng's website). Note that in their model, X is considered a Gaussian-distributed Random Variable, whereas in ours we consider it to be Bernoulli-Distributed (which is more appropriate the the MNIST data).

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 10.

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 $\mu \in [0,1]^{D_X}$
$\mu_{X;\theta} : D_Z \times |\theta| \to D_X$ 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 $\mu_{X;\theta}$.

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

The sampling procedure can be formulated as follows:

1. Sample z from $\mathcal{N}(0,1)^{D_z}$.
2. Compute $\mu_{X;\theta(z)}$ using the sampled z.
3. Sample each pixel value X_d 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

The model makes the assumption that the latent variables have a standard normal distribution. While this might seem restrictive, actually it is not. As it is stated in Carl Doersch's tutorial, the reason is that we can generate any distribution in d dimensions by taking a set of d normally distributed variables and mapping them through a sufficiently complicated function. As we see in Equation 12, we use a neural network $\mu_{X;\theta}(z)$ which learns to map our independent normally distributed values of z to the appropriate mean of the Bernoulli distribution, in order to get the likelihood of X given z.

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).

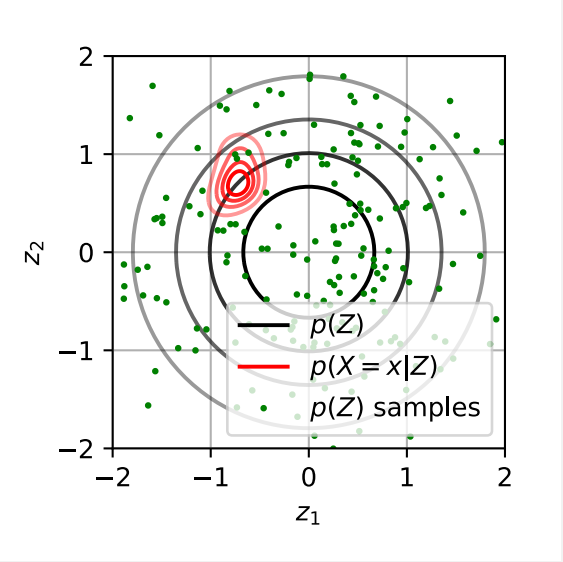
$$\log p(X) = \log \int_z p(X, z) dz \tag{13}$$

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

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

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) We can approximate $\mathbb{E}_{z\sim p(Z)}[p(X|z)]$ using sampling as follows:

$$\mathbb{E}_{z\sim p(Z)}[p(X|z)] \coloneqq \int p(Z)p(Z|z)dz \approx \frac{1}{M}\sum_{m=1}^{M} p(X|z_m)$$

Hence, an expression for approximating $\log p(x)$ by sampling is

$$\log p(X) = \log\left(\frac{1}{M}\sum_{m=1}^{M} p(X|z_m)\right)$$

(B)
The problem with this approach is that z_m is a D_z-dimensional vector. Each additional dimension of z exponentially increases the number of samples we need to properly approximate the volume of the space. This means that if D_z is large, we must sample over a very large number of z values, so as to approximate the integral in Equation 14 properly. Furthermore, this approach is inefficient because for each x, we must average over a large number of samples (M). As we see in the given figure, for a given observation $X = x$, most of the z_m samples contribute very little to the likelihood (most of the green points lie outside the red area). Due to this, we are wasting a lot of computing time and power in order to average over all these samples, while in fact we only need the z values which contribute a significant amount to the likelihood. By combining this waste of time and power with a highly dimensional z, we can realize that this method is inefficient.

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

(A) The normal distributions can be expanded as follows

$$\mathcal{N}(\mu_q, \sigma_q^2) = q = \frac{1}{\sqrt{2\pi\sigma_q^2}} e^{-\frac{(x-\mu_q)^2}{2\sigma^2}}$$

$$\mathcal{N}(0,1) = p = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

Thus the KL divergence becomes

$$\begin{aligned}
\mathcal{D}(q\|p) &= \int \frac{1}{\sqrt{2\pi\sigma_q^2}} e^{-\frac{(x-\mu_q)^2}{2\sigma^2}} \log \left(\frac{\frac{1}{\sqrt{2\pi\sigma_q^2}} e^{-\frac{(x-\mu_q)^2}{2\sigma^2}}}{\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}} \right) dx \\
&= \frac{1}{\sqrt{2\pi\sigma_q^2}} \int e^{-\frac{(x-\mu_q)^2}{2\sigma^2}} \log \left(\frac{1}{\sigma_q} e^{-\frac{(x-\mu_q)^2}{2\sigma^2} + \frac{1}{2}x^2} \right) dx \qquad (17)
\end{aligned}$$

From Equation 17 we see that if $\mu_q = 0$ and $\sigma_q = 1$, the quantity in the logarithm becomes 1 and hence, $log1 = 0$. So in this case the KL divergence is 0, since the two distributions are identical. The more far we move from $\mu_q = 0$ and $\sigma_q = 1$, the larger the KL divergence will be, as the two distributions will be less similar. For example, if $\mu_q = 4$ and $\sigma_q = 3$, the KL divergence will be

$$\mathcal{D}(q\|p) = \frac{1}{\sqrt{18\pi}} \int e^{-\frac{(x-4)}{18}} \left(-log3 - \frac{(x-4)^2}{18} + \frac{1}{2}x^2 \right) dx$$

which results in a large KL divergence.

(B)

$$\begin{aligned}
\mathcal{D}(\mathcal{N}(\mu_q, \sigma_q^2)\|\mathcal{N}(0,1)) &= \int \mathcal{N}(\mu_q, \sigma_q^2) \log \frac{\mathcal{N}(\mu_q, \sigma_q^2)}{\mathcal{N}(0,1)} dx \\
&= -\int \mathcal{N}(0,1) \log \mathcal{N}(\mu_q, \sigma_q^2) dx + \\
&+ \int \mathcal{N}(0,1) \log \mathcal{N}(0,1) dx \\
&= \frac{1}{2} \log(2\pi 1^2) + \frac{\sigma_q^2 + (\mu_q - 0)^2}{2 \cdot 1^2} - \frac{1}{2}(1 + \log(2\pi\sigma_q^2)) \\
&= \frac{1}{2} \log 2\pi + \frac{\sigma_q^2 + \mu_q^2}{2} - \frac{1}{2} \log 2\pi - \frac{1}{2} \log \sigma_q^2 \\
&= -\log \sigma_q + \frac{\sigma_q^2 + \mu_q^2}{2} - \frac{1}{2}
\end{aligned}$$

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:

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

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

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

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

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

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:

$$\mathcal{L} = -\frac{1}{N}\sum_n \left(\mathbb{E}_{z\sim q(Z|X)}\left[\log p(X|z)\right] - \mathcal{D}(q(z|X)\|p(z))\right) \tag{23}$$

Problem 15 (5 Points)

Why is the right-hand-side of Equation 22 called the *lower bound* on the log-probability?

Answer 15

From Equation 22 we see that the lower bound equals

$$\text{lower bound} = \log p(X) - \mathcal{D}(q(z|X)\|p(z))$$

One property of the KL divergence is that it is always non-negative

$$\mathcal{D}(q(z|X)\|p(z)) \geq 0 \tag{24}$$

By multiplying Equation 24 with -1 and by adding $\log p(X)$ we get

$$\log p(X) - \mathcal{D}(q(z|X)\|p(z)) \leq \log p(X) \tag{25}$$

where the equality holds when $\mathcal{D}(q(z|X)\|p(z)) = 0$ and the inequality holds when $\mathcal{D}(q(z|X)\|p(z)) > 0$. Due to the form of Equation 25, the quantity in the left-hand-side is called the lower bound on the log-probability $\log p(X)$ (right-hand-side).

Problem 16 (5 Points)

Looking at Equation 22, why must we optimize the lower-bound, instead of optimizing the log-probability directly?

Answer 16

In order to optimize $\log p(X)$ directly, we should optimize

$$\log p(X) = \log \frac{p(X, Z)}{p(Z|X)}$$

which means that we must evaluate the posterior $p(Z|X) = \frac{p(X,Z)}{p(X)}$. However the posterior is intractable and to solve the problem of intractability we can use the sampling procedure: sample a large number of z values and compute $p(X) \approx \frac{1}{n} \sum_i p(X|z_i)$. Using this procedure, another problem arises: if our space is highly dimensional, then the number of samples z must be extremely large so as to have an accurate estimate of $p(X)$. To overcome this problem, we should only sample the z values that have a large contribution on the likelihood. To achieve that, we sample from a variational distribution $q(Z|X)$ with which we approximate the intractable posterior. To measure how well we approximate the posterior we use the KL divergence. Finally, we reformulate our problem as in Equation 22 and we optimize the lower bound which is the right-hand-side. We do this because the right-hand-side includes quantities that can be directly computed and maximized. By maximizing the lower bound we also maximize the log-probability (something that before was impossible) and this is the reason why we reformulated our problem.

Problem 17 (6 Points)

Now, looking at the two terms on left-hand side of 22: Two things can happen when the lower bound is pushed up. Can you describe what these two things are?

Answer 17

From Equation 25 we can see that when the lower bound $\log p(X) - \mathcal{D}(q(z|X)\|p(z))$ is pushed up, it is closer to the log-probability $\log p(X)$ which is what we want to optimize. Hence, this means that our generator is improved.
At the same time, if the lower bound is pushed up, it means that $\mathcal{D}(q(z|X)\|p(z))$ is closer to 0 and the approximate posterior $q(Z|X)$ is closer to the true posterior $p(Z|X)$. Thus, this means that our latent representation is getting better.

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.

$$q_\phi(Z|X = x_n) = \mathcal{N}(Z; \mu_{Z;\phi}(x_n), \Sigma_{Z;\phi}(x_n)) \tag{26}$$

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
$\Sigma_{Z;\phi}(X) \in \mathbb{R}^{D_Z}_{>0}$ 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 10). To ensure that Σ'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 23:

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

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

Maximizing the term $\mathbb{E}_{z\sim q(Z|X=x_n)}[\log p(X=x_n|z)]$, means that given an input Z and an output $X=x_n$, we want to maximize the conditional distribution $p(X|Z)$ under the model parameters. This loss term shows how accurately the output (X) replicates/reconstructs the input z and hence it can be seen as a reconstruction loss.
The term $\mathcal{D}(q(z|X=x_n)\|p(z))$ is the KL divergence of the prior $p(z)$ from the approximate posterior $q(Z|X)$ over the latent space. This measures the extent to which the model must update its parameters, so as to accommodate new observations. By forcing the approximated posterior $q(Z|X)$ to match the prior $p(z)$, this term makes the model more robust to small perturbations along the latent manifold. Due to this, it can be seen as a regularization term.
Together, these two loss terms capture the trade-off between accurate reconstruction and similarity to the prior. That is, the reconstruction loss pushes the output to replicate the input, while the regularization term regularizes the reconstruction loss such that the latent representation $q(Z|X)$ is not far from the prior $p(z)$.

Problem 19 (10 Points)

Now, putting together your model definition (Equations 11, 12), your variational distribution definition (Equation 26), 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

For the reconstruction loss we have

$$\begin{aligned}
\mathcal{L}_{recon,n} &= -\mathbb{E}_{z\sim q(Z|X=x_n)}\left[\log p(X=x_n|z)\right] \\
&= -\mathbb{E}_{z\sim q(Z|X=x_n)}\left[\log \prod_{d=1}^{D_x} \text{Bern}(X_d = x_{n,d}; \mu_{X;\theta}(z))\right] \\
&= -\mathbb{E}_{z\sim q(Z|X=x_n)}\left[\sum_{d=1}^{D_x} \log \text{Bern}(X_d = x_{n,d}; \mu_{X;\theta}(z))\right] \\
&\overset{\text{sampling}}{=} -\frac{1}{M}\sum_{m=1}^{M}\sum_{d=1}^{D_x} \log \text{Bern}(X_d = x_{n,d}; \mu_{X;\theta}(z_m))
\end{aligned}$$

where M is the number of samples we use to approximate the expectation.
For the regularization loss we use the formula from problem 14 and we have

$$\mathcal{L}_{reg,n} = -\sum_{i=1}^{D_z} \log \Sigma_{z_i;\phi}(x_n) + \frac{\Sigma_{z_i;\phi}(x_n) + \mu_{z_i;\phi}(x_n)}{2} - \frac{1}{2}$$

6.4 The Reparametrization Trick

Note that we're not quite done yet. We use sampling to approximate the term $E_{q_\phi(Z|X)}\left[\log p(X|Z)\right]$, 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) In order to train the model we must apply backpropagation to learn the parameters of the model. To do so, we need the gradients both for the encoder (∇_ϕ) and the decoder (∇_θ). Computing $\frac{\partial \mathcal{L}}{\partial \phi}$ is essential in order to learn the model parameters ϕ through gradient based methods.
(B) While for the decoder we are able to get the gradients, we cannot get $\frac{\partial \mathcal{L}}{\partial \phi}$ for the encoder, because we must backpropagate the error through a layer that samples z from $q(Z|X)$, which is a non-continuous operation and hence, it has no gradient. Thus, under this setting we are unable to apply backpropagation to do the learning.
(C) However, we can solve this problem by applying the reparametrization trick. That is, we move the sampling operation to an input layer. We can do that by reparametrizing $\mathcal{N}(\mu_Z, \Sigma_Z)$ as $\mathcal{N}(0, I) \odot \Sigma_Z + \mu_Z$. This means that we sample from a standard isotropic normal and then we scale and shift our sample to transform it to a sample from $\mathcal{N}(\mu_Z, \Sigma_Z)$. By doing this reparametrization trick, we are now able to obtain the gradient ∇_ϕ of the loss and learn both parameters ϕ and θ.

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

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.

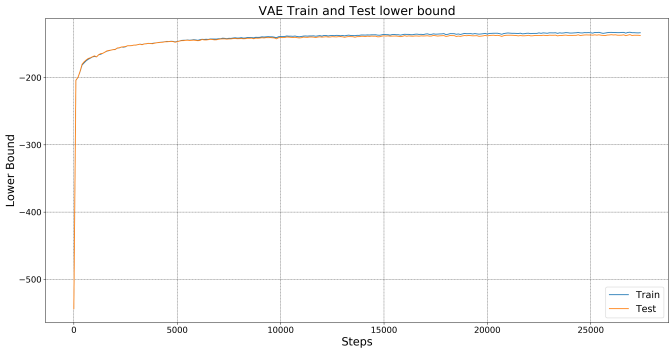


Figure 11: Lower bounds for entire training and test set during training.

Answer 22

After implementing the VAE in tensorflow, we can keep track of its performance by evaluating the mean lower bound both for training and test sets, every 100 training steps. In detail, we measure

$$\frac{1}{N}\sum_{n=1}^{N}\log p(X=x_n) - \mathcal{D}(q(z|X=x_n)\|p(z))$$

for the whole training and test sets and we plot the result for 20 epochs in Figure 11. We observe that the lower bounds increase rapidly in the first iterations and then the increase becomes slower. Furthermore, we can see that after 15k iterations the training lower bound becomes larger than the test lower bound. This is due to the fact that the model starts to specialize in the training examples and hence it achieves a larger lower bound than the test set. We note that this behavior was not observed in the Naive Bayes model, since it was much simpler and less powerful than the VAE.

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

A qualitative way to measure the model's performance is to plot digits sampled from the model as the training progresses. To sample from the model we apply the procedure provided in Problem 11. Figure 12 depicts samples from the model before the training. As can be seen the images are just noise. This is because the pixel space is so large (784), that it is impossible to construct a meaningful image by random sampling. After training our network for 5k training steps, we plot samples from the model as it is shown in Figure 13. We observe that many of the images depict well formed digits, however there are several cases where the digits are not clear. The reason why the model is able now to generate digit-like images, is because it learned the latent representation of the digits. Quantitatively, it can be seen in Figure 11 where the lower bound at 5k training steps is around -160, which is much larger than the initial -544. Finally, we plot samples from our model after training is finished (27k training steps) as can be seen in Figure 14. In this figure we can see that all of the images are well formed and resemble handwritten digits. Simple digits such as '1' and '0' appear more clear than more complicated ones, like '8' or '2'. The reason for this improvement in the digit quality is that the lower bound after the end of the training is around -130.

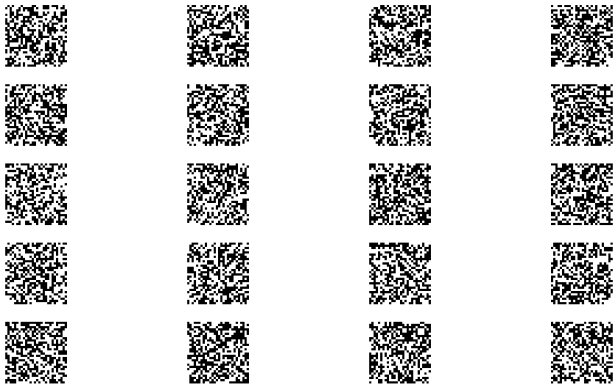


Figure 12: Samples from VAE before any training

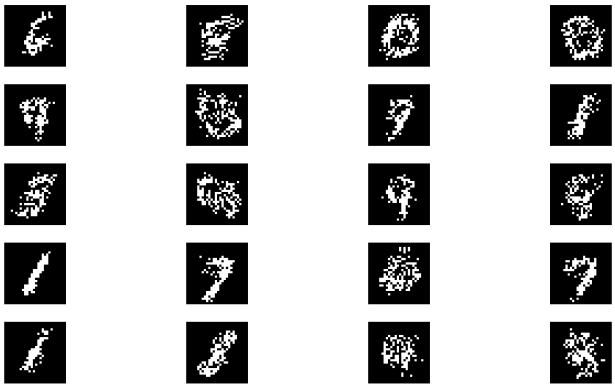


Figure 13: Samples from VAE after 5k training steps

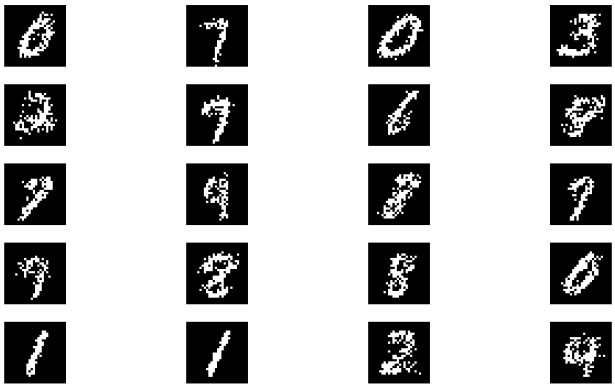


Figure 14: Samples from VAE after 27k training steps

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'

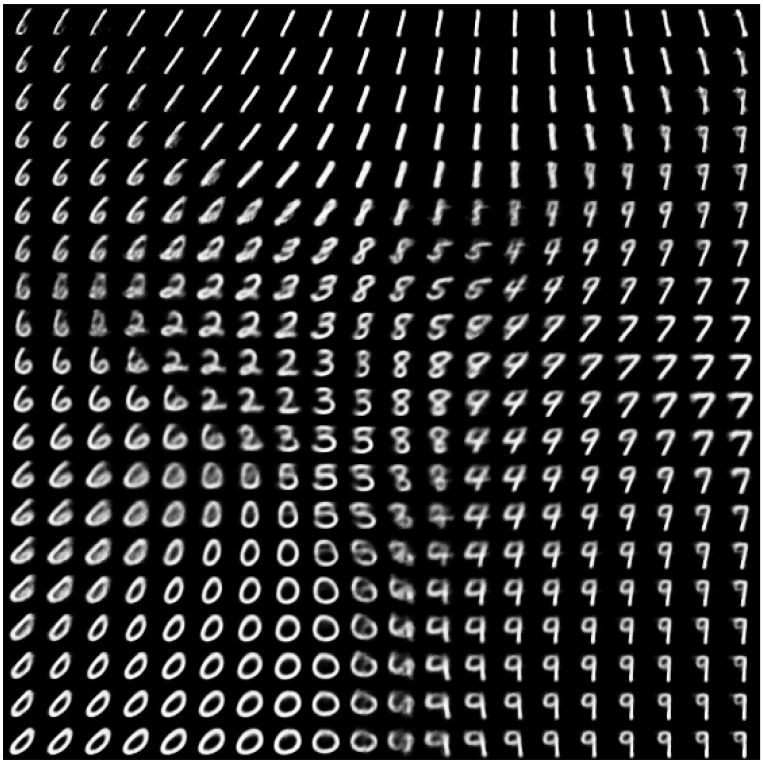


Figure 15: Learned MNIST manifold using the VAE

Answer 24

The final part of the analysis includes plotting the digit manifold learned by the VAE after complete training. The plotted MNIST manifold can be seen in Figure 15. The code which reproduces this plot can be found in the file `manifold.py`. In the figure we see the mean-probability of X given a 2-D grid of points in Z-space. We observe that the model is able to navigate through digits by moving along the two dimensions of z. From this figure we can also see that the model may still need further training, as several digits are not well formed.

Total Points: 204