MATH30022 Project (Semester 2)

Variational Auto-encoders with Application to Unsupervised Representation Learning

Author: Wenlin Chen (10407337)
Supervisor: Professor Korbinian Strimmer

Department of Mathematics, University of Manchester

May 15, 2020

Abstract

Auto-encoding variational Bayes provides a powerful and flexible framework for performing efficient inference in nonlinear latent variable models. In this thesis, we study the basics of auto-encoding variational Bayes and use the method to train variational auto-encoders. Such models allow us to not only compress and reconstruct data but also sample from the latent space to generate new data. For the method part, we obtain an evidence lower bound as the training objective and develop a practical estimator of it using the reparameterization trick, which makes the method scalable to large-scale problems. In addition, we carry out empirical studies on the MNIST, Fashion-MNIST and CIFAR-10 datasets, in order to examine the quality of the learned visual representations and to visualize their latent manifolds. It turns out that variational auto-encoders can learn good visual representations for simple vision tasks but struggle with complex ones.

Contents

Chapter 1

Introduction

1.1 Motivation

Supervised deep learning, which employs neural networks to learn a map from the input space to the output space, have achieved many successes over the past decade (e.g. [7, 15]). However, unsupervised learning, which corresponds to latent structure discovery, is also an important topic in statistical machine learning, especially for the tasks where labelling data is expensive or even impossible. In recent years, there have been many developments and advances in this area in terms of both modelling and inference.

Auto-encoders are important unsupervised models. An auto-encoder consists of an encoder and a decoder. Encoder maps from input layer to intermediate layer, and decoder maps from intermediate layer to output layer. Such a model discovers latent structures from data by reconstructing its input at the output layer so that the intermediate layer is forced to learn some lower dimensional latent representation of the data. This is one way of performing unsupervised representation learning, and the learned representations are useful for compression, visualization, clustering and many other different tasks.

Traditional auto-encoders employ deterministic functions (e.g. neural networks) as their encoders and decoders. In contrast, variational auto-encoders are unsupervised probabilistic latent variable models, where the term "probabilistic" means that everything has a probability distribution. In this way, variational auto-encoders allow us to not only reconstruct and compress data but also generate new data by sampling from the latent space. Learning in such models is an inference problem. However, it is impossible to perform exact Bayesian inference in such nonlinear latent variable models, as their posterior distributions are intractable. Therefore, we need to make approximations.

There are two classes of approximate inference methods - sampling-based approximate inference and deterministic approximate inference. Sampling-based methods such as Markov Chain Monte Carlo [24] are accurate but slow for large datasets [11]. Deterministic methods, such as Laplace approximation [21], variational inference [2] and expectation propagation [23], were invented to perform efficient Bayesian inference in complex models and for large datasets. In this thesis, we will develop auto-encoding variational Bayes method, a Bayesian inference algorithm that applies variational inference to learn probabilistic encoders and decoders, which is where the term "variational auto-encoder" comes from.

1.2 Overview of the Thesis

In this thesis, we will study auto-encoding variational Bayes and apply the method to train variational auto-encoders for learning unsupervised visual representations. We follow the expositions of the original paper [11] and the tutorial [13] for the method part. Readers are expected to be familiar with multivariate statistics and probabilistic machine learning. Some of our experiments are inspired by the GitHub repositories [18, 22, 30].

We begin Chapter 2 by introducing linear and nonlinear latent variable models - three typical examples of linear models are given and the technical difficulties in training nonlinear models are discussed. We also introduce unsupervised representation learning as an application of latent variable models.

In Chapter 3, we study the theory of Auto-encoding Variational Bayes. We develop the essential ingredients of variational inference and apply them to train auto-encoders. We introduce a key trick that enables mini-batch training of large-scale models on large datasets, which leads to the AEVB algorithm. We finish the chapter by giving a concrete example, where we describe the factorized Gaussian variational auto-encoder model and obtain a practical estimator for its objective function.

In Chapter 4, we study the empirical properties and performance of factorized Gaussian variational auto-encoders, training them on three standard machine learning image datasets - MNIST, Fashion-MNIST and CIFAR-10. The effect of latent space dimension is investigated. We also visualize the reconstructed images produced and latent manifolds learned by such variational auto-encoders, in order to gain an understanding of the quality of the visual representations that they can discover and learn.

In Chapter 5, we summarize the thesis and point out some relevant directions for future study.

Chapter 2

Latent Variable Models and Representation Learning

2.1 Linear Latent Variable Models

Unsupervised learning in latent variable models is an important problem in statistical machine learning. Common latent variable models such as Principal Component Analysis (PCA), Probabilistic PCA and Factor Analysis are linear (Gaussian state space) models and can be learned using the Expectation Maximization (EM) Algorithm [28].

The generative process of the observed variable $\boldsymbol{x}$ in such linear latent variable models can be described by the linear equation

$$\boldsymbol{x} = \boldsymbol{\Phi}\boldsymbol{z} + \boldsymbol{e},$$

where $\boldsymbol{\Phi} \in \mathbb{R}^{p\times d}(p \gg d)$ is the generative matrix, the latent variable

$$\boldsymbol{z} \sim \mathcal{N}_d(\boldsymbol{0}, \boldsymbol{S})$$

is assumed to have mean zero without loss of generality, and the noise $\boldsymbol{e}$ involved in the generative process is distributed as

$$\boldsymbol{e} \sim \mathcal{N}_d(\boldsymbol{0}, \boldsymbol{\Sigma}).$$

The conditional distribution and marginal distribution of the observed variable $\boldsymbol{x}$ are analytically tractable and given by

$$\boldsymbol{x}|\boldsymbol{z} \sim \mathcal{N}_p(\boldsymbol{\Phi}\boldsymbol{z}, \boldsymbol{\Sigma}) \quad \text{and} \quad \boldsymbol{x} \sim \mathcal{N}_p(\boldsymbol{0}, \boldsymbol{\Phi}\boldsymbol{S}\boldsymbol{\Phi}^T + \boldsymbol{\Sigma}),$$

respectively [28]. Observing that $\boldsymbol{S}$ only occurs in the term $\boldsymbol{\Phi}\boldsymbol{S}\boldsymbol{\Phi}^T$, we set $\boldsymbol{S}$ to be the identity matrix $\boldsymbol{I}$, since the structures of $\boldsymbol{S}$ can be absorbed into $\boldsymbol{\Phi}$. Then, $\boldsymbol{\Sigma}$ must be restricted, otherwise a trivial solution would be $\boldsymbol{\Sigma} = \boldsymbol{0}$ and $\frac{1}{n}\boldsymbol{\Phi}\boldsymbol{\Phi}^T$ being the sample covariance of the observed data points (i.e. $\boldsymbol{\Phi}$ being the observed data matrix). In fact, different restrictions on $\boldsymbol{\Sigma}$ give different models, as discussed in the paper [28]:

- **Factor analysis** restricts $\boldsymbol{\Sigma}$ to be a non-trivial diagonal matrix $\boldsymbol{\Lambda}$, which gives

$$\boldsymbol{x}|\boldsymbol{z} \sim \mathcal{N}_p(\boldsymbol{\Phi}\boldsymbol{z}, \boldsymbol{\Lambda}) \quad \text{and} \quad \boldsymbol{x} \sim \mathcal{N}_p(\boldsymbol{0}, \boldsymbol{\Phi}\boldsymbol{\Phi}^T + \boldsymbol{\Lambda}).$$

- **Probabilistic PCA** restricts $\boldsymbol{\Sigma}$ to be a scale matrix $\alpha \boldsymbol{I}$ ($\alpha > 0$), which gives

$$\boldsymbol{x}|\boldsymbol{z} \sim \mathcal{N}_p(\boldsymbol{\Phi}\boldsymbol{z}, \alpha \boldsymbol{I}) \quad \text{and} \quad \boldsymbol{x} \sim \mathcal{N}_p(\boldsymbol{0}, \boldsymbol{\Phi}\boldsymbol{\Phi}^T + \alpha \boldsymbol{I}).$$

- **Standard PCA** is a special case of Probabilistic PCA, which restricts

$$\boldsymbol{\Sigma} = \lim_{\alpha \to 0} \alpha \boldsymbol{I}.$$

  The directions of the columns of $\boldsymbol{\Phi}$ are known as the *principal components* in such a model.

For factor analysis and probabilistic PCA, we can learn the parameters $\boldsymbol{\Phi}$ and $\boldsymbol{\Sigma}$ using the usual EM algorithm, since the posterior distributions $p(\boldsymbol{z}|\boldsymbol{x})$ are analytically tractable Gaussian distributions. The EM algorithm starts from random initialization of the parameters. Then, it iteratively computes the posterior distribution $p(\boldsymbol{z}|\boldsymbol{x})$ using current parameters and updates the parameters by optimizing the expected complete data log-likelihood with respect to the newly-computed posterior distribution until convergence.

For the standard PCA model, the posterior distribution becomes a single point as $\alpha \to 0$ (i.e. as the noise $\boldsymbol{e}$ vanishes):

$$p(\boldsymbol{z}|\boldsymbol{x}) = \delta(\boldsymbol{z} - (\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T\boldsymbol{x}),$$

which implies that the inference in such a model becomes least square projections [28]. Having said that, there is still an EM algorithm for learning the generative matrix $\boldsymbol{\Phi}$ in standard PCA, as described in the paper [27]. An alternative method is to compute the eigenvectors of a (full rank) sample covariance matrix and rank them by their corresponding eigenvalues. Then, by definition, the first d-eigenvetors are the principal components. Although the alternative method only needs two steps – forming and diagonalizing the sample covariance matrix, eigenvalue decomposition in the second step is computationally expensive and data-hungry for high dimensional data. Finally, we note that the standard PCA model does not define a proper distribution for the observed variable $\boldsymbol{x}$, and so it makes no sense to ask for the likelihood of an observed data point [28].

2.2 Nonlinear Latent Variable Models

In this thesis, we will instead work with nonlinear latent variable models which have a continuous latent random variable $\boldsymbol{z}$ and a continuous or discrete observed random variable $\boldsymbol{x}$. Nonlinear latent variable models have the ability to learn more effective latent representations $\boldsymbol{z}$, as the expressive power of nonlinear functions is much stronger than that of linear functions in general. The (observed) dataset $\mathcal{D} = \{\boldsymbol{x}_n\}_{n=1}^N$ that we consider in this thesis consists of independent and identically distributed samples of the observed random variable $\boldsymbol{x}$. The generating process of $\mathcal{D}$ in such latent variable models can be described as follows:

1. sample $\boldsymbol{z}_n$ from some prior distribution $p_{\boldsymbol{\theta}^*}(\boldsymbol{z})$;
2. sample $\boldsymbol{x}_n$ from some conditional distribution $p_{\boldsymbol{\theta}^*}(\boldsymbol{x}|\boldsymbol{z}_n)$.

The dimension of the latent variable space is assumed to be much lower than that of the observed variable space, noting that each latent value $\boldsymbol{z}_n$ can be regarded as a summary

or a compression of the information contained in $\boldsymbol{x}_n$. However, the latent values $\{\boldsymbol{z}_n\}_{n=1}^N$ are not observable to us, as its name suggests.

The true parameters $\boldsymbol{\theta}^*$ are also unknown to us. Our modelling approach is to parameterize the prior $p_{\boldsymbol{\theta}^*}(\boldsymbol{z})$ and the likelihood $p_{\boldsymbol{\theta}^*}(\boldsymbol{x}|\boldsymbol{z})$ with some parametric families of distributions $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ and $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$, respectively, of which the probability distribution functions are assumed to be almost everywhere differentiable with respect to $\boldsymbol{\theta}$ and $\boldsymbol{z}$.

Having specified the model and collected the observed data, one could then learn the parameters $\boldsymbol{\theta}$ and the latent values $\boldsymbol{z}_n$ using the EM algorithm, variational inference method [2], or sampling-based inference methods (e.g. Markov chain Monte Carlo [24]). However, there are several difficulties in learning in nonlinear latent variable models, as discussed in the paper [11]:

1. When the likelihood function $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ is complicated, the marginal distribution

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int_{\mathcal{Z}} p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{z}) d\boldsymbol{z}$$

   and the posterior distribution

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) = \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{x})}$$

   would be intractable. This means that the EM algorithm and the mean-field variational inference method would no longer be applicable.

2. When a large dataset $\mathcal{D}$ is available, sampling-based inference methods would be very slow in general. This is because such methods often require an expensive sampling loop running over the whole dataset for each update of parameters.

Auto-encoding Variational Bayes, abbreviated as AEVB, provides a powerful and flexible framework for learning in general latent variable models. The method was first published in the paper [11]. As we will show in this thesis, the method has many good properties and is well suited to address the problems mentioned above:

1. It provides an efficient way to approximate the maximum likelihood estimation (or even the distribution) of the parameters $\boldsymbol{\theta}$, which allows us to mimic the random process of data generation.

2. It allows us to efficiently approximate the posterior distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$, which is useful for dimensionality reduction and data representation/compression.

3. It enables us to efficiently approximate the marginal distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x})$.

4. Stochastic (mini-batch) gradient-based optimization is applicable, which makes the method scalable to very large datasets.

(These are discussed in the paper [11].)

2.3 Unsupervised Representation Learning

Unsupervised representation learning has been gaining more and more attention in the machine learning community in recent years. It is a class of learning methods that automatically discover useful representations from unlabelled raw data. Such representations can then be used for many future tasks, such as feature selection, dimensionality reduction, data compression, visualization, clustering, classification and so on.

Contrastive learning and reconstructive learning are two popular ways of performing unsupervised representation learning. Contrastive learning methods (e.g. [3, 29]) learn only an encoder by maximizing agreements and minimizing disagreements between its output representations. If we let $\boldsymbol{f_\theta}$ be an encoder and $(\boldsymbol{x}, \boldsymbol{x}_+, \boldsymbol{x}_-)$ be a tuple where $\boldsymbol{x}_+$ is an augmentation of $\boldsymbol{x}$ and $\boldsymbol{x}_-$ is a data point in the dataset other than $\boldsymbol{x}$, then the training objective of a contrastive learning model has the form

$$\max_{\boldsymbol{\theta}} \sum_{(\boldsymbol{x},\boldsymbol{x}_+,\boldsymbol{x}_-)\in\mathcal{D}} \left[\operatorname{Sim}(\boldsymbol{f_\theta}(\boldsymbol{x}), \boldsymbol{f_\theta}(\boldsymbol{x}_+)) - \operatorname{Sim}(\boldsymbol{f_\theta}(\boldsymbol{x}), \boldsymbol{f_\theta}(\boldsymbol{x}_-)) \right],$$

where the similarity measure is usually vector inner product in practice. The paper [32] reveals the connections between traditional mutual information maximization methods and contrastive learning methods for unsupervised representation learning.

In contrast, reconstructive learning methods such as auto-encoders learn not only an encoder but also a decoder that together minimize the reconstruction error. In this thesis, we focus on the reconstructive learning method achieved by variational auto-encoders, of which more details will be discussed in Chapter 3.

Chapter 3

Auto-encoding Variational Bayes

3.1 Auto-encoders and Variational Auto-encoders

A typical auto-encoder model consists of an encoder and a decoder (see Figure 3.1). The encoder produces a latent representation $\boldsymbol{z}_n$ for a given data point $\boldsymbol{x}_n$, whereas the decoder generates (or reconstructs) a data point $\hat{\boldsymbol{x}}_n$ from a given latent vector $\boldsymbol{z}_n$. The training objective of such a model is to minimize the reconstruction errors $Err(\boldsymbol{x}_n, \hat{\boldsymbol{x}}_n)$ for all n. For a standard auto-encoder model, the encoder and decoder can be linear functions, which recovers standard PCA, or nonlinear functions, for example, neural networks.

Unlike standard auto-encoder models, *variational auto-encoders* (VAEs) instead use probabilistic encoders and decoders, which are conditional probability distributions. The probabilistic decoder is the likelihood function $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$, which is a distribution over all possible values of $\boldsymbol{x}$ for a given latent vector $\boldsymbol{z}$. The probabilistic encoder $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ with variational parameters $\boldsymbol{\phi}$ is defined to be a variational approximation of the intractable posterior distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$, which is a distribution over all possible values of $\boldsymbol{z}$ for a given data point $\boldsymbol{x}$. As we will see in Section 3.4, the variational parameters $\boldsymbol{\phi}$ and the generative parameters $\boldsymbol{\theta}$ in a variational auto-encoder are learned jointly using the AEVB algorithm.

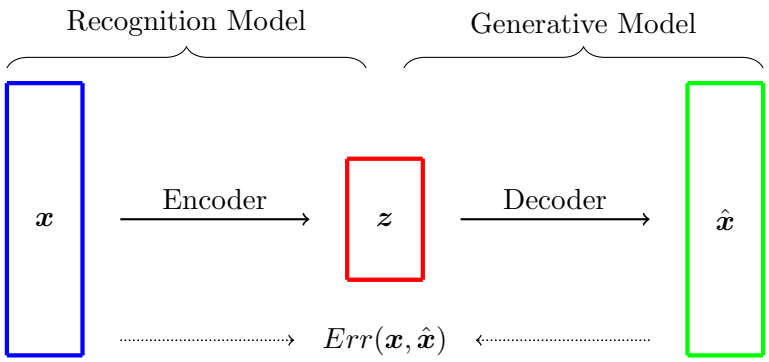


Figure 3.1: Encoder and decoder in an auto-encoder model.

3.2 Variational Inference as an Auto-encoder

As we saw in Section 3.1, the core idea of variational auto-encoders is to treat variational inference as an auto-encoder, where the likelihood $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ is treated as a probabilistic decoder, and the intractable posterior $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ is approximated by the variational distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, which is treated as a probabilistic encoder. We also point out that this is called amortized variational inference [5], as the variational parameters $\boldsymbol{\phi}$ are assumed to be shared across all data points, which leverages the efficiency of stochastic gradient-based optimization techniques at training time and allows efficient inference at test time [13, 31].

Instead of directly maximizing the marginal likelihood of the observed data

$$\log p_{\boldsymbol{\theta}}(\mathcal{D}) = \sum_{\boldsymbol{x}_n \in \mathcal{D}} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_n),$$

the optimization objective of variational inference is to maximize the so-called *evidence lower bound* (or *free energy*, *variational lower bound*), abbreviated as ELBO.

We first derive the ELBO for a single data point $\boldsymbol{x}$. Let $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ be a variational distribution with its variational parameters $\boldsymbol{\phi}$ of any choice. Then, we have

$$\begin{aligned}
\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x})\right] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}) p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \left(\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}\right)\right] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] + \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&\equiv \mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) + \mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) || p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})),
\end{aligned}$$

where the first term in the penultimate line is defined to be the ELBO

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right],$$

and the second term in the penultimate line is the Kullback-Leibler divergence of the true posterior distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ from the variational distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, which is non-negative since

$$\begin{aligned}
\mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) || p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})) &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[-\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&\geq -\log\left(\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\frac{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right]\right) \\
&= -\log\left(\int_{\mathcal{Z}} q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \frac{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} d\boldsymbol{z}\right) \\
&= -\log\left(\int_{\mathcal{Z}} p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) d\boldsymbol{z}\right) \\
&= -\log(1) \\
&= 0,
\end{aligned}$$

where the inequality follows by Jensen's inequality, since $-\log$ is a convex function, and is attained when the two distributions are identical.

Therefore, the ELBO is a lower bound of the observed data log-likelihood:

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \log p_{\boldsymbol{\theta}}(\boldsymbol{x}) - \mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})) \leq \log p_{\boldsymbol{\theta}}(\boldsymbol{x}),$$

where the term $\mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$ measures the tightness of the ELBO as a lower bound of the observed data log-likelihood [13].

We note that, by maximizing the ELBO $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x})$ with respect to both parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$, two things are simultaneously optimized, as discussed on page 19 of [13]:

1. The observed data log-likelihood $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ are approximately maximized, which gives a better generative model, hence a better decoder. However, unlike the EM algorithm, the observed data likelihood $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ is not guaranteed to be non-decreasing after every update of parameters in this case.
2. $\mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$ is minimized, which gives a better encoder $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$.

Furthermore, we rewrite the ELBO in an alternative form:

$$\begin{aligned}
\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})p_{\boldsymbol{\theta}}(\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&= -\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z})}\right] + \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right] \\
&= -\mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z})) + \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right], \qquad (3.1)
\end{aligned}$$

which will be useful for estimating the ELBO and its gradients in Section 3.4. It is worth noting that (3.1) also reveals a connection between variational auto-encoders and standard auto-encoders, as the second term can be viewed as the expected negative reconstruction error and the first term a regularizer [11].

3.3 Estimating the Gradients of the ELBO

The ELBO for a dataset $\mathcal{D}$ is defined as

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{D}) = \frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}_n \in \mathcal{D}} \mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}_n).$$

Our strategy is to jointly learn the variational parameters $\boldsymbol{\phi}$ and the generative parameters $\boldsymbol{\theta}$ by maximizing the ELBO using stochastic gradient-based optimization techniques (e.g. stochastic gradient ascent). For example, for a mini-batch $\mathcal{M} \subseteq \mathcal{D}$, one can update the parameters iteratively using stochastic gradient ascent, which gives

$$\boldsymbol{\theta}^{(new)} \leftarrow \boldsymbol{\theta} + \alpha_1 \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{M})$$

and

$$\boldsymbol{\phi}^{(new)} \leftarrow \boldsymbol{\phi} + \alpha_2 \nabla_{\boldsymbol{\phi}} \mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{M}),$$

where the hyper-parameters α_1 and α_2 are learning rates. We note that such mini-batch optimization techniques are scalable to very large datasets.

However, the ELBO

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right]$$

and its gradients $\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x})$ and $\nabla_{\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x})$ are intractable in general. Hence, we need to estimate them using random samples. Ideally, we would want the estimators to be unbiased with low variance.

We first obtain an unbiased estimator of the gradient of the ELBO with respect to the generative parameters $\boldsymbol{\theta}$. This is simple and straightforward, since we can interchange the gradient operator and the expectation operator:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) &= \nabla_{\boldsymbol{\theta}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\nabla_{\boldsymbol{\theta}}\left(\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right)\right] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})\right].
\end{aligned}$$

Therefore, an unbiased Monte Carlo estimator of this is given by

$$\hat{\nabla}_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \frac{1}{L}\sum_{l=1}^{L}\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z}^{(l)}),$$

where $\boldsymbol{z}^{(l)} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ are independent and identically distributed samples of the variational distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$.

However, such an argument cannot be applied to obtain an estimator of the gradient of the ELBO with respect to the variational parameters $\boldsymbol{\phi}$ in general, since

$$\begin{aligned}
\nabla_{\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) &= \nabla_{\boldsymbol{\phi}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right] \\
&\neq \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\nabla_{\boldsymbol{\phi}}\left(\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right)\right].
\end{aligned}$$

This is because the expectation is taken with respect to the variational distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, which is a function of the variational parameters $\boldsymbol{\phi}$. If we let

$$f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x}) = \log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}),$$

then one might have

$$\begin{aligned}
\nabla_{\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) &= \nabla_{\boldsymbol{\phi}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x})\right] \\
&= \nabla_{\boldsymbol{\phi}}\int_{\mathcal{Z}} f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x})q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})d\boldsymbol{z} \\
&= \int_{\mathcal{Z}} \nabla_{\boldsymbol{\phi}}\left[f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x})q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right]d\boldsymbol{z} \\
&= \int_{\mathcal{Z}} q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\nabla_{\boldsymbol{\phi}}f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x})d\boldsymbol{z} + \int_{\mathcal{Z}} f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x})\nabla_{\boldsymbol{\phi}}q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})d\boldsymbol{z} \\
&= \int_{\mathcal{Z}} q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\left[-\nabla_{\boldsymbol{\phi}}\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right]d\boldsymbol{z} + \int_{\mathcal{Z}} f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x})q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\nabla_{\boldsymbol{\phi}}\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})d\boldsymbol{z} \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\left(f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x}) - 1\right)\nabla_{\boldsymbol{\phi}}\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\left(\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) - 1\right)\nabla_{\boldsymbol{\phi}}\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right],
\end{aligned}$$

of which an unbiased Monte Carlo estimator would be

$$\hat{\nabla}_{\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \frac{1}{L}\sum_{l=1}^{L}\Big(\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z}^{(l)}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}^{(l)}|\boldsymbol{x}) - 1\Big)\,\nabla_{\boldsymbol{\phi}}\log q_{\boldsymbol{\phi}}(\boldsymbol{z}^{(l)}|\boldsymbol{x}),$$

where $\boldsymbol{z}^{(l)} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ are independent and identically distributed samples of the variational distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$. However, the variance of this naive Monte Carlo gradient estimator can be very large [25], which is impractical for our purpose. In Section 3.4, we will take a different approach to estimate the gradients of the ELBO.

3.4 Reparameterizing the ELBO

In this section, we introduce the so-called *reparameterization trick* [11], which is a simple but powerful technique that allows us to efficiently obtain good estimators of the ELBO and its gradients with respect to both parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ through a change of variable.

Let $\boldsymbol{\varepsilon}$ be a random noise with density $p(\boldsymbol{\varepsilon})$ that is independent of $\boldsymbol{\theta}$, $\boldsymbol{\phi}$ and $\boldsymbol{x}$. Given $\boldsymbol{\phi}$ and $\boldsymbol{x}$, we consider the random variable $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ as being transformed from the random noise $\boldsymbol{\varepsilon}$ via an invertible and differentiable transformation:

$$\boldsymbol{z} = \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon},\boldsymbol{x}).$$

For the case where the distribution of $\boldsymbol{z}$ is in a location-scale family, one can let the random noise $\boldsymbol{\varepsilon}$ be distributed as the standard distribution in that family (with location $\mathbf{0}$ and scale $\boldsymbol{I}$) and use the coloring transformation

$$\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon},\boldsymbol{x}) = \boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}) + \boldsymbol{\Sigma}_{\boldsymbol{\phi}}(\boldsymbol{x})^{1/2}\boldsymbol{\varepsilon},$$

where $\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x})$ and $\boldsymbol{\Sigma}_{\boldsymbol{\phi}}(\boldsymbol{x})$ are location and scale parameters, respectively, computed from $\boldsymbol{x}$ by some function parameterized by $\boldsymbol{\phi}$. Another typical choice is to let $\boldsymbol{\varepsilon} \sim \mathcal{U}(\mathbf{0},\boldsymbol{I})$ and use a tractable inverse cumulative distribution function as the transformation function [11].

For a given invertible and differentiable transformation $\boldsymbol{g}_{\boldsymbol{\phi}}$, using the identity

$$d\boldsymbol{z} = \left|\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\varepsilon}}\right| d\boldsymbol{\varepsilon} = \left|\det J_{\boldsymbol{\varepsilon}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{\varepsilon}))\right| d\boldsymbol{\varepsilon},$$

we can reparameterize the ELBO as

$$\begin{aligned}
\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x})\right] \\
&= \int_{\mathcal{Z}} f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x}) q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) d\boldsymbol{z} \\
&= \int_{\mathcal{E}} f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon},\boldsymbol{x}),\boldsymbol{x}) q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon},\boldsymbol{x})|\boldsymbol{x}) \left|\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\varepsilon}}\right| d\boldsymbol{\varepsilon} \\
&= \int_{\mathcal{E}} f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon},\boldsymbol{x}),\boldsymbol{x}) p(\boldsymbol{\varepsilon}) d\boldsymbol{\varepsilon} \\
&= \mathbb{E}_{p(\boldsymbol{\varepsilon})}\left[f_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon},\boldsymbol{x}),\boldsymbol{x})\right] \\
&= \mathbb{E}_{p(\boldsymbol{\varepsilon})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon},\boldsymbol{x})) - \log q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon},\boldsymbol{x})|\boldsymbol{x})\right],
\end{aligned}$$

where the scalar

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\varepsilon}} = \det J_{\boldsymbol{\varepsilon}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{\varepsilon}))$$

is the Jacobian determinant of the transformation $\boldsymbol{g}_{\boldsymbol{\phi}}$, and the fourth equality follows by the multivariate density transformation formula (see Section 3.6 for more details).

Therefore, a simple Monte Carlo estimator of the reparameterized ELBO is given by

$$\hat{\mathcal{L}}^{\text{SGVB}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \frac{1}{L}\sum_{l=1}^{L}\left(\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}^{(l)}, \boldsymbol{x})) - \log q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}^{(l)}, \boldsymbol{x})|\boldsymbol{x})\right), \tag{3.2}$$

where $\boldsymbol{\varepsilon}^{(l)} \sim p(\boldsymbol{\varepsilon})$ are independent and identically distributed samples of the noise density $p(\boldsymbol{\varepsilon})$. This estimator is called the *Stochastic Gradient Variational Bayes* estimator, abbreviated as SGVB estimator.

The reason for doing this change of variable is that it allows us to interchange the expectation operator and the gradient operator when differentiating the ELBO:

$$\begin{aligned}\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) &= \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathbb{E}_{p(\boldsymbol{\varepsilon})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x})) - \log q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x})|\boldsymbol{x})\right] \\ &= \mathbb{E}_{p(\boldsymbol{\varepsilon})}\left[\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x})) - \log q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x})|\boldsymbol{x})\right)\right],\end{aligned}$$

since the expectation is now taken with respect to the density $p(\boldsymbol{\varepsilon})$ that is independent of both parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$. Using this property, we can show that the gradients $\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\hat{\mathcal{L}}^{\text{SGVB}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x})$ of the SGVB estimator are unbiased:

$$\begin{aligned}\mathbb{E}_{p(\boldsymbol{\varepsilon})}[\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\hat{\mathcal{L}}^{\text{SGVB}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x})] &= \mathbb{E}_{p(\boldsymbol{\varepsilon})}\left[\frac{1}{L}\sum_{l=1}^{L}\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}^{(l)}, \boldsymbol{x})) - \log q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}^{(l)}, \boldsymbol{x})|\boldsymbol{x})\right)\right] \\ &= \frac{1}{L}\sum_{l=1}^{L}\mathbb{E}_{p(\boldsymbol{\varepsilon})}\left[\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}^{(l)}, \boldsymbol{x})) - \log q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}^{(l)}, \boldsymbol{x})|\boldsymbol{x})\right)\right] \\ &= \frac{1}{L}\sum_{l=1}^{L}\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathbb{E}_{p(\boldsymbol{\varepsilon})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}^{(l)}, \boldsymbol{x})) - \log q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}^{(l)}, \boldsymbol{x})|\boldsymbol{x})\right] \\ &= \frac{1}{L}\sum_{l=1}^{L}\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathbb{E}_{p(\boldsymbol{\varepsilon})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x})) - \log q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x})|\boldsymbol{x})\right] \\ &= \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathbb{E}_{p(\boldsymbol{\varepsilon})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x})) - \log q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x})|\boldsymbol{x})\right] \\ &= \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}).\end{aligned}$$

Recall that, in Section 3.2, we obtained an alternative form (3.1) of the ELBO:

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = -\mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z})) + \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right].$$

This is useful when the first term $\mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z}))$ is analytically tractable, in which case only the second term $\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]$ needs to be estimated using random samples. Applying the reparameterization trick to (3.1), we can obtain another form of the SGVB estimator:

$$\hat{\mathcal{L}}^{\text{SGVB}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = -\mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z})) + \frac{1}{L}\sum_{l=1}^{L}\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}^{(l)}, \boldsymbol{x})), \tag{3.3}$$

where $\boldsymbol{\varepsilon}^{(l)} \sim p(\boldsymbol{\varepsilon})$ are independent and identically distributed samples of the noise density $p(\boldsymbol{\varepsilon})$. In Section 3.5, we will see an example that makes use of this form of the SGVB

estimator as its optimization objective.

For a mini-batch $\mathcal{M} \subseteq \mathcal{D}$, we can estimate the ELBO for the whole dataset by

$$\hat{\mathcal{L}}^{\text{SGVB}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{D}) \approx \hat{\mathcal{L}}^{\text{SGVB}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{M}) = \frac{1}{|\mathcal{M}|} \sum_{\boldsymbol{x}_m \in \mathcal{M}} \hat{\mathcal{L}}^{\text{SGVB}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}_m), \tag{3.4}$$

where either (3.2) or (3.3) can be used as the SGVB estimator $\hat{\mathcal{L}}^{\text{SGVB}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x})$ for a single data point.

Algorithm 1 summarizes the procedure for training a variational auto-encoder via stochastic gradient-based optimization using the SGVB estimator.

Algorithm 1: The *Auto-encoding Variational Bayes* (AEVB) algorithm.

Input : $\mathcal{D} = \{\boldsymbol{x}_n\}_{n=1}^N$: observed data points
$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$: probabilistic encoder (the variational distribution)
$p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$: probabilistic decoder (the generative distribution)
L : number of random samples for each data point
$|\mathcal{M}|$: mini-batch size
Opt : gradient-based (ascent) optimizer

Output: $\boldsymbol{\phi}$: variational parameters
$\boldsymbol{\theta}$: generative parameters

Randomly initialize the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$
while *Opt not converged* **do**
  Sample a mini-batch $\mathcal{M} \subseteq \mathcal{D}$ with size $|\mathcal{M}|$
  Sample L random noises $\boldsymbol{\varepsilon}_m^{(l)} \sim p(\boldsymbol{\varepsilon})$ for each data point $\boldsymbol{x}_m$ in $\mathcal{M}$
  Estimate the ELBO according to (3.4) using $\mathcal{M}$ and $\boldsymbol{\varepsilon}_m^{(l)}$
  Compute the gradients $\boldsymbol{g}$ of the estimated ELBO with respect to $\boldsymbol{\phi}, \boldsymbol{\theta}$
  Apply the optimizer *Opt* to update the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ using $\boldsymbol{g}$
end

3.5 Example: Factorized Gaussian Variational Auto-encoder

In this section, we study the factorized Gaussian variational auto-encoder (FGVAE), which is an example that makes use of the AEVB algorithm described in Section 3.4. Figure 3.2 is a diagram that shows how a Factorized Gaussian variational auto-encoder works. One thing to note is that the model employs neural networks for learning the parameters of the probabilistic encoder $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ and the probabilistic decoder $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$.

Assume that $\boldsymbol{z}$ is a d-dimensional latent variable. For the generative model (decoder), the prior distribution over the latent variable is defined by a multivariate standard Gaussian distribution:

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}) = \mathcal{N}_d(\boldsymbol{z}|\boldsymbol{0}, \boldsymbol{I}).$$

The likelihood $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ can be defined to be a multivariate Gaussian distribution (for continuous data), a categorical distribution (for categorical data), or a negative binomial distribution (for count data) [19], of which the distribution parameters are computed from $\boldsymbol{z}$ using the decoding neural network parameterized by $\boldsymbol{\theta}$ as usual.

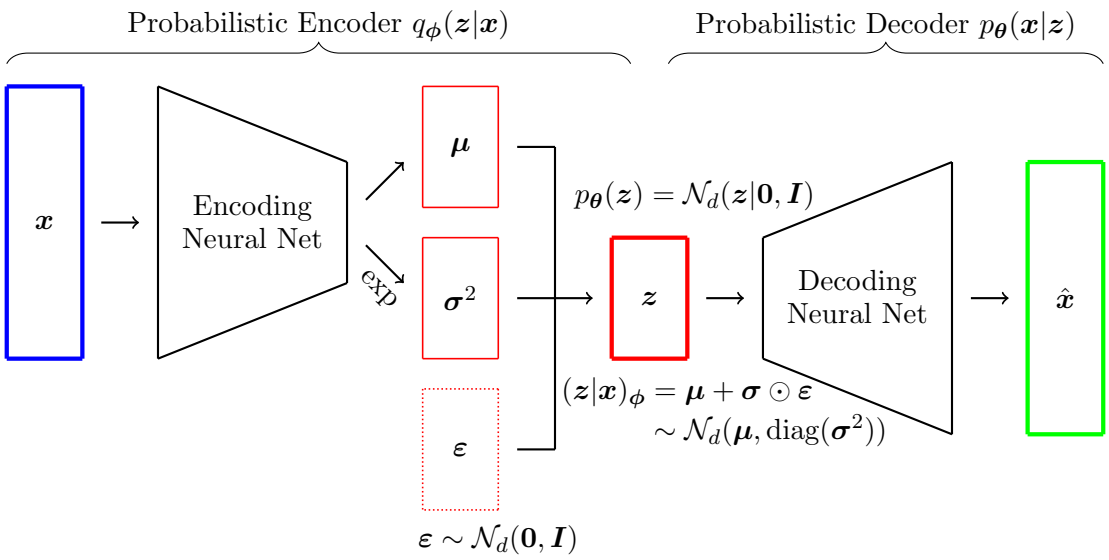


Figure 3.2: Diagram of a factorized Gaussian variational auto-encoder.

Clearly, the true posterior distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ is intractable. Therefore, we use a variational distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ to approximate it. The factorized Gaussian variational auto-encoder assumes that the true posterior distribution approximately takes the form of a multivariate Gaussian distribution with diagonal covariance matrix:

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}_d(\boldsymbol{z}|\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}), \text{diag}(\boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x})^2)) = \prod_{j=1}^{d} \mathcal{N}(z_j|\mu_{\boldsymbol{\phi},j}(\boldsymbol{x}), \sigma_{\boldsymbol{\phi},j}(\boldsymbol{x})^2),$$

where $\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}) = (\mu_{\boldsymbol{\phi},1}(\boldsymbol{x}), \cdots, \mu_{\boldsymbol{\phi},d}(\boldsymbol{x}))^T$ and $\log \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x})^2 = (\log \sigma_{\boldsymbol{\phi},1}(\boldsymbol{x})^2, \cdots, \log \sigma_{\boldsymbol{\phi},d}(\boldsymbol{x})^2)^T$ are the outputs of the encoding neural network parameterized by $\boldsymbol{\phi}$ with input data point $\boldsymbol{x}$. For example, a fully-connected encoding neural network with one nonlinear hidden layer would be

$$\boldsymbol{a} = \boldsymbol{h}(\boldsymbol{W}_{\boldsymbol{\phi}}^{[1]}\boldsymbol{x} + \boldsymbol{b}_{\boldsymbol{\phi}}^{[1]}),$$

$$\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}) = \boldsymbol{W}_{\boldsymbol{\phi}}^{[2]}\boldsymbol{a} + \boldsymbol{b}_{\boldsymbol{\phi}}^{[2]} \quad \text{and} \quad \log \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x})^2 = \boldsymbol{W}_{\boldsymbol{\phi}}^{[3]}\boldsymbol{a} + \boldsymbol{b}_{\boldsymbol{\phi}}^{[3]},$$

where $\boldsymbol{\phi} = \{\boldsymbol{W}_{\boldsymbol{\phi}}^{[1]}, \boldsymbol{W}_{\boldsymbol{\phi}}^{[2]}, \boldsymbol{W}_{\boldsymbol{\phi}}^{[3]}, \boldsymbol{b}_{\boldsymbol{\phi}}^{[1]}, \boldsymbol{b}_{\boldsymbol{\phi}}^{[2]}, \boldsymbol{b}_{\boldsymbol{\phi}}^{[3]}\}$ ($\boldsymbol{W}_{\boldsymbol{\phi}}^{[i]}$ are matrices and $\boldsymbol{b}_{\boldsymbol{\phi}}^{[i]}$ are vectors) and $\boldsymbol{h}$ is a nonlinear (element-wise operating) activation function (e.g. sigmoid, tanh or ReLu). One can also replace fully-connected neural networks with convolutional neural networks [17], especially for image dataset.

Since both $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ and $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ are Gaussian distributed, of which the Kullback-Leibler divergence is analytically tractable, we use (3.3) as the optimization objective. The paper [4] gives a general formula for calculating the Kullback-Leibler divergence between two multivariate Gaussian distributions:

$$\begin{aligned}
&\mathbb{D}_{KL}(\mathcal{N}_d(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)||\mathcal{N}_d(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)) \\
&= -\frac{1}{2}\left(d + \log\left(\frac{\det \boldsymbol{\Sigma}_1}{\det \boldsymbol{\Sigma}_2}\right) - (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) - \text{tr}(\boldsymbol{\Sigma}_2^{-1}\boldsymbol{\Sigma}_1)\right),
\end{aligned}$$

which, for the factorized Gaussian variational auto-encoder, becomes

$$\begin{aligned}\mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z})) &= \mathbb{D}_{KL}(\mathcal{N}_d(\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}), \operatorname{diag}(\boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x})^2))||\mathcal{N}_d(\mathbf{0}, \boldsymbol{I})) \\ &= -\frac{1}{2}\sum_{j=1}^{d}\left(1 + \log \sigma_{\boldsymbol{\phi},j}(\boldsymbol{x})^2 - \mu_{\boldsymbol{\phi},j}(\boldsymbol{x})^2 - \sigma_{\boldsymbol{\phi},j}(\boldsymbol{x})^2\right).\end{aligned}$$

For the second term in the objective, we choose the transformation $\boldsymbol{g}_{\boldsymbol{\phi}}$ to be

$$\tilde{\boldsymbol{z}}^{(l)} = \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{\varepsilon}^{(l)}) = \boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}) + \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x}) \odot \boldsymbol{\varepsilon}^{(l)},$$

where $\odot$ is the element-wise product operator, and the noise samples

$$\boldsymbol{\varepsilon}^{(l)} \sim p(\boldsymbol{\varepsilon}) = \mathcal{N}_d(\boldsymbol{\varepsilon}|\mathbf{0}, \boldsymbol{I})$$

are independent and identically distributed.

Putting them together, we obtain the SGVB estimator for the factorized Gaussian variational auto-encoder for a single data point $\boldsymbol{x}$:

$$\hat{\mathcal{L}}^{\text{FGVAE}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \frac{1}{2}\sum_{j=1}^{d}\left(1 + \log \sigma_{\boldsymbol{\phi},j}(\boldsymbol{x})^2 - \mu_{\boldsymbol{\phi},j}(\boldsymbol{x})^2 - \sigma_{\boldsymbol{\phi},j}(\boldsymbol{x})^2\right) + \frac{1}{L}\sum_{l=1}^{L}\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\tilde{\boldsymbol{z}}^{(l)}),$$

where

$$\tilde{\boldsymbol{z}}^{(l)} = \boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}) + \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x}) \odot \boldsymbol{\varepsilon}^{(l)} \quad \text{and} \quad \boldsymbol{\varepsilon}^{(l)} \overset{i.i.d.}{\sim} \mathcal{N}_d(\mathbf{0}, \boldsymbol{I}).$$

Then, we can estimate the ELBO for the whole dataset according to (3.4), compute its gradients, and update the parameters as described in Algorithm 1.

3.6 Computing the Variational Distribution

If we use (3.2) as the objective function for training variational auto-encoders, then we would need to compute the variational distribution

$$\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}), \quad \text{where } \boldsymbol{z} = \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x}) \text{ and } \boldsymbol{\varepsilon} \sim p(\boldsymbol{\varepsilon}).$$

Since the transformation $\boldsymbol{g}_{\boldsymbol{\phi}}$ is differentiable and invertible, by the general density transformation formula for multivariate distributions, we have

$$p(\boldsymbol{\varepsilon}) = \left|\det J_{\boldsymbol{\varepsilon}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{\varepsilon}))\right| q_{\boldsymbol{\phi}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x})|\boldsymbol{x}),$$

where the Jacobian matrix

$$J_{\boldsymbol{\varepsilon}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{\varepsilon})) \equiv \begin{pmatrix} \frac{\partial z_1}{\partial \varepsilon_1} & \cdots & \frac{\partial z_1}{\partial \varepsilon_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_d}{\partial \varepsilon_1} & \cdots & \frac{\partial z_d}{\partial \varepsilon_d} \end{pmatrix}$$

contains all the first order partial derivatives of $\boldsymbol{z} = \boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{\varepsilon}, \boldsymbol{x})$ with respect to $\boldsymbol{\varepsilon}$.

Hence, the formula for computing the variational distribution is given by

$$\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \log p(\boldsymbol{\varepsilon}) - \log\left|\det J_{\boldsymbol{\varepsilon}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{\varepsilon}))\right|.$$

For the factorized Gaussian variational auto-encoder considered in Section 3.5, we have

$$p(\boldsymbol{\varepsilon}) = \mathcal{N}_d(\boldsymbol{\varepsilon}|\mathbf{0}, \boldsymbol{I}) = \prod_{j=1}^{d} \mathcal{N}(\varepsilon_j|0,1) \quad \text{and} \quad \left|\det J_{\boldsymbol{\varepsilon}}(\boldsymbol{g}_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{\varepsilon}))\right| = \prod_{j=1}^{d} \sigma_{\boldsymbol{\phi},j}(\boldsymbol{x}).$$

Therefore, the variational distribution of the factorized Gaussian variational auto-encoder can be computed by

$$\begin{aligned} \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) &= \log \left(\prod_{j=1}^{d} \mathcal{N}(\varepsilon_j|0,1) \right) - \log \left(\prod_{j=1}^{d} \sigma_{\boldsymbol{\phi},j}(\boldsymbol{x}) \right) \\ &= \sum_{j=1}^{d} \left(\log \mathcal{N}(\varepsilon_j|0,1) - \log \sigma_{\boldsymbol{\phi},j}(\boldsymbol{x}) \right), \end{aligned}$$

where

$$\boldsymbol{z} = \boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}) + \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x}) \odot \boldsymbol{\varepsilon}.$$

Chapter 4

Empirical Evaluation

4.1 Experiment Settings

In this chapter, we present some empirical results of factorized Gaussian variational auto-encoders (FGVAE, described in Section 3.5), training them on three standard machine learning image datasets using the AEVB algorithm described in Section 3.4, in order to examine the empirical properties and performance and to gain an understanding of the capacity and limitation of such models. We visualize the reconstructed images produced and the latent manifolds learned by such variational auto-encoders for each dataset (where applicable) and analyze the results.

The programming language that we use for this empirical study is Python (version 3.7.3). We use PyTorch [26] (version 1.4.0), a CUDA-accelerated machine learning library that provides basic machine learning building blocks and supports automatic differentiation, to build and train our models, and Matplotlib [9] to produce graphs and plot images. Random seed is set for reproducibility. The computer code for all our experiments can be found in Appendix A.

In order to be compatible with PyTorch, we define the loss function of the FGVAE for a mini-batch $\mathcal{M}$ by equation (3.4) but with a negative sign:

$$l_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{M}) = -\frac{1}{|\mathcal{M}|}\sum_{\boldsymbol{x}\in\mathcal{M}}\left[\frac{1}{2}\sum_{j=1}^{d}\left(1+\log\sigma_{\boldsymbol{\phi},j}(\boldsymbol{x})^2-\mu_{\boldsymbol{\phi},j}(\boldsymbol{x})^2-\sigma_{\boldsymbol{\phi},j}(\boldsymbol{x})^2\right)+\frac{1}{L}\sum_{l=1}^{L}\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\tilde{\boldsymbol{z}}^{(l)})\right],$$

where

$$\tilde{\boldsymbol{z}}^{(l)} = \boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}) + \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x}) \odot \boldsymbol{\varepsilon}^{(l)} \quad \text{and} \quad \boldsymbol{\varepsilon}^{(l)} \overset{i.i.d.}{\sim} \mathcal{N}_d(\mathbf{0}, \boldsymbol{I}).$$

The optimization problem now becomes find parameters $\boldsymbol{\theta}, \boldsymbol{\phi}$ such that the loss function $l_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{D})$ is minimized. In our experiments, we set $L = 1$ and use Adam [12] optimizer with learning rate 10^{-3}, weight decay factor 10^{-5} and mini-batch size 128 to minimize the loss function. For both of the encoder and decoder, we use fully-connected neural networks, unless otherwise stated. Table 4.1 shows the overall architectures of them and also the size of each layer. All layers in the table except Output are followed by a batch normalization [10] layer. Activation functions are ReLU unless otherwise stated.

	Encoder	Decoder
Input size	784	d
FC1	512	128
FC2	256	256
FC3	128	512
Output FC	$d + d$ (linear)	784 (Sigmoid)

Table 4.1: Fully-connected neural network architectures for the encoder and decoder.

4.2 MNIST

The MNIST dataset [16] is one of the most commonly used image datasets for evaluating visual learning models, which consists of 60,000 training examples and 10,000 test examples of handwritten digit images ranging from 0 to 9. These are greyscale images of the size 28×28 pixels. We train our FGVAEs for 20 epochs on the training set and evaluate their empirical performance and properties using the test set.

We first explore the effect of the latent space dimension d on the quality of reconstructed images by training 6 FGVAEs with $d = 2, 5, 10, 15, 20, 25$, respectively. Figure 4.1 shows the test loss and (average) test reconstruction error for each of them, from which we can see that $d = 20$ is sufficient for the MNIST dataset in terms of both criteria.

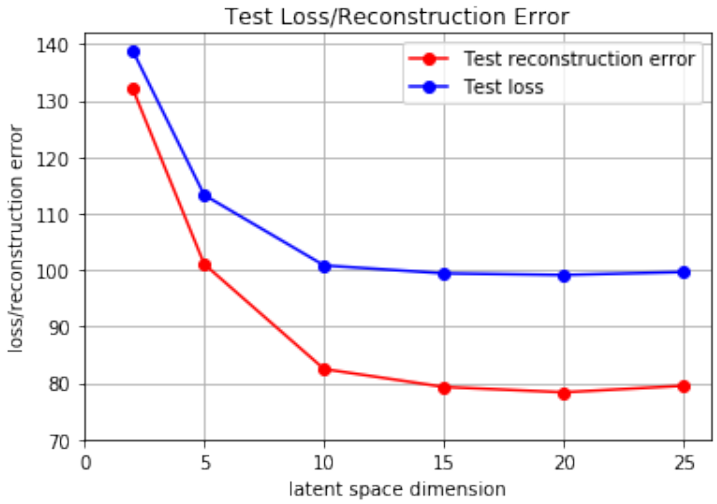


Figure 4.1: Effect of the latent space dimension d on the quality of reconstructed images for the MNIST dataset.

Then, we plot the reconstructed images from FGVAEs with different latent space dimensions d and compare them with the original images. Figure 4.2 shows the reconstructed images of a random sample of 30 images from the test set using the FGVAEs with $d = 2$ (column 2) and $d = 20$ (column 3) in comparison with the original images (column 1). We can see that the FGVAE with $d = 20$ get all of the 30 reconstructed images correct. However, the one with $d = 2$ sometimes mistakes the digits 9 for 4, 7 for 9 and 2 for 3.

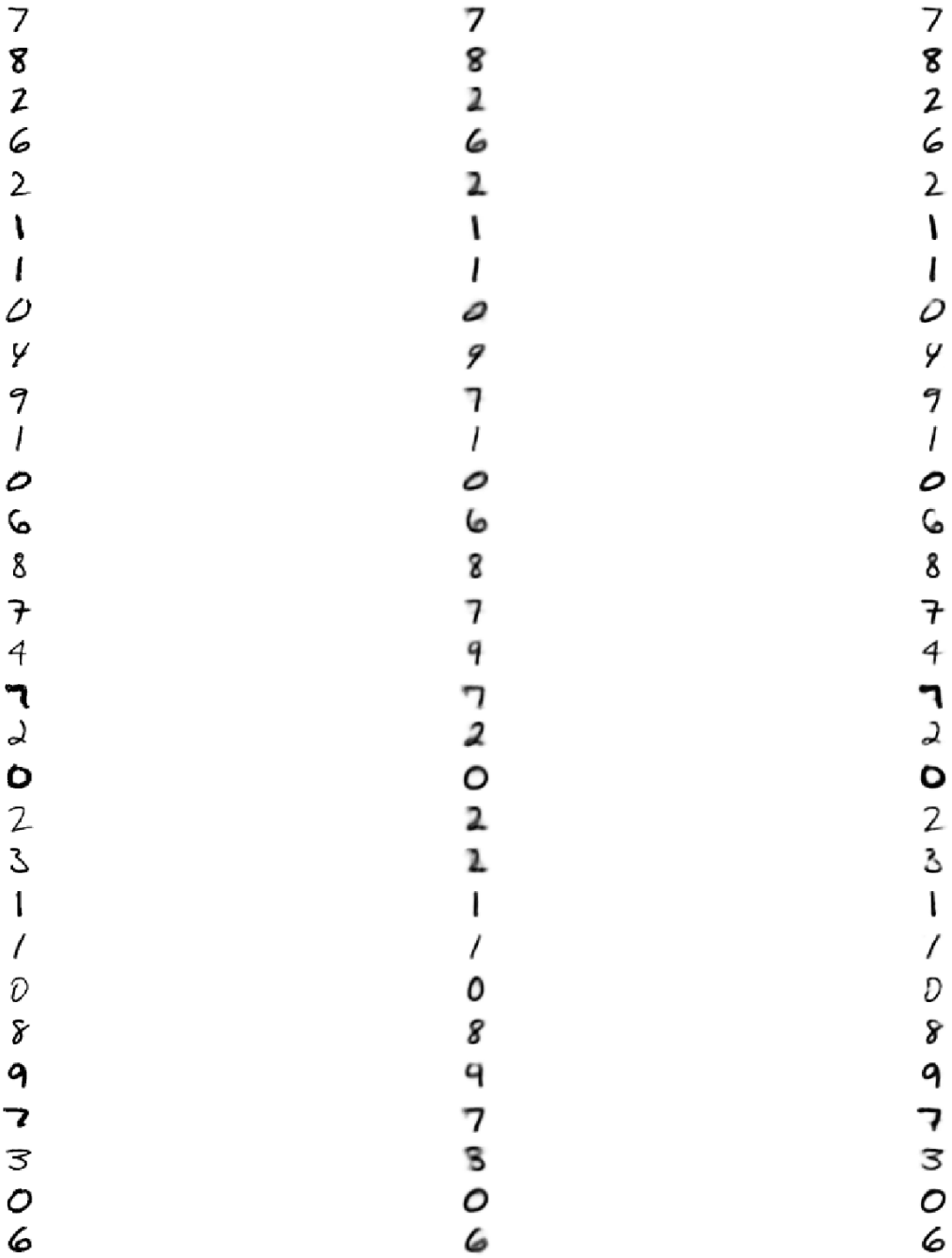
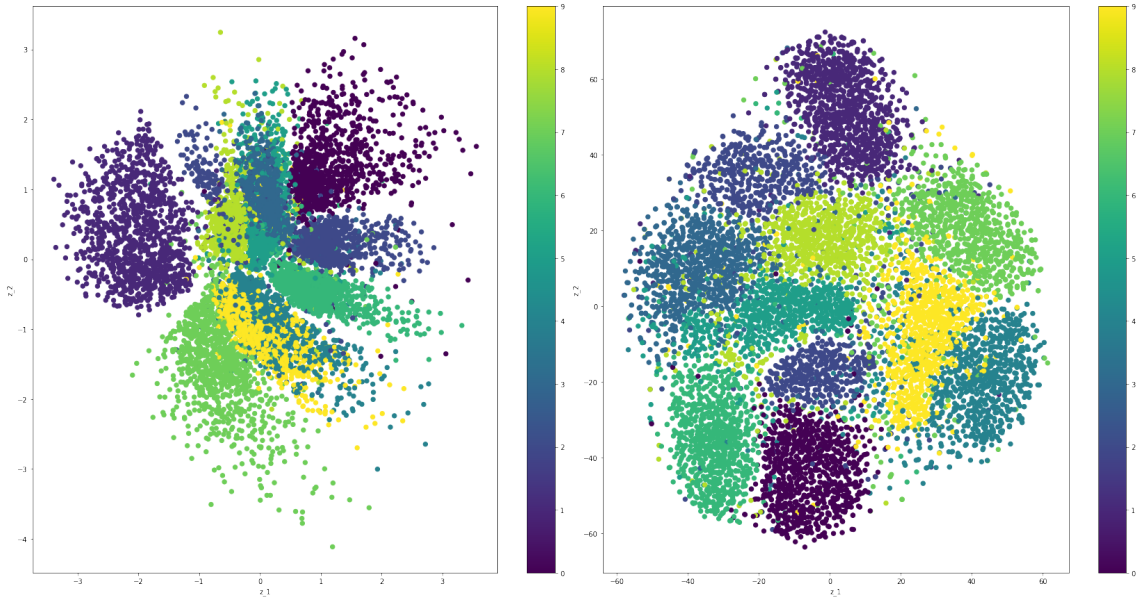


Figure 4.2: Reconstructed images using the FGVAEs with $d = 2$ (column 2) and $d = 20$ (column 3) in comparison with the original images (column 1) on the MNIST dataset.

To understand why the FGVAE with $d = 20$ reconstructs images well whereas the one with $d = 2$ fails for some cases, we need to examine the properties of the latent manifolds learned by them. There are two ways of visualizing a latent manifold - projecting images onto the manifold and sampling images from the manifold.



(a) $d = 2$. (b) $d = 20$ (visualized using t-SNE).

Figure 4.3: Projections of the test set onto manifolds of FGVAEs trained on MNIST.

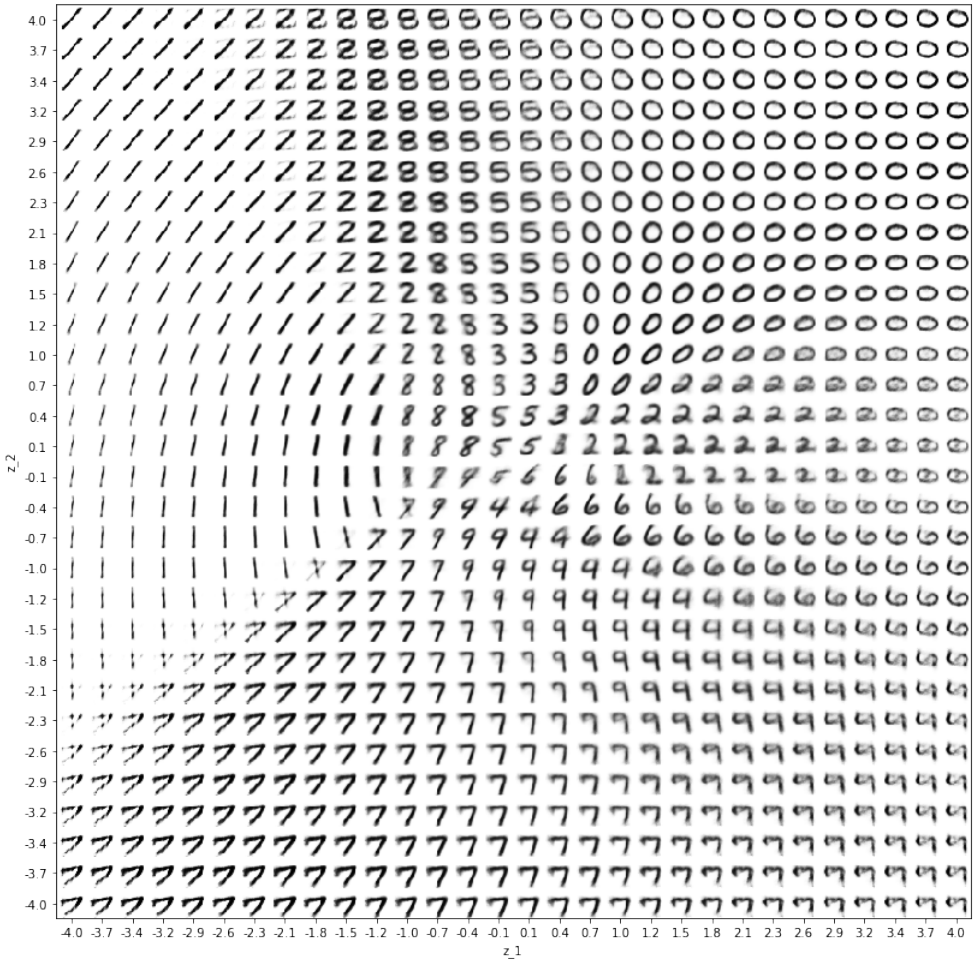


Figure 4.4: Equally spaced sampling from the manifold of the FGVAE with $d = 2$ trained on the MNIST dataset.

In figure 4.3, the points of the scatter plot are the projections of the test set data points onto the latent spaces of the FGVAEs with (a) $d = 2$ and (b) $d = 20$ (visualized using t-SNE [20]), of which different colors represent different digits. We can see clearly that there are much fewer overlaps between different clusters in (b) than in (a). Hence, we can conclude that the FGVAE with $d = 20$ has learned a good visual representation for this vision task, since the clusters for different classes of digits are separable from each other. If we trained a linear classifier using this representation, it would perform better than that trained on the original training set.

In figure 4.4, we generate these images using equally spaced sampling points (ranging from -4 to 4 in both dimensions) from the latent space of the FGVAE with $d = 2$, which gives us a better view of how the latent manifold of this model looks like (e.g. how digits change from one to another in the manifold).

4.3 Fashion-MNIST

The Fashion-MNIST dataset [33] is an MNIST-like fashion product image dataset for evaluating visual learning models, which consists of 60,000 training examples and 10,000 test examples of fashion product images, including 10 classes of different types of shoes, clothes, dress, trousers and bags. The format of the images in this dataset is exactly the same as those in the MNIST dataset. However, this dataset provides a more difficult visual learning task in comparison to the MNIST dataset, as the objects are richer and more complicated. For the Fashion-MNIST dataset, we carry out the same kinds of experiments that were done for the MNIST dataset with exactly the same architectures and configurations, in order to examine whether the FGVAE has the capacity to reconstruct images and learn a good visual representation for a slightly more complex vision task.

Figure 4.5 shows the test loss and (average) test reconstruction error for each of the 6 FGVAEs with $d = 2, 5, 10, 15, 20, 25$, from which we can see that $d = 20$ is also sufficient for the Fashion-MNIST dataset in terms of both criteria.

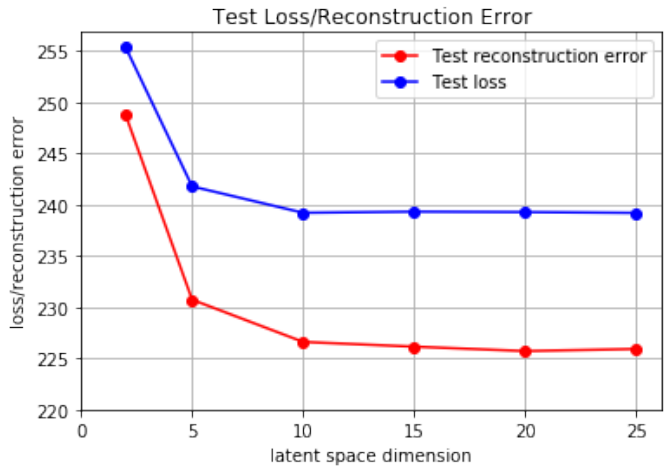


Figure 4.5: Effect of the latent space dimension d on the quality of reconstructed images for the Fashion-MNIST dataset.

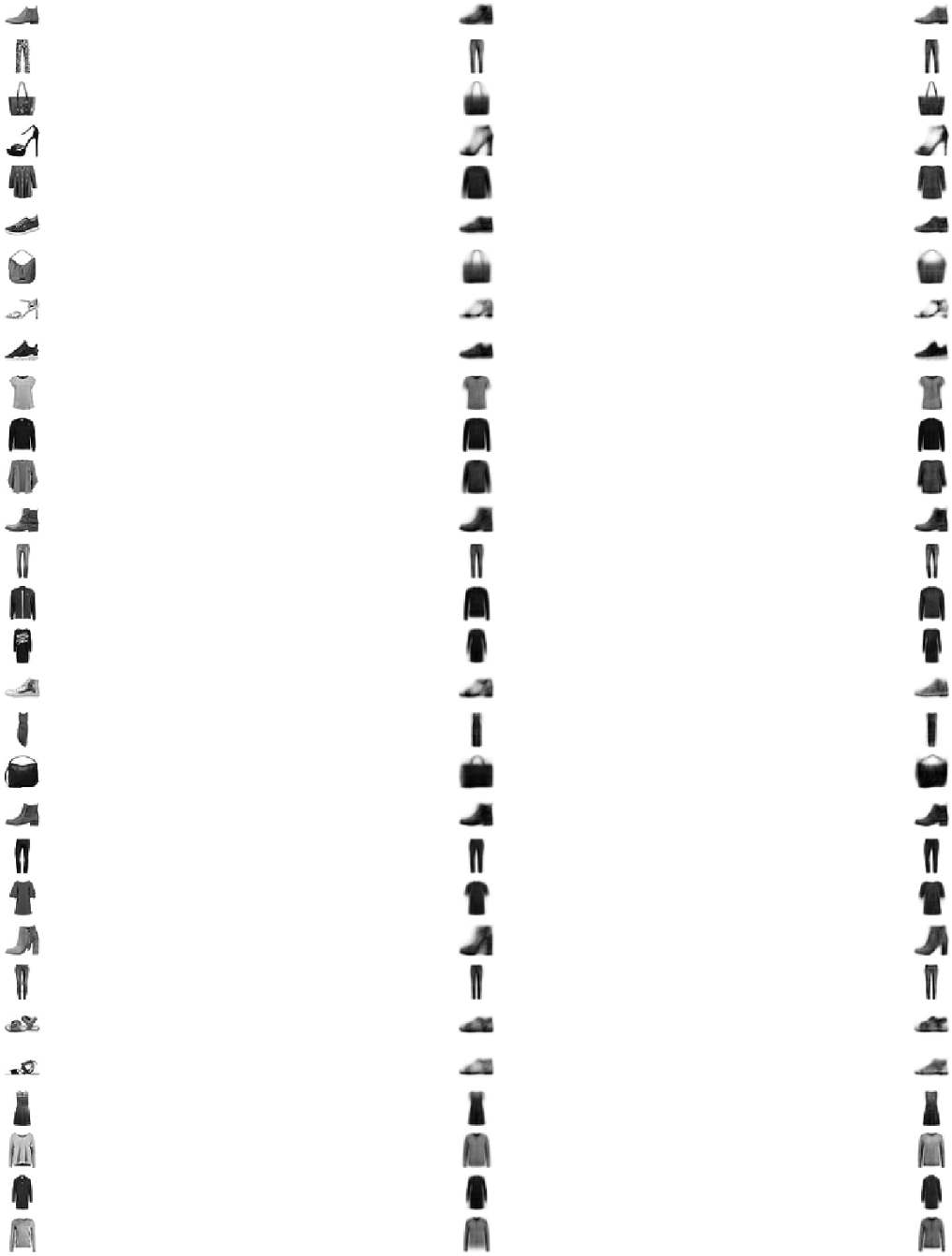
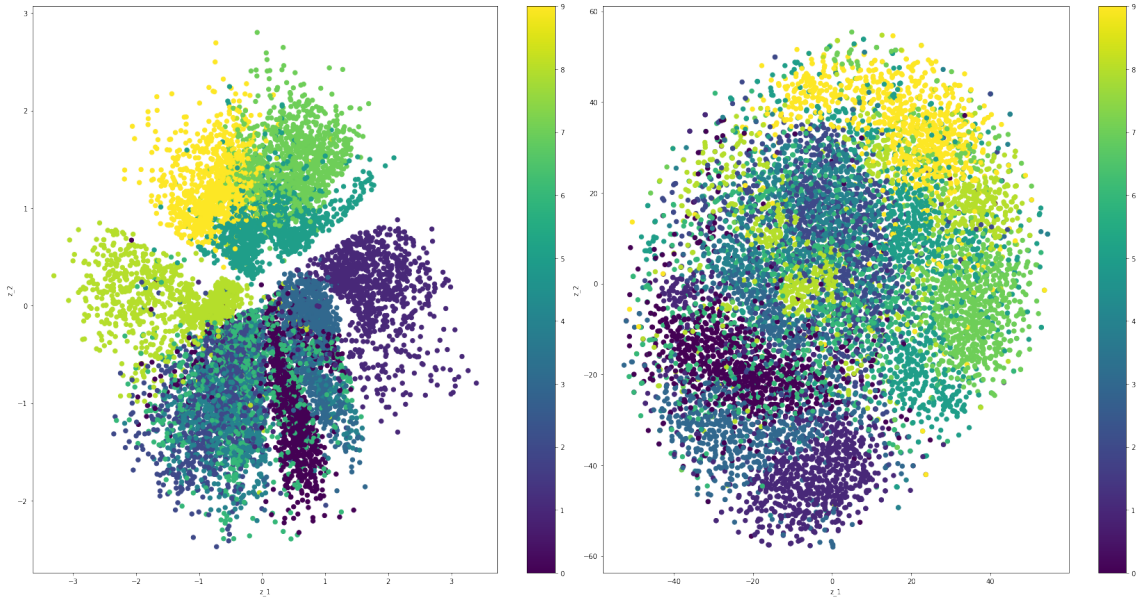


Figure 4.6: Reconstructed images using the FGVAEs with $d = 2$ (column 2) and $d = 20$ (column 3) in comparison with the original images (column 1) on Fashion-MNIST.

Figure 4.6 shows the reconstructed images of a random sample of 30 images from the test set using the FGVAEs with $d = 2$ (column 2) and $d = 20$ (column 3) in comparison with the original images (column 1). We can see that, although images in column 3 are slightly better than those in column 2, the reconstructed images from both models are blurry/noisy and lack many details, with some being incorrect. This is actually one of the drawbacks of variational auto-encoders, especially when applied to complex images. This drawback will become clearer in Section 4.4, where a real-world image dataset is used.



(a) $d = 2$. (b) $d = 20$ (visualized using t-SNE).

Figure 4.7: Projections of the test set onto manifolds of FGVAEs trained on the Fashion-MNIST dataset.

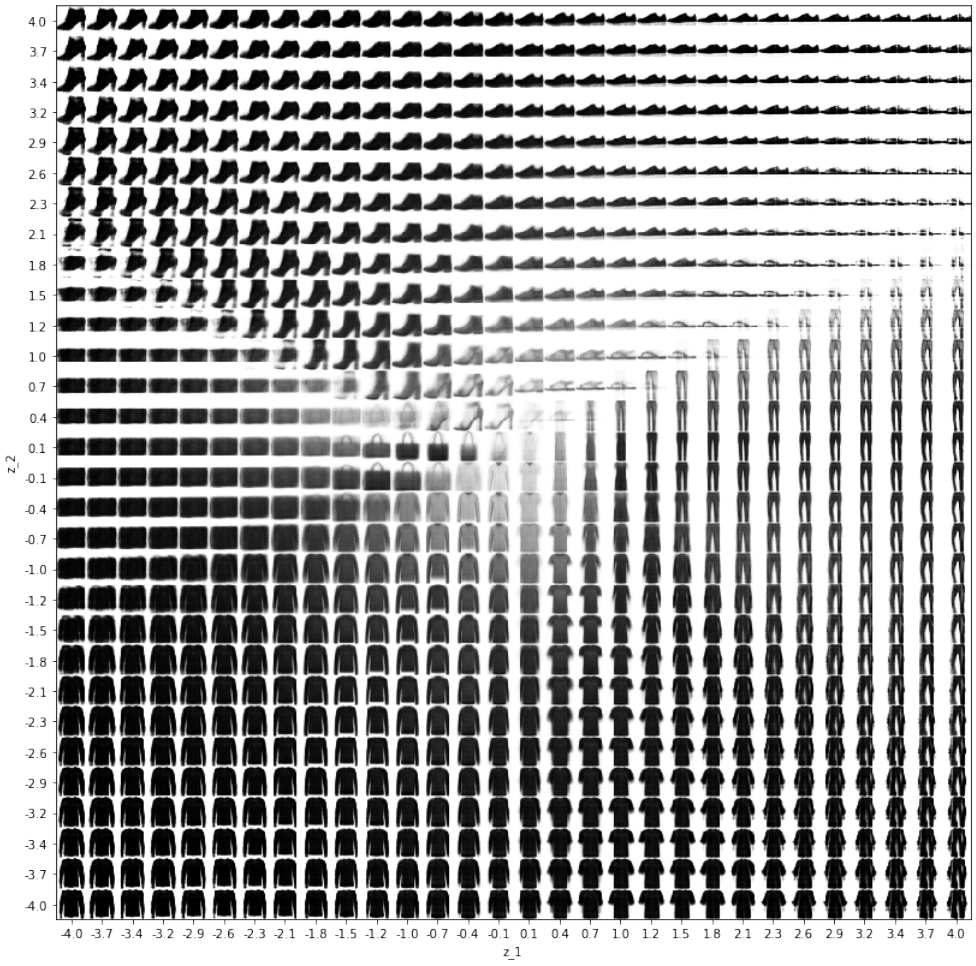


Figure 4.8: Equally spaced sampling from the manifold of the FGVAE with $d = 2$ trained on the Fashion-MNIST dataset.

The points of scatter plot in Figure 4.7 are the projections of the test set data points onto the latent spaces of the FGVAEs with (a) $d = 2$ and (b) $d = 20$ (visualized using t-SNE [20]), of which different colors represent different kinds of fashion products. We can see that there are many overlaps between different clusters in both (a) and (b). Therefore, we conclude that even the FGVAE with $d = 20$ fails to learn a good visual representation for this vision task, since the clusters for different kinds of fashion products are not separable from each other.

The images in Figure 4.8 are generated using equally spaced sampling points (ranging from -4 to 4 in both dimensions) from the latent space of the FGVAE with $d = 2$, which gives us a better view of how the latent manifold of this model looks like (e.g. how fashion products change from one to another in the manifold).

4.4 CIFAR-10

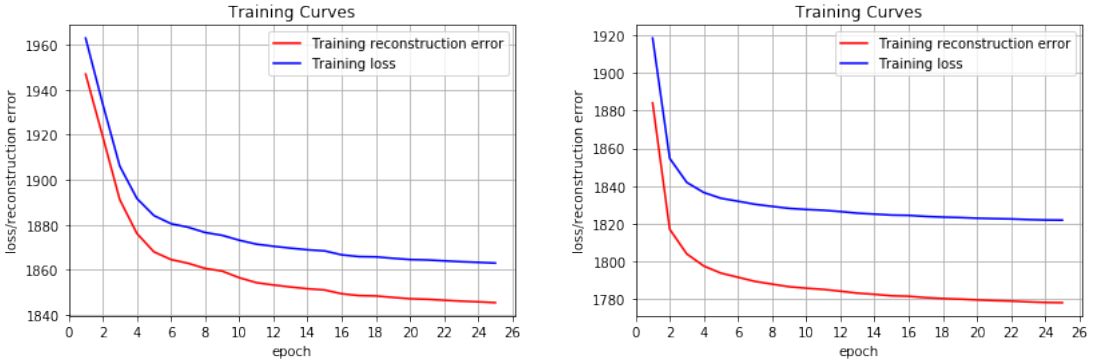
The CIFAR-10 dataset [14] is another popular image dataset for evaluating visual learning models, which consists of 50,000 training examples and 10,000 test examples of real world objects, including 10 classes of different kinds of animals and vehicles. These are color images of the size 32×32 pixels. This dataset has been widely used in deep learning literature, since it provides a reasonably difficult visual learning task from real world for quick evaluation. We train our FGVAEs for 25 epochs on the training set and evaluate their empirical performance and properties using the test set.

We train 2 FGVAEs (both with $d = 256$) - one uses fully-connected neural network architecture described in Table 4.1 in Section 4.1 for its encoder/decoder and the other uses convolutional neural network [17] architecture described in Table 4.2.

	Encoder	Decoder
Input size	$32 \times 32 \times 3$	d
Conv1/Input FC	#out channels = 32	2048 ($4 \times 4 \times 128$)
Conv2/Deconv1	#out channels = 64	#out channels = 64
Conv2/Deconv2	#out channels = 128	#out channels = 32
Output FC/Deconv3	$d + d$ (linear)	#out channels = 3 (Sigmoid)

Table 4.2: Convolutional neural network architectures for the encoder and decoder. All layers except the final one are followed by a batch normalization [10] layer. Activation functions are ReLU unless otherwise stated. For all convolutional/deconvolutional layers, we use padding = 1, stride = 1 and kernel size = 4×4.

Figure 4.9 shows the training curves for both FGVAEs and Table 4.3 shows the test results for them, from which we can see that even with convolutional neural networks and high dimensional latent space ($d = 256$), the FGVAE still have very high training and test losses and (average) reconstruction errors.



(a) FGVAE with fully-connected neural networks. (b) FGVAE with convolutional neural networks.

Figure 4.9: Training loss and reconstruction error against epoch of FGVAEs with $d = 256$ for different encoder/decoder architectures for the CIFAR-10 dataset.

	Test loss	Test reconstruction error
FC FGVAE	1864.54	1847.17
CNN FGVAE	1822.52	1779.06

Table 4.3: Test losses and reconstruction errors of the FGVAEs with $d = 256$ using fully-connected neural networks and convolutional neural networks trained on CIFAR-10.

Now, we plot the reconstructed images from both FGVAEs and compare them with the original images. Figure 4.10 shows the reconstructed images of a random sample of 10 images from the test set using FGVAEs with fully-connected neural networks (column 2) and convolutional neural networks (column 3) in comparison with the original images (column 1). We can see that, although the FGVAE with convolutional neural networks reconstructs more details than the one with fully-connected neural networks do, the reconstructed images are very blurry/noisy and not identifiable. Hence, it becomes clear that the quality of reconstructed images get worse and worse when we use more and more complex images.

Finally, we project the test set data points onto the latent spaces of the FGVAEs with (a) fully-connected neural networks and (b) convolutional neural networks, of which different colors of points represent objects in different classes. It is not surprising to see that, in each of the two scatter plots in Figure 4.11, almost all projection points are located in one cluster. This means that FGVAEs completely failed to learn a sensible visual representation for this real-world dataset of complex images.

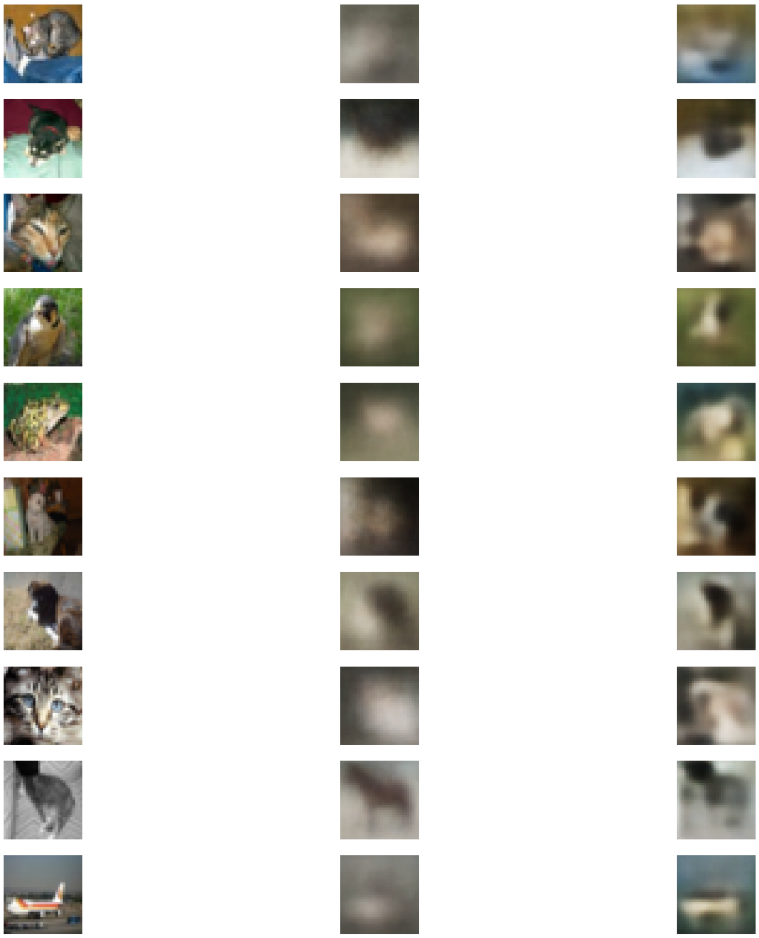
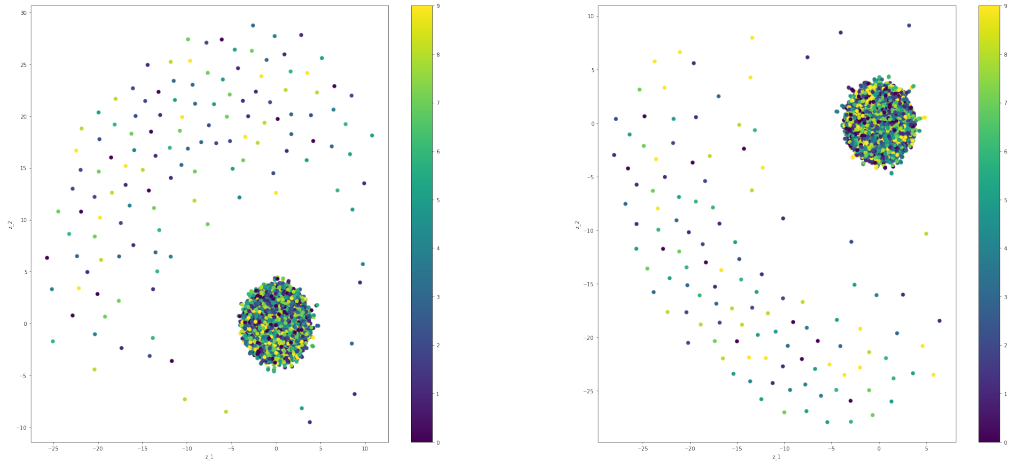


Figure 4.10: Reconstructed images using the FGVAEs with fully-connected neural networks (column 2) and convolutional neural networks (column 3) in comparison with the original images (column 1) on the CIFAR-10 dataset.



(a) FC FGVAE (visualized using t-SNE). (b) CNN FGVAE (visualized using t-SNE).

Figure 4.11: Projections of the test set onto manifolds of FGVAEs trained on CIFAR-10.

Chapter 5

Conclusion

In this thesis, we studied auto-encoding variational Bayes method, treating Bayesian approximate inference as an auto-encoder. We learned a probabilistic way of modelling auto-encoders as nonlinear latent variable models and implemented variational inference to train them efficiently. We obtained the ELBO of the observed data log-likelihood and developed a practical estimator of it using the reparameterization trick, which makes it scalable to large datasets with stochastic gradient-based optimization techniques. In comparison with traditional auto-encoders, variational auto-encoders allow us to not only compress and reconstruct data but also sample from the latent space to generate new data. An important application of auto-encoders is to perform unsupervised representation learning. As we saw from the empirical results, variational auto-encoders learned fairly sensible latent manifolds and good visual representations for simple image datasets, but struggled with complex ones. Overall, this is an interesting method but has some drawbacks, which introduces some exciting and active research topics for future study:

Disentangled representations. Disentangled representations are very useful for many tasks in machine learning, as they have been shown to be able to significantly improve both task performance and robustness compared to raw data [1, 8]. If time permits, we could study how to enforce variational auto-encoders to learn disentangled representations. For example, the paper [8] proposed β-VAE, which uses a hyper-parameter to constrain the term $\mathbb{D}_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z}))$ in the ELBO to be small so that the approximate posterior distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ is enforced to be close to the factorized Gaussian prior $p_{\boldsymbol{\theta}}(\boldsymbol{z})$.

Generative adversarial networks. As we saw from the empirical results, images generated from variational auto-encoders are blurry/noisy and sometimes even not identifiable, especially when using complex image datasets. For image generating tasks, Generative Adversarial Networks [6] may be a better choice. GAN consists of a discriminator D, which discriminates between real and fake images, and a generator G, which generates fake images to fool the discriminator, with D and G being trained alternately. If time permits, we could study GANs and compare their performance with VAEs.

Contrastive learning. As discussed in Section 2.3, contrastive learning is another popular way of learning unsupervised representations by learning distinctiveness in data, which has recently been turned out to be the state-of-the-art technique [3]. If time permits, we could study contrastive learning methods and compare their performance with VAEs.

Bibliography

[1] Y. Bengio, A. Courville, P. Vincent, *Representation learning: A review and new perspectives*, IEEE transactions on pattern analysis and machine intelligence 35, no. 8, pp. 1798-1828, 2013.

[2] D.M. Blei, A. Kucukelbir, J.D. McAuliffe, *Variational Inference: A Review for Statisticians*, Journal of the American statistical Association 112, no. 518, pp. 859-877, 2017.

[3] T. Chen, S. Kornblith, M. Norouzi, G. Hinton, *A simple framework for contrastive learning of visual representations*, arXiv:2002.05709, 2020.

[4] C. Doersch, *Tutorial on Variational Autoencoders*, arXiv:1606.05908, 2016.

[5] S. Gershman, N. Goodman, *Amortized Inference in Probabilistic Reasoning*, Proceedings of the annual meeting of the cognitive science society, Vol. 36, No. 36, 2014.

[6] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, *Generative adversarial nets*, Advances in neural information processing systems (NIPS), pp. 2672-2680, 2014.

[7] K. He, X. Zhang, S. Ren, J. Sun, *Deep residual learning for image recognition*, Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778, 2016.

[8] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, A. Lerchner, *beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework*, International Conference on Learning Representations (ICLR) 2, no. 5, pp. 6, 2017.

[9] J.D. Hunter, *Matplotlib: A 2D graphics environment*, Computing in science & engineering 9, no. 3, pp. 90-95, 2007.

[10] S. Ioffe, C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, arXiv:1502.03167, 2015.

[11] D.P. Kingma, M. Welling, *Auto-Encoding Variational Bayes*, arXiv:1312.6114, 2013.

[12] D.P. Kingma, J. Ba, *Adam: A method for stochastic optimization*, arXiv:1412.6980, 2014.

[13] D.P. Kingma, M. Welling, *An Introduction to Variational Autoencoders*, Foundations and Trends in Machine Learning 12, no. 4, pp. 307-392, 2019.

[14] A. Krizhevsky, G. Hinton, *Learning multiple layers of features from tiny images*, technical report, 2009.

[15] A. Krizhevsky, I. Sutskever, G.E. Hinton, *Imagenet classification with deep convolutional neural networks*, Advances in neural information processing systems (NIPS), pp. 1097-1105, 2012.

[16] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, 86(11):2278-2324, Nov. 1998.

[17] Y. LeCun, P. Haffner, L. Bottou, Y. Bengio, *Object Recognition with Gradient-based Learning*, Shape, contour and grouping in computer vision, pp. 319-345, Springer, Berlin, Heidelberg, 1999.

[18] A. Lee, *Variational AutoEncoder on the MNIST data set using the PyTorch*, GitHub repository, https://github.com/lyeoni/pytorch-mnist-VAE, 2018.

[19] R. Lopez, J. Regier, M.B. Cole, M.I. Jordan, N. Yosef, *Deep generative modeling for single-cell transcriptomics*, Nature methods 15, no. 12, pp. 1053-1058, 2018.

[20] L.V.D Maaten, G. Hinton, *Visualizing data using t-SNE*, Journal of machine learning research, 9. Nov, pp.2579-2605, 2008.

[21] D.J.C. MacKay, *A practical Bayesian framework for backpropagation networks*, Neural computation 4, no. 3, pp. 448-472, 1992.

[22] S. Malysheva, *Pytorch-VAE*, GitHub repository, https://github.com/SashaMalysheva/Pytorch-VAE, 2018.

[23] T.P. Minka, *Expectation propagation for approximate Bayesian inference*, arXiv:1301.2294, 2013.

[24] R. Paap, *What are the advantages of MCMC based inference in latent variable models*, Statistica Neerlandica 56, no. 1, pp.2-22, 2002

[25] J. Paisley, D. Blei, M. Jordan, *Variational Bayesian Inference with Stochastic Search*, arXiv:1206.6430, 2012.

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, *PyTorch: An imperative style, high-performance deep learning library*, Advances in neural information processing systems (NeurIPS), pp. 8024-8035, 2019.

[27] S.T. Roweis, *EM Algorithms for PCA and SPCA*, Advances in neural information processing systems (NIPS), pp. 626-632, 1998.

[28] S. Roweis, Z. Ghahramani, *A Unifying Review of Linear Gaussian Models*, Neural computation 11, no. 2, pp 305-345, 1999.

[29] N. Saunshi, O. Plevrakis, S. Arora, M. Khodak, H. Khandeparkar, *A Theoretical Analysis of Contrastive Unsupervised Representation Learning*, International Conference on Machine Learning (ICML), pp. 5628-5637, 2019.

[30] P. Singh, *Variational Autoencoder model in Keras*, GitHub Repository, https://github.com/piyush-kgp/VAE-MNIST-Keras, 2018.

[31] Y.W. Teh, *Advanced Topics in Statistical Machine Learning*, lecture notes, https://github.com/ywteh/advml2020, 2020.

[32] M. Tschannen, J. Djolonga, P.K. Rubenstein, S. Gelly, M. Lucic, *On mutual information maximization for representation learning*, arXiv:1907.13625, 2019.

[33] H. Xiao, K. Rasul, R. Vollgraf, *Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms*, arXiv:1708.07747, 2017.

Appendices

Appendix A

Computer Code

We attach here all computer code used for defining encoder/decoder architectures and for training variational auto-encoders on the three image datasets - MNIST, Fashion-MNIST and CIFAR-10. Some of our code and experiments are inspired by and adapted from the GitHub repositories [18, 22, 30].

A.1 Encoder/Decoder Architectures

```
import torch
import torch.nn as nn
import torch.distributions as tdist


def reparam(z_mean, z_log_var, device):

    std_normal_dist = tdist.Normal(0.0, 1.0)
    e = std_normal_dist.sample(z_mean.size()).to(device)
    z = z_mean + torch.exp(0.5 * z_log_var) * e
    return z


def fc_layer(in_dim, out_dim, final=False):

    if final:
        return nn.Linear(in_dim, out_dim)
    else:
        return nn.Sequential(nn.Linear(in_dim, out_dim),
                             nn.BatchNorm1d(out_dim),
                             nn.ReLU())


def conv_layer(c_in, c_out, kernel_size, stride=2, padding=1, final=False):
```

```
25
26      if final:
27          return nn.Conv2d(c_in, c_out, kernel_size, stride, padding)
28      else:
29          return nn.Sequential(nn.Conv2d(c_in, c_out, kernel_size, stride,
        padding),
30                               nn.BatchNorm2d(c_out),
31                               nn.ReLU())
32
33
34  def deconv_layer(c_in, c_out, kernel_size, stride=2, padding=1, final=False
        ):
35
36      if final:
37          return nn.ConvTranspose2d(c_in, c_out, kernel_size, stride, padding
        )
38      else:
39          return nn.Sequential(nn.ConvTranspose2d(c_in, c_out, kernel_size,
        stride, padding),
40                               nn.BatchNorm2d(c_out),
41                               nn.ReLU())
42
43
44  class VAE_MLP(nn.Module):
45      def __init__(self, input_dim, latent_dim, device):
46          super(VAE_MLP, self).__init__()
47
48          self.x_size = None
49          self.device = device
50
51          # Encoder
52          self.en_1 = fc_layer(input_dim, 512)
53          self.en_2 = fc_layer(512, 256)
54          self.en_3 = fc_layer(256, 128)
55          self.mean_fc = fc_layer(128, latent_dim, final=True)
56          self.log_var_fc = fc_layer(128, latent_dim, final=True)
57
58          # Decoder
59          self.de_1 = fc_layer(latent_dim, 128)
60          self.de_2 = fc_layer(128, 256)
61          self.de_3 = fc_layer(256, 512)
62          self.reconst_fc = fc_layer(512, input_dim, final=True)
63
64      def encoder(self, x):
65          self.x_size = x.size()
```

```
66          x = x.view(-1, x.size()[1]*x.size()[2]*x.size()[3])
67          x = self.en_1(x)
68          x = self.en_2(x)
69          x = self.en_3(x)
70          z_mean = self.mean_fc(x)
71          z_log_var = self.log_var_fc(x)
72          z = reparam(z_mean, z_log_var, self.device) # random sample
73
74          return z, z_mean, z_log_var
75
76      def decoder(self, z):
77          z = self.de_1(z)
78          z = self.de_2(z)
79          z = self.de_3(z)
80          x_reconst = torch.sigmoid(self.reconst_fc(z)).view(self.x_size)
81
82          return x_reconst
83
84      def forward(self, x):
85          z, z_mean, z_log_var = self.encoder(x)
86          x_reconst = self.decoder(z)
87
88          return z, z_mean, z_log_var, x_reconst
89
90
91  class VAE_CNN(nn.Module):
92      def __init__(self, input_dim, latent_dim, device, n_channels=3):
93          super(VAE_CNN, self).__init__()
94
95          self.x_size = None
96          self.device = device
97          self.n_channels = n_channels
98          self.f_dim = int((input_dim//n_channels)**(1/2)) // 8
99
100         # Encoder
101         self.en_1 = conv_layer(self.n_channels, 32, 4)
102         self.en_2 = conv_layer(32, 64, 4)
103         self.en_3 = conv_layer(64, 128, 4)
104         self.mean_fc = fc_layer(self.f_dim*self.f_dim*128, latent_dim,
    final=True)
105         self.log_var_fc = fc_layer(self.f_dim*self.f_dim*128, latent_dim,
    final=True)
106
107         # Decoder
108         self.de_fc = fc_layer(latent_dim, self.f_dim*self.f_dim*128)
```

```
109         self.de_1 = deconv_layer(128, 64, 4)
110         self.de_2 = deconv_layer(64, 32, 4)
111         self.de_3 = deconv_layer(32, self.n_channels, 4, final=True)
112 
113     def encoder(self, x):
114         self.x_size = x.size()
115         x = self.en_1(x)
116         x = self.en_2(x)
117         x = self.en_3(x)
118         x = x.view(-1, self.f_dim*self.f_dim*128)
119         z_mean = self.mean_fc(x)
120         z_log_var = self.log_var_fc(x)
121         z = reparam(z_mean, z_log_var, self.device) # random sample
122 
123         return z, z_mean, z_log_var
124 
125     def decoder(self, z):
126         z = self.de_fc(z)
127         z = z.view(-1, 128, self.f_dim, self.f_dim)
128         z = self.de_1(z)
129         z = self.de_2(z)
130         x_reconst = torch.sigmoid(self.de_3(z))
131 
132         return x_reconst
133 
134     def forward(self, x):
135         z, z_mean, z_log_var = self.encoder(x)
136         x_reconst = self.decoder(z)
137 
138         return z, z_mean, z_log_var, x_reconst
```

Listing A.1: architectures.py

A.2 Utilities Code

```
1  import torch
2  import torchvision.datasets as datasets
3  import torch.utils.data as data
4  import torchvision.transforms as transforms
5  import torch.nn.functional as F
6  import torch.optim as optim
7  import matplotlib.pyplot as plt
8  from mpl_toolkits.mplot3d import Axes3D
9  import numpy as np
10 from sklearn.manifold import TSNE
```

```
import architectures


def loss_function(x_reconst, inputs, z_mean, z_log_var):

    reconst_error = F.binary_cross_entropy(x_reconst, inputs, reduction='sum')
    KL_divergence = -0.5 * torch.sum(1 + z_log_var - z_mean.pow(2) - z_log_var.exp())

    return reconst_error, KL_divergence, reconst_error + KL_divergence


def prepare_vae(input_dim, latent_dim, architecture, device, lr=0.001, weight_decay=1e-5, n_channels=3):
    if architecture == 'fc':
        vae = architectures.VAE_MLP(input_dim, latent_dim, device)
    elif architecture == 'cnn':
        vae = architectures.VAE_CNN(input_dim, latent_dim, device, n_channels)
    vae.to(device)
    optimizer = optim.Adam(vae.parameters(), lr=lr, weight_decay=weight_decay)

    return vae, optimizer


def train(n_epochs, vae, optimizer, trainloader, testloader, device):

    train_losses = []
    train_reconst_errors = []

    for epoch in range(n_epochs):

        vae.train()

        train_loss = 0.0
        train_reconst_error = 0.0
        n_train = 0

        for i, data in enumerate(trainloader, 0):
            inputs = data[0].to(device)
            optimizer.zero_grad()
            z, z_mean, z_log_var, x_reconst = vae(inputs)
```

```
50            reconstruction_error, KL_divergence, loss = loss_function(
    x_reconst, inputs, z_mean, z_log_var)
51            loss.backward()
52            optimizer.step()
53
54            n_train += z.size()[0]
55            train_loss += loss.item()
56            train_reconst_error += reconstruction_error.item()
57
58        train_loss /= n_train
59        train_reconst_error /= n_train
60        print('epoch: %d, training loss: %.3f, reconstruction error: %.3f'
    % (epoch + 1, train_loss, train_reconst_error))
61        train_losses.append(train_loss)
62        train_reconst_errors.append(train_reconst_error)
63
64    print('Finished training!')
65
66    vae.eval()
67
68    n_test = 0
69    test_loss = 0.0
70    test_reconst_error = 0.0
71    for i, data in enumerate(testloader, 0):
72        inputs = data[0].to(device)
73        z, z_mean, z_log_var, x_reconst = vae(inputs)
74        reconstruction_error, KL_divergence, loss = loss_function(x_reconst
    , inputs, z_mean, z_log_var)
75
76        n_test += z.size()[0]
77        test_loss += loss.item()
78        test_reconst_error += reconstruction_error.item()
79
80    test_loss /= n_test
81    test_reconst_error /= n_test
82    print('test loss: %.3f, reconstruction error: %.3f' % (test_loss,
    test_reconst_error))
83
84    x = np.arange(1, n_epochs+1)
85    plt.plot(x, train_reconst_errors, color='red', label='Training
    reconstruction error')
86    plt.plot(x, train_losses, color='blue', label='Training loss')
87    plt.xlabel('epoch')
88    plt.ylabel('loss/reconstruction error')
89    plt.legend(loc="upper right")
```

```
    plt.xlim(left=0)
    plt.xticks(list(range(0, n_epochs+2, 2)))
    plt.grid()
    plt.title('Training Curves')
    plt.show()

    return train_losses, test_loss, train_reconst_errors,
    test_reconst_error


def compare(num_plots, input_size, vae1, vae2, testloader, device):

    vae1.eval()
    vae2.eval()

    fig = plt.figure(figsize=(input_size, input_size))

    for i, data in enumerate(testloader, 0):
        inputs = data[0].to(device)
        x_size = list(inputs.size())
        x_size[0] = 1
        x = inputs[0].view(x_size)
        z1, z_mean1, z_log_var1, x_reconst1 = vae1(x)
        z1, z_mean2, z_log_var2, x_reconst2 = vae2(x)
        if x_size[1] == 1:
            img_reconst1 = x_reconst1.cpu().detach().numpy().reshape(x_size
    [2:])
            img_reconst2 = x_reconst2.cpu().detach().numpy().reshape(x_size
    [2:])
            img = x.cpu().detach().numpy().reshape(x_size[2:])
        else:
            img_reconst1 = x_reconst1.cpu().detach().numpy().reshape(x_size
    [1:]).transpose(1, 2, 0)
            img_reconst2 = x_reconst2.cpu().detach().numpy().reshape(x_size
    [1:]).transpose(1, 2, 0)
            img = x.cpu().detach().numpy().reshape(x_size[1:]).transpose(1,
     2, 0)

        # Left: ground truth
        fig.add_subplot(num_plots, 3, i*3+1)
        plt.axis('off')
        if x_size[1] == 1:
            plt.imshow(img, cmap = 'gray_r')
        else:
            plt.imshow(img)
```

```
129
130
131         fig.add_subplot(num_plots, 3, i*3+2)
132         plt.axis('off')
133         if x_size[1] == 1:
134             plt.imshow(img_reconst1, cmap = 'gray_r')
135         else:
136             plt.imshow(img_reconst1)
137
138         fig.add_subplot(num_plots, 3, i*3+3)
139         plt.axis('off')
140         if x_size[1] == 1:
141             plt.imshow(img_reconst2, cmap = 'gray_r')
142         else:
143             plt.imshow(img_reconst2)
144
145         if i >= num_plots-1:
146             break
147
148     plt.show()
149
150
151 def manifold_sample(n, input_size, vae, device):
152
153     vae.eval()
154
155     figure = np.zeros([input_size * n, input_size * n])
156
157     grid_x = np.linspace(-4, 4, n)
158     grid_y = np.linspace(-4, 4, n)[::-1]
159
160
161     for i, yi in enumerate(grid_y):
162         for j, xi in enumerate(grid_x):
163             z = np.array([[xi, yi]]).reshape(1, 2)
164             z = torch.from_numpy(z).float().to(device)
165             x_reconst = vae.decoder(z).cpu().detach().numpy().reshape(
    input_size, input_size)
166             figure[i * input_size: (i + 1) * input_size, j * input_size: (j
     + 1) * input_size] = x_reconst
167
168     plt.figure(figsize=(input_size//2, input_size//2))
169     start_range = input_size // 2
170     end_range = n * input_size + start_range + 1
171     pixel_range = np.arange(start_range, end_range, input_size)
```

```
172     sample_range_x = np.round(grid_x, 1)
173     sample_range_y = np.round(grid_y, 1)
174     plt.xticks(pixel_range, sample_range_x)
175     plt.yticks(pixel_range, sample_range_y)
176     plt.xlabel("z_1")
177     plt.ylabel("z_2")
178     plt.imshow(figure, cmap='gray_r')
179     plt.show()
180
181
182 def manifold_scatter(vae, testloader, device, tsne=False):
183
184     vae.eval()
185
186     target = []
187     latent = []
188
189     for i, data in enumerate(testloader, 0):
190         inputs, labels = data[0].to(device), data[1].tolist()
191         target += labels
192         z, z_mean, z_log_var, x_reconst = vae(inputs)
193         z = z.cpu().detach().numpy()
194         latent.append(z)
195
196     latent = np.concatenate(latent)
197
198     if tsne:
199         latent = TSNE(n_components=2, perplexity=50).fit_transform(latent)
200
201     plt.figure(figsize=(15, 15))
202     plt.scatter(latent[:, 0], latent[:, 1], c=target, s=40)
203     plt.colorbar()
204     plt.xlabel("z_1")
205     plt.ylabel("z_2")
206     plt.show()
207
208
209
210 def get_dataset(dataset, batch_size, root='./data'):
211
212     transform = transforms.Compose([transforms.ToTensor()])
213
214     if dataset == 'MNIST':
215         trainset = datasets.MNIST(root=root, train=True, download=True,
    transform=transform)
```

```
216         trainloader = data.DataLoader(trainset, batch_size=batch_size,
    shuffle=True)
217 
218         testset = datasets.MNIST(root=root, train=False, download=True,
    transform=transform)
219         testloader = data.DataLoader(testset, batch_size=batch_size,
    shuffle=False)
220 
221     elif dataset == 'FashionMNIST':
222         trainset = datasets.FashionMNIST(root=root, train=True, download=
    True, transform=transform)
223         trainloader = data.DataLoader(trainset, batch_size=batch_size,
    shuffle=True)
224 
225         testset = datasets.FashionMNIST(root=root, train=False, download=
    True, transform=transform)
226         testloader = data.DataLoader(testset, batch_size=batch_size,
    shuffle=False)
227 
228     elif dataset == 'CIFAR10':
229         trainset = datasets.CIFAR10(root=root, train=True, download=True,
    transform=transform)
230         trainloader = data.DataLoader(trainset, batch_size=batch_size,
    shuffle=True)
231 
232         testset = datasets.CIFAR10(root=root, train=False, download=True,
    transform=transform)
233         testloader = data.DataLoader(testset, batch_size=batch_size,
    shuffle=False)
234 
235     elif dataset == 'SVHN':
236         trainset = datasets.SVHN(root=root, split='train', download=True,
    transform=transform)
237         trainloader = data.DataLoader(trainset, batch_size=batch_size,
    shuffle=True)
238 
239         testset = datasets.SVHN(root=root, split='test', download=True,
    transform=transform)
240         testloader = data.DataLoader(testset, batch_size=batch_size,
    shuffle=False)
241 
242     else:
243         raise NameError('Unknown dataset!')
244 
```

```
245     return trainloader, testloader
```

Listing A.2: utils.py

A.3 MNIST

```
1  from utils import *
2
3
4  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
5  print(device)
6
7  torch.manual_seed(723)
8  np.random.seed(723)
9
10 input_size = 28
11 input_dim = input_size**2
12 batch_size = 128
13
14 trainloader, testloader = get_dataset(dataset='MNIST', batch_size=
       batch_size, root='./data')
15
16 latent_dims = [2, 5, 10, 15, 20, 25]
17 n_epochs = 20
18 t_losses = []
19 t_reconst_errors = []
20 vaes = []
21
22
23 for latent_dim in latent_dims:
24     print('latent_dim = %d' % (latent_dim))
25     vae, optimizer = prepare_vae(input_dim, latent_dim, 'fc', device, lr
       =0.001, weight_decay=1e-5)
26     vaes.append(vae)
27     _, test_loss, _, test_reconst = train(n_epochs, vae, optimizer,
       trainloader, testloader, device)
28     t_losses.append(test_loss)
29     t_reconst_errors.append(test_reconst)
30
31
32 plt.plot(latent_dims, t_reconst_errors, marker='o', color='red', label='
       Test reconstruction error')
33 plt.plot(latent_dims, t_losses, marker='o', color='blue', label='Test loss'
       )
34 plt.xlabel('latent space dimension')
```

```
plt.ylabel('loss/reconstruction error')
plt.xlim(left=0)
plt.ylim(bottom=70)
plt.title('Test Loss/Reconstruction Error')
plt.legend(loc="upper right")
plt.grid()
plt.show()


compare(30, input_size, vaes[0], vaes[-2], testloader, device)
manifold_sample(30, input_size, vaes[0], device)
manifold_scatter(vaes[0], testloader, device)
manifold_scatter(vaes[-2], testloader, device, True)
```

Listing A.3: MNIST.py

A.4 Fashion-MNIST

```
from utils import *


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

torch.manual_seed(237)
np.random.seed(237)

input_size = 28
input_dim = input_size**2
batch_size = 128

trainloader, testloader = get_dataset(dataset='FashionMNIST', batch_size=
    batch_size, root='./data')


latent_dims = [2, 5, 10, 15, 20, 25]
n_epochs = 20
t_losses = []
t_reconst_errors = []
vaes = []

for latent_dim in latent_dims:
    print('latent_dim = %d' % (latent_dim))
    vae, optimizer = prepare_vae(input_dim, latent_dim, 'fc', device, lr
    =0.001, weight_decay=1e-5)
```

```
    vaes.append(vae)
    _, test_loss, _, test_reconst = train(n_epochs, vae, optimizer,
    trainloader, testloader, device)
    t_losses.append(test_loss)
    t_reconst_errors.append(test_reconst)


plt.plot(latent_dims, t_reconst_errors, marker='o', color='red', label='
    Test reconstruction error')
plt.plot(latent_dims, t_losses, marker='o', color='blue', label='Test loss'
    )
plt.xlabel('latent space dimension')
plt.ylabel('loss/reconstruction error')
plt.ylim(bottom=220)
plt.xlim(left=0)
plt.title('Test Loss/Reconstruction Error')
plt.legend(loc="upper right")
plt.grid()
plt.show()


compare(30, input_size, vaes[0], vaes[-2], testloader, device)
manifold_sample(30, input_size, vaes[0], device)
manifold_scatter(vaes[0], testloader, device)
manifold_scatter(vaes[-2], testloader, device, True)
```

Listing A.4: FashionMNIST.py

A.5 CIFAR-10

```
from utils import *


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

torch.manual_seed(723)
np.random.seed(723)

input_size = 32
input_dim = input_size*input_size*3
batch_size = 128

trainloader, testloader = get_dataset(dataset='CIFAR10', batch_size=
    batch_size, root='./data')
```

```
15
16
17 latent_dim = 256
18 n_epochs = 25
19
20 vaeCNN, optimizer = prepare_vae(input_dim, latent_dim, 'cnn', device, lr=1e
       -3, weight_decay=1e-5)
21 train(n_epochs, vaeCNN, optimizer, trainloader, testloader, device)
22
23
24 latent_dim = 256
25 n_epochs = 25
26
27 vaeMLP, optimizer = prepare_vae(input_dim, latent_dim, 'fc', device, lr=1e
       -3, weight_decay=1e-5)
28 train(n_epochs, vaeMLP, optimizer, trainloader, testloader, device)
29
30
31 compare(10, input_size, vaeMLP, vaeCNN, testloader, device)
32 manifold_scatter(vaeMLP, testloader, device, True)
33 manifold_scatter(vaeCNN, testloader, device, True)
```

Listing A.5: CIFAR10.py