DEPARTMENT OF INFORMATICS
MSc IN COMPUTER SCIENCE

Postgraduate Dissertation for
Master of Science Degree

Variational Autoencoders & Applications

Student:
Christos KORMARIS
ΕΥ1617

Supervisor Professor:
Dr. Michalis TITSIAS

ATHENS, MAY 2018

Declaration of Authorship

I, Christos Kormaris, declare that this thesis titled, 'Variational Autoencoders' and the work presented in it are my own. I confirm that:

- This work was done wholly while in candidature for a research degree at Athens University of Economics & Business.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Athens University of Economics & Business or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Original Text in Ancient Greek

“ ξεῖν’, ἦ τοι μὲν ὄνειροι ἀμήχανοι ἀκριτόμυθοι
γίγνοντ’, οὐδέ τι πάντα τελείεται ἀνθρώποισι.
δοιαί γάρ τε πύλαι ἀμενηνῶν εἰσίν ὀνείρων·
αἱ μὲν γὰρ κεράεσσι τετεύχαται, αἱ δ’ ἐλέφαντι·
τῶν οἳ μέν κ’ ἔλθωσι διὰ πριστοῦ ἐλέφαντος,
οἵ ῥ’ ἐλεφαίρονται, ἔπε’ ἀκράαντα φέροντες·
οἳ δὲ διὰ ξεστῶν κεράων ἔλθωσι θύραζε,
οἵ ῥ’ ἔτυμα κραίνουσι, βροτῶν ὅτε κέν τις ἴδηται. “

῾Ομήρου Ὀδύσσεια, ῾Ραψῳδία τ, στ. 562 (Original text)

English Translation

“Oneiroi are beyond our unravelling –who can be sure what tale they tell? Not all that men look for comes to pass. Two gates there are that give passage to fleeting Oneiroi; one is made of horn, one of ivory. The Oneiroi that pass through sawn ivory are deceitful, bearing a message that will not be fulfilled; those that come out through polished horn have truth behind them, to be accomplished for men who see them.“

Homer, Odyssey 19. 562 ff (Shewring translation)

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS

Abstract

School of Information Sciences & Technology
Department of Informatics

Master of Science in Computer Science

Variational Autoencoders & Applications

by Student: Christos Kormaris
Supervisor Professor: Dr. Michalis Titsias

A variational autoencoder is a method that can produce artificial data which will resemble a given dataset of real data. For instance, if we want to produce new artificial images of cats, we can use a variational autoencoder algorithm to do so, after training on a large dataset of images of cats. The input dataset is unlabeled on the grounds that we are not interested in classifying the data to a specific class, but we would rather be able to learn the most important features or similarities among the data. Because of the fact that the data are not labeled, the variational autoencoder is described as an unsupervised learning algorithm. As far as the example of cat images is concerned, the algorithm can learn to detect that a cat should have two ears, a nose, whiskers, four legs, a tail and a diversity of colors. The algorithm uses two neural networks, an encoder and a decoder, which are trained simultaneously. A variational autoencoder should have good applications in cases where we would like to produce a bigger dataset, for better training on various neural networks. Also, it runs dimensionality reduction on the initial data, by compressing them into latent variables. We run implementations of variational autoencoders on various datasets, MNIST, Binarized MNIST, CIFAR-10, OMNIGLOT, YALE Faces, ORL Face Database, MovieLens, written in Python 3 with three different libraries, TensorFlow, PyTorch and Keras and we present the results. We introduce a simple missing values completion algorithm using K-NN collaborative filtering for making predictions (e.g. on missing pixels). Finally, we make use of the variational autoencoders to run missing values completion algorithms and predict missing values on various datasets. The K-NN algorithm did surprisingly well on the predictions, while the variational autoencoder completion system brought very satisfactory results. A graphical user interface has also been implemented, as well.

Acknowledgements

I would like to thank professor *Dr. Michalis Titsias* for his help and his valuable advice. I would also like to thank the director of the Master Program, professor *Dr. George Polyzos*, for his understanding and his encouragement. Finally, I want to thank the professors of the Master Program, *Dr. Georgios Stamoulis*, *Dr. Yannis Kotidis*, *Dr. Vana Kalogeraki*, *Dr. Vangelis Markakis*, *Dr. Stavros Toumpis*, *Dr. Ion Androutsopoulos*, *Dr. Georgios Papaioannou*, *Dr. Michalis Vazirgiannis*, *Dr. Vasilios Siris* & *Dr. Sophia Dimeli*, for all the knowledge I have received.

Contents

List of Figures

List of Tables

Listings

Abbreviations

CF	**C**ollaborative **F**iltering
CIFAR dataset	**C**anadian **I**nstitute **f**or **A**dvanced **R**esearch dataset
ELBO	**E**vidence **L**ower **Bo**und, also Variational Lower Bound
EM algorithm	**E**xpectation **M**aximization algorithm
KL divergence	**K**ullback-**L**eibler divergence
K-NN	**K** **N**earest **N**eighbors
MNIST dataset	**M**odified **N**ational **I**nstitute of **S**tandards & **T**echnology dataset
VAE	**V**ariational **A**uto**e**ncoder
VB methods	**V**ariational **B**ayes methods

Symbols

$D_{KL}(P \,\|\| \, Q)$	**K**ullback-**L**eibler **D**ivergence between two distributions **P** and **Q**
E	Mean value
L	Variational **L**ower **Bo**und (ELBO)
$N(\mu, \sigma^2)$	**N**ormal distribution of Mean Value μ and Variance σ^2
$P(X)$	**P**robability distribution of a random variable X
x	Input data variable - One example
X	Input data variables - Many examples
z	Latent variable
Z	Latent variables
D	Input data dimensionality / Number of pixels
M_1	Number of neurons in the encoder
M_2	Number of neurons in the decoder
Z_{dim}	Latent variable dimensionality
$\epsilon \sim N(\mu, \sigma^2)$	Random variable ϵ follows Normal distribution
θ	Decoder parameters
μ	Mean value
M	Mean values
σ	Standard deviation
Σ	Standard deviations
σ^2	Variance
Σ^2	Variances
ϕ	Encoder parameters
@	Matrix multiplication operator in PyTorch
$.*$	Element-wise matrix multiplication operator

I dedicate my Thesis to my parents.

Chapter 1

Introduction

1.1 Preliminaries

Let X be our training set. We aim to maximize the probability of each training instance x in the training set X, according to:

$$P(X) = \int P(X, z)\, dz = \int P(X \mid z, \phi) \cdot P(z)\, dz$$

where Z is a continuous and NOT a discrete distribution and every z is an instance of Z. Hence, we use an integral of joint distributions and not a sum, to estimate the distribution X. X could be either a continuous or a discrete distribution. For instance, if our dataset describes images, X could be a Bernoulli discrete distribution taking binary values 1 or 0, 1 for white pixel color and 0 for black pixel color. On the other hand, if X contains real values between the interval $[0, 1]$, then the distribution is continuous. With a continuous distribution, our dataset can represent even more colors. (Carl Doersch 2016. [2])

1.2 Model Representation

The input data of our model are mainly images. The input images are being represented as a variable of size N x D. N denotes the number of examples and D denotes the number of pixels. Each image should have the same number of pixels. A good example, is the *MNIST* dataset, which contains images of digits, from *0-9*. Other examples of datasets with images are the *Binarized MNIST* dataset, the *CIFAR-10 / CIFAR-100* datasets, *OMNIGLOT* dataset, the *YALE Faces* dataset & the *The Database of Faces* dataset. We will also examine the *MovieLens* dataset, which provides the ratings that many users have given to certain movies.

The variational autoencoder is a sequence of two neural networks one after the other. The first neural is called the encoder and the second neural network is called the decoder. Also, the encoder and the decoder are trained simultaneously. The purpose of the encoder is to

learn how to represent the hidden features of the given dataset, using latent variables of lower dimensionality to store them. On the other end, the decoder constructs artificial data, from the latent variables we have learned. The artificial data must be similar to our original data and NOT exactly the same. Otherwise, we have failed. Once the decoder finishes, our goal is complete.

1.3 Variational Inference

First, we want to calculate the encoder, i.e. we want to estimate:

$$P(Z \mid X) = \frac{P(X \mid Z) \cdot P(Z)}{P(X)} = \frac{P(X \mid Z) \cdot P(Z)}{\int P(X, Z)\, dz}$$

Each term can be represented as follows:

$$\textbf{posterior} = \frac{\textbf{likelihood} \cdot \textbf{prior}}{\textbf{normalizing constant}}$$

Where posterior: *P(Z|X)*, likelihood: *P(X|Z)*, prior: *P(Z)*, normalizing constant: *P(X)*. The denominator *P(X)*, which is called *normalizing constant*, is also called *evidence*. We can calculate it by marginalizing out the latent variables Z:

$$P(X) = \int P(X \mid Z, \theta) \cdot P(Z)\, dz = \int P(X, Z, \theta)\, dz$$

However, calculating this integral requires exponential time, because the distribution of the latent variables Z is continuous. The term $P(X \mid Z, \theta)$ is a complicated likelihood function, because of the non-linearity of the hidden layers. The problem of maximizing the term *logP(Z|X)*, via Bayes' rule, reduces to:

$$logP(Z \mid X) = log\frac{P(X \mid Z) \cdot P(Z)}{P(X)} = logP(X \mid Z) + logP(Z) - logP(X)$$

Since the term $P(X)$ is intractable, the $P(Z \mid X)$ term is intractable too via using Bayes' rule. To estimate $P(Z \mid X)$, we will use another method called **variational inference**, using a family of distributions we are calling $Q_\phi(Z \mid X)$, where φ is a parameter. We learn the parameter φ with stochastic (or mini-batch) gradient ascent (or descent). In each iteration, we compute the cost function or the likelihood, which is the lower bound of the

term $logP(X)$. We want to maximize this term, thus we want to maximize the lower bound. We will analyze this process further in Chapter 2.

Using variational inference, we have made the estimation of the term $P(Z \mid X)$ tractable. For the second part of the variational autoencoder, the decoder, we want to estimate the term $P_\theta(X \mid Z)$. We will use stochastic (or mini-batch) gradient ascent (or descent) to learn the parameters $\boldsymbol{\vartheta}$.

1.3.1 Benefits of Variational Inference

(Max Welling, Diederik P. Kingma 2013. [3])

1. **Managing Intractability:** As explained above, the calculation of the term $P(X)$ is intractable and so is the term $P(Z \mid X)$. Thus, the **Expectation-Maximization (EM)** algorithm and other mean-field Variational Bayes algorithms cannot be used. This is where the variational inference comes in to offer a solution.
2. **Large datasets:** Optimization algorithms that train on the whole set of data on each iteration (e.g. batch gradient descent) are too costly. With variational inference, the parameters are updated using small mini-batches or even single data points (e.g. stochastic gradient descent). Sampling based solutions, such as Monte Carlo EM would be too slow since they involve expensive sampling loops per datapoint.

1.3.2 Drawbacks of Variational Autoencoders

(Aaron Courville, Ian Goodfellow, Yoshua Bengio 2016. [4])

The variational autoencoder approach is elegant, theoretically pleasing, and simple to implement. It also obtains almost excellent results and is among the state of the art approaches to generative modeling. Its main drawback is that samples from variational autoencoders trained on images tend to be somewhat blurry. The causes of this phenomenon are not yet known and remain to be researched. One possibility is that blurriness is an intrinsic effect of maximum likelihood, which minimizes the divergence $D_{KL}(P(Z|X) \,||\, Q(Z|X))$. This means that the model will assign high probability to points that occur in the training set, but may also assign high probability to other points. These other points may include blurry images.

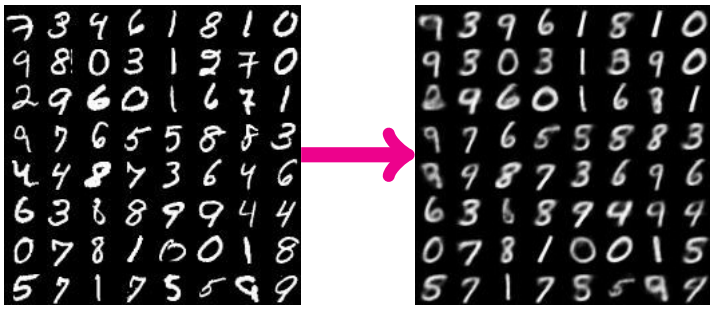


FIGURE 1.1: An example of the input and the output of a VAE. As you can observe, the resulting data are somewhat blurry.

1.3.3 Main Steps for a Variational Autoencoder

1. Get the input data (images) X, as a variable of size N x D.
2. Train the encoder, which is denoted as $Q_\phi(Z \mid X)$, using batch gradient descent (where φ are the encoder parameters and Z are the latent variables with reduced dimensions).
3. Synchronously with the encoder, we train the decoder, which is denoted as $P_\theta(X \mid Z)$, using batch gradient descent (where ϑ are the decoder parameters).
4. We are ready to show and use our new artificial data (images) $\tilde{\mathbf{X}}$, which should resemble the original data.

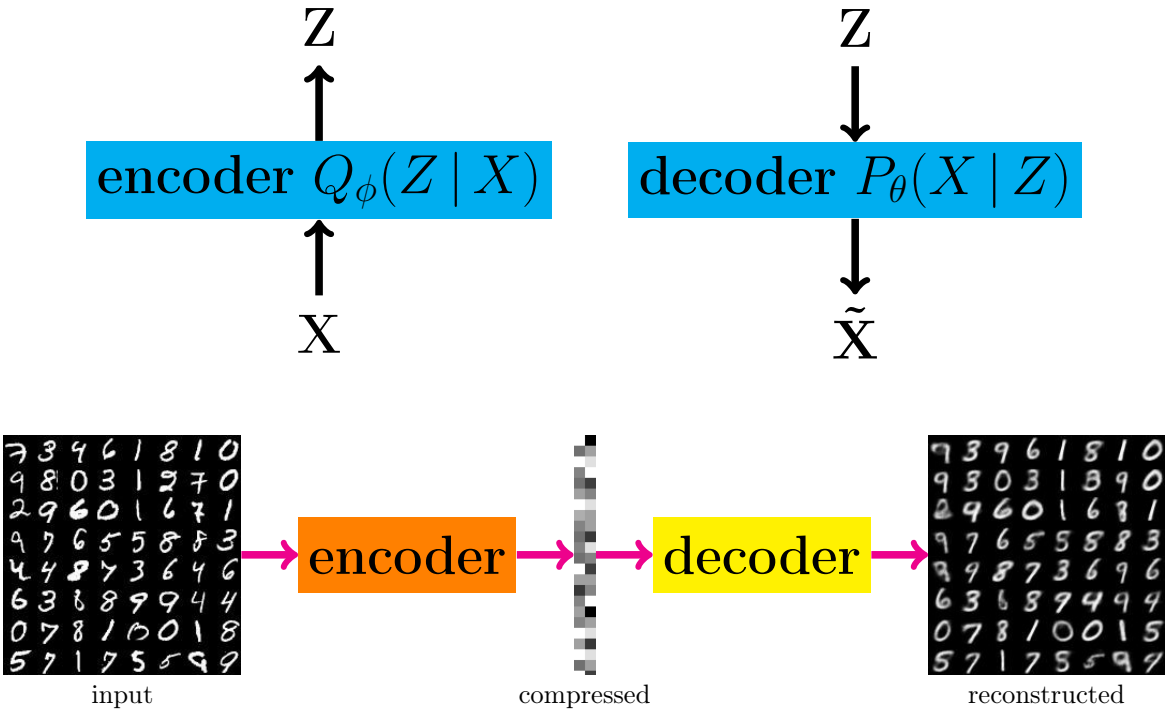


FIGURE 1.2: Variational Autoencoders schema.

Chapter 2

Estimating the Variational Lower Bound (ELBO)

2.1 Using Jensen's Inequality to calculate the Variational Lower Bound

Since the term $P(Z|X)$ is intractable, as we explained in Chapter 1, we use an arbitrary distribution $Q(Z)$ to approximate the true posterior distribution $P(Z|X)$.

$$logP(X) = log \int P(X,z)\,dz = log \int P(X,z) \cdot \frac{Q(Z)}{Q(Z)}\,dz = log(E_q[\frac{P(X,z)}{Q(Z)}])$$

From Jensen's inequality for concave functions we have $f(E[X]) \geq E[f(X)]$. Since the logarithmic function is concave, we have:

$$logP(X) \geq E_q[logP(X,z)] - E_q[logQ(Z)]$$

Let us denote:

$$ELBO = L(X,Q) = E_q[logP(X,z)] - E_q[logQ(Z)]$$

Then, it is obvious that ELBO is a lower bound of the log probability of the observations ($logP(X)$). As a result, if in some cases we want to maximize the marginal probability, we can instead maximize its variational lower bound (ELBO).

2.2 Using KL Divergence to calculate the Variational Lower Bound

(Carl Doersch 2016. [2]) In the case of Variational Autoencoders, the KL-divergence (Kullback-Leibler divergence) is the likelihood between the real distribution of the latent variables Z given X, $P(Z|X)$ and the estimated distribution of the latent variable Z given X, $Q_\phi(Z|X)$. For the latter term, $Q_\phi(Z|X)$, the following equality stands:

$$Q_\phi(Z|X) \approx Q_\phi(Z)$$

The KL divergence between the two distributions takes the following form:

$$D_{KL}[Q_\phi(Z) \,||\, P(Z|X)] = E_{Z\sim Q}[\frac{logQ_\phi(Z)}{logP(Z|X)}] \Rightarrow$$
$$D_{KL}[Q_\phi(Z) \,||\, P(Z|X)] = E_{Z\sim Q}[logQ_\phi(Z) - logP(Z|X)] \Rightarrow$$

where D denotes the KL-divergence between two distributions.

After applying Bayes rule on the second term, we have:

$$D_{KL}[Q_\phi(Z) \,||\, P(Z|X)] = E_{Z\sim Q}[logQ_\phi(Z) - log[\frac{P_\theta(X|Z)\cdot P(Z)}{P(X)}]] \Rightarrow$$
$$D_{KL}[Q_\phi(Z) \,||\, P(Z|X)] = E_{Z\sim Q}[logQ_\phi(Z) - logP_\theta(X|Z) - logP(Z) + logP(X)] \Rightarrow$$
$$logP(X) - D_{KL}[Q_\phi(Z) \,||\, P(Z|X)] = E_{Z\sim Q}[logP_\theta(X|Z)] - D_{KL}[Q_\phi(Z) \,||\, P(Z)] \Rightarrow$$
$$logP(X) - D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = E_{Z\sim Q}[logP_\theta(X|Z)] - D_{KL}[Q_\phi(Z|X) \,||\, P(Z)]$$

The last equation is the variational lower bound, which we will call **ELBO** from now on. The left side of the equation has the term we want to maximize, $P(X)$, plus an error term. The error term is the Kullback-Leibler divergence between $Q_\phi(Z|X) \approx Q_\phi(Z)$ and $P(Z|X)$, which makes Q produce latent variables Z, given input variables X. We want to minimize the Kullback-Leibler divergence between the two distributions. This problem reduces to maximizing the ELBO term. If the distribution Q is approximated with great accuracy, then the error term becomes small.

To summarize, the ELBO is obtained from this formula:

$$ELBO = L(X,Q) = logP(X) - D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] \Rightarrow$$

$$logP(X) = L(X,Q) + D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)]$$

since the KL-divergence is non-negative, we have:

$$logP(X) \geq L(X,Q)$$

which is why we call L, ELBO or variational lower bound (Max Welling, Diederik P. Kingma 2013. [3]). ELBO is also equal to:

$$ELBO = L(X,Q) = E_{Z\sim Q}[logP_\theta(X|Z)] - D_{KL}[Q_\phi(Z|X) \,||\, P(Z)] \Longrightarrow_{Q_\phi(Z|X) \approx Q_\phi(Z)}$$

$$\mathbf{L(X,Q)} = \mathbf{E_{Z\sim Q}[logP_\theta(X|Z)]} - \mathbf{D_{KL}[Q_\phi(Z) \,||\, P(Z)]}$$

The term $E_{Z\sim\ Q}[logP_\theta(X|Z)]$ is called **reconstruction cost**. The term $D_{KL}[Q_\phi(Z)||P(Z)]$ is called **penalty** or **regularization term**. The penalty term ensures that the explanation of the data, $Q_\phi(Z|X) \approx Q_\phi(Z)$ doesn't deviate too far from the beliefs term $P(Z)$. The penalty term also helps us apply **Occam's Razor** (aka Ockham's Razor) to our inference model. It is always greater or equal to 0 and so it can be omitted.

We can train simultaneously both the encoder and the decoder.

FIGURE 2.1: The Elbo Room is a bar located in the historic Mission District of San Francisco on Valencia Street between 17th Street and 18th Street.

Chapter 3

Back-Propagation Algorithm

3.1 Optimizing the Objective (ELBO)

As we mentioned in the previous chapter, we simultaneously train both the encoder (inference model), $Q_\phi(Z|X)$ and the decoder (generative model), $P_\theta(X|Z)$, by optimizing the variational lower bound (ELBO), using gradient **back-propagation**. We have have obtained the following formula for the ELBO:

$$L(X, Q) = E_{Z\sim Q}[logP_\theta(X|Z)] - D_{KL}[Q_\phi(Z) \,||\, P(Z)]$$

The update rules are determined based on the selected back-propagation algorithm.

Now, we will try find a suitable formula, for the KL-divergence between the real distribution $P(Z|X)$ and the latent distribution $Q(Z|X)$. (Carl Doersch 2016. [2])

we have:

$$\mathbf{Q}_\phi(\mathbf{Z}) = N_1 = N(Z|\mu_1, \sigma_1^2) = N(Z|M, \Sigma^2), \text{ where: } \mu_1 = M \text{ and } \sigma_1 = \Sigma$$

$$\mathbf{P}(\mathbf{Z}) = N_2 = N(Z|\mu_2, \sigma_2^2) = N(Z|0, I), \text{ where: } \mu_2 = 0 \text{ and } \sigma_2 = I$$

we also have:

$$\int \mathbf{Q}_\phi(\mathbf{Z}) \cdot \log \mathbf{P}(\mathbf{Z})\, \mathbf{dz} = \int N(Z|M, \Sigma^2) \cdot \log N(Z|0, I)\, dz = -\frac{J}{2}\log 2\pi - \frac{1}{2}\cdot\sum_{j=1}^{J}(\mu_j^2 + \sigma_j^2) \Rightarrow$$

$$\int \mathbf{Q}_\phi(\mathbf{Z}) \cdot \log \mathbf{P}(\mathbf{Z})\, \mathbf{dz} = -\frac{J}{2}\log 2\pi - \frac{1}{2}\cdot(M^2 + \Sigma^2)\ (\mathbf{1})$$

and:

$$\int \mathbf{Q}_\phi(\mathbf{Z}) \cdot \log \mathbf{Q}_\phi(\mathbf{Z})\, \mathbf{dz} = \int N(Z|M, \Sigma^2) \cdot \log N(Z|M, \Sigma^2)\, dz = -\frac{J}{2}\log 2\pi - \frac{1}{2}\cdot\sum_{j=1}^{J}(1 + \log \sigma_j^2) \Rightarrow$$

$$\int \mathbf{Q}_\phi(\mathbf{Z}) \cdot \log \mathbf{Q}_\phi(\mathbf{Z})\, \mathbf{dz} = -\frac{J}{2} \log 2\pi - \frac{1}{2} \cdot (1 + \log \Sigma^2)\ (\mathbf{2})$$

where J is the dimensionality of the latent variables Z. Now, M and Σ are defined as follows:

$$M = \begin{bmatrix} M_{11} & M_{12} & M_{13} & \dots & M_{1J} \\ M_{21} & M_{22} & M_{23} & \dots & M_{2J} \\ & \dots\dots\dots\dots & & & \\ M_{N1} & M_{N2} & M_{N3} & \dots & M_{NJ} \end{bmatrix}, \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} & \Sigma_{13} & \dots & \Sigma_{1J} \\ \Sigma_{21} & \Sigma_{22} & \Sigma_{23} & \dots & \Sigma_{2J} \\ & \dots\dots\dots\dots & & & \\ \Sigma_{N1} & \Sigma_{N2} & \Sigma_{N3} & \dots & \Sigma_{NJ} \end{bmatrix}$$

where N is the number of examples

So, we can calculate the Kullback-Leibler divergence between the distributions P and Q of the ELBO formula, as follows:

$$D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = D_{KL}[N_1 \,||\, N_2] = D_{KL}[N(Z|\mu_1, \sigma_1^2) \,||\, N(Z|\mu_2, \sigma_2^2)] \Rightarrow$$

$$D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = D_{KL}[N(Z|\mu_1, \sigma_1^2) \,||\, N(Z|0, I)] \Rightarrow$$

$$D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = \int Q_\phi(Z) \cdot \log \frac{P(Z)}{Q_\phi(Z)}\, dz \Rightarrow$$

$$D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = \int Q_\phi(Z) \cdot (\log P(Z) - \log Q_\phi(Z))\, dz \Rightarrow$$

$$D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = \int Q_\phi(Z) \cdot \log P(Z) - Q_\phi(Z) \cdot \log Q_\phi\, dz \Longrightarrow_{(\mathbf{1}),(\mathbf{2})}$$

$$D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = -\frac{J}{2} \log 2 \cdot \pi - \frac{1}{2} \cdot \sum_{j=1}^{J} (\mu_j^2 + \sigma_j^2) - (-\frac{J}{2} \log 2 \cdot \pi - \frac{1}{2} \cdot \sum_{j=1}^{J} (1 + \log \sigma_j^2)) \Rightarrow$$

$$D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = -\frac{J}{2} \log 2 \cdot \pi + \frac{J}{2} \cdot \log 2 \cdot \pi - \frac{1}{2} \cdot \sum_{j=1}^{J} (\mu_j^2 + \sigma_j^2) + \frac{1}{2} \cdot \sum_{j=1}^{J} (1 + \log \sigma_j^2) \Rightarrow$$

$$D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = \frac{1}{2} \cdot \sum_{j=1}^{J}(1 + \log \sigma_j^2 - \mu_j^2 - \sigma_j^2) \Rightarrow$$

$$D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = \frac{1}{2} \cdot (J + \log \Sigma^2 - M^2 - \Sigma^2)$$

if the dimensionality of the parameter $J = 1$, of the latent variables Z, which means that we have univariate Gaussian distributions and we end up with the formula:

$$\mathbf{D_{KL}[Q_\phi(Z|X) \,||\, P(Z|X)] = \frac{1}{2} \cdot (1 + \log \Sigma^2 - M^2 - \Sigma^2)}$$

Let us remember that the KL-divergence term has a negative sign in the variational lower bound (ELBO) formula, thus, we want to minimize it.

3.2 The Reparameterization Trick

(Carl Doersch 2016. [2]) After having found a suitable formula for the KL-divergence term, we still have to do stochastic (or mini-batch) gradient descent over different values of X sampled from a dataset D. The full equation we want to optimize is:

$$E_{X\sim D}[E_{Z\sim Q}[logP_\theta(X|Z)] - D_{KL}[Q_\phi(Z) \,||\, P(Z)]] \,(\mathbf{1})$$

We want to take the gradient of this equation. The gradient symbol can be moved into the expectations. Therefore, we can sample a single value of X and a single value of Z from the distribution $Q(Z|X)$ and compute the gradient of:

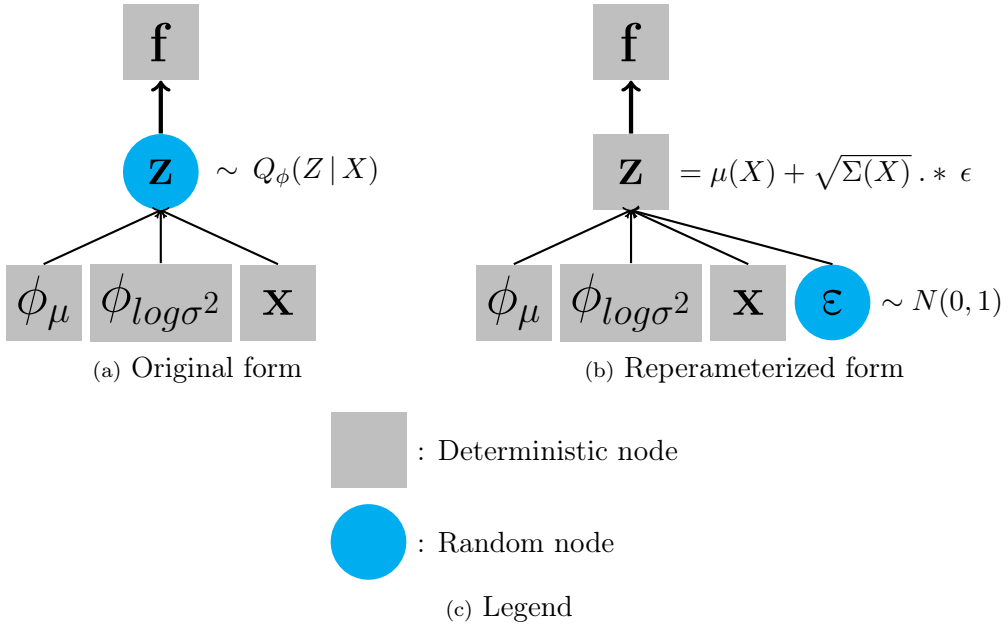
$$logP_\theta(X|Z) - D_{KL}[Q_\phi(Z) \,||\, P(Z)](\mathbf{2})$$

We can then average the gradient of this function over arbitrarily many samples of X and Z, and the result converges to the gradient of Equation 1. There is, however, a significant problem with Equation 2. $E_{Z\sim Q}[logP_\theta(X|Z)]$ depends not just on the parameters of P, but also on the parameters of Q. However, in Equation 2, this dependency has disappeared! In order to make VAEs work, it's essential to drive Q to produce codes for X that P can reliably decode. The forward pass of this network works fine and, if the output is averaged over many samples of X and Z, produces the correct expected value. However, we need

to back-propagate the error through a layer that samples Z from $Q(Z|X)$, which is a non-continuous operation and has no gradient. Stochastic gradient descent via back-propagation can handle stochastic inputs, but not stochastic units within the network! The solution, called the **"reparameterization trick"**, is to move the sampling to an input layer. Given $\mu(X)$ and $\Sigma(X)$ - the mean and covariance of $Q(Z|X)$ - we can sample from the Normal distribution $N(\mu(X), \Sigma(X))$ by first sampling $\epsilon \sim N(0, I)$. Then, we can compute the $Z = \mu(X) + \sqrt{\Sigma(X)} \,.*\, \epsilon$, which is a Gaussian distribution, $Z \sim N(\mu(X), \Sigma(X))$, since any linear transformation of a Gaussian random variable is again Gaussian (see Appendix B, Gaussian (or Normal) Distribution). Thus, the equation we actually take the gradient of is:

$$E_{X\sim D}[E_{Z\sim Q}[logP_\theta(X|Z = \mu(X) + \sqrt{\Sigma(X)} \cdot \epsilon)] - D_{KL}[Q_\phi(Z) \,||\, P(Z)]]$$

It is worth noticing that the distribution $Q(Z|X)$ (and therefore $P(Z)$) must be continuous! The reparameterization trick allows us to make the computation of the gradient of the mean value of the ELBO and thus back-propagation can be applied.



(a) Original form (b) Reperameterized form (c) Legend

FIGURE 3.1: The right network uses the reparameterization trick. Back-propagation can only be applied to this network.

3.3 Calculating the update rules of the weights of the encoder and the decoder

In the implementations presented in this thesis, the update rules for the weights (Phis) of the encoder and the weights (Thetas) of the decoder are calculated automatically. Both TensorFlow and PyTorch libraries contain builtin backprogration algorithms that calculate the gradients of the weights used for learning. The update rules, which are also included as a builtin function, are then applied using the gradients calculated before.

To make it more clear, here's an example of calling the builtin back-propagation algorithm and the update weights function in TensorFlow:

```
var_list  # the weights and the biases of the variational autoencoder
elbo  # the loss function of the variational autoencoder to be minimized
lr  # learning rate for the weights and the biases updates

# Adam Optimizer #
grads_and_vars = tf.train.AdamOptimizer(learning_rate=lr).
        compute_gradients(loss=elbo, var_list=var_list)
apply_updates = tf.train.AdamOptimizer(learning_rate=lr).
        apply_gradients(grads_and_vars=grads_and_vars)
```

Listing 3.1: TensorFlow back-propagation

Here's an example of calling the builtin back-propagation algorithm and the update weights function in PyTorch. Note that both processes are executed with the same command:

```
params  # the weights and the biases of the variational autoencoder
lr  # learning rate for the weights and the biases updates
solver = optim.Adam(params, lr=lr)  # Adam Optimizer #
elbo_loss.backward()  # Backward #
solver.step()  # Update #
for p in params:  # Housekeeping #
    # initialize the parameter gradients of the next epoch as zeros
    p.grad.data.zero_()
```

Listing 3.2: PyTorch back-propagation

For instructions on how to apply update rules, see Appendix A, section A.11 Update Rules of Neural Network Weights.

NOTE: As we explained in the previous chapter, we want to maximize the variational lower bound (ELBO). However, in the implementations presented in this thesis, the ELBO loss is getting lower and lower in each epoch. The reason for that is because we use gradient descent instead of gradient ascent.

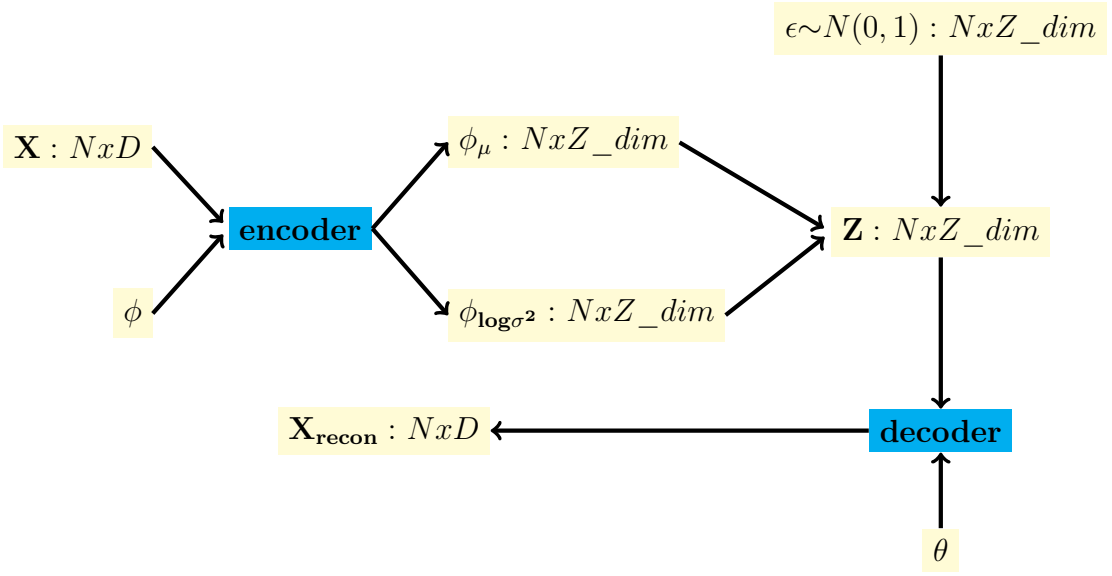
Chapter 4

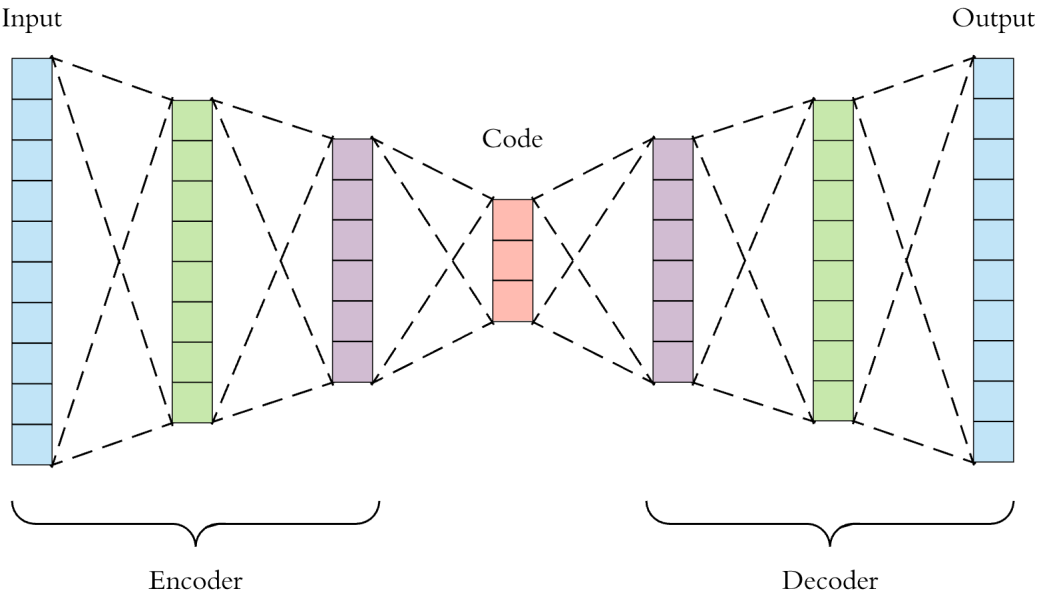
Variational Autoencoder Structure & Experiment Results

4.1 Variational Autoencoder Structure

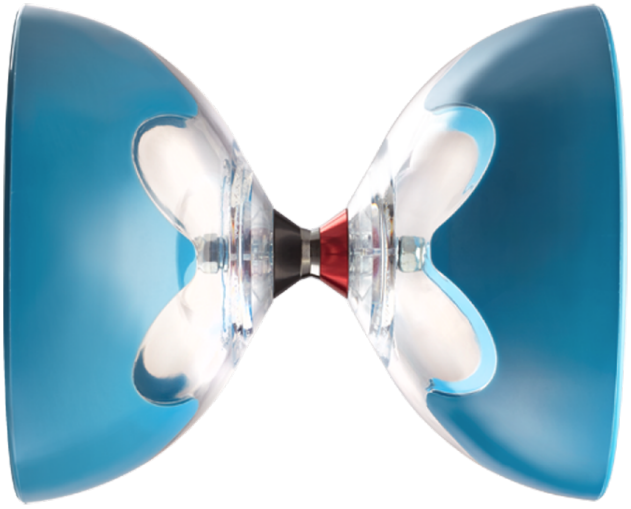
Usually, a variational autoencoder should need a great many epochs to be trained with. However, there are cases where too many epochs may give bad results, i.e. blurry images. When the lower bound gets too small, the VAE should stop training, because in the next epoch or iteration, the lower bound will eventually become very large. In an autoencoder, the number of neurons in the encoder, which we denote as M_1, must be equal to the number of neurons in the decoder, which we denote M_2.

The graph below demonstrates the steps that the variational autoencoder follows, to construct the artificial data, $\mathbf{X_{recon}}$, from the original data $\mathbf{X}$. We can examine how the original data, $\mathbf{X}$ are transformed into the latent data, $\mathbf{Z}$, which are a representation of $\mathbf{X}$ in a lower dimensionality $\mathbf{Z_{dim}}$:





(a) Autoencoder structure



(b) A diabolo juggling prop

FIGURE 4.1: The autoencoder is also called Diabolo network due to its structure's look. [1] (Rumelhart et al., 1986a; Bourlard & Kamp, 1988; Hinton & Zemel, 1994; Schwenk & Milgram, 1995; Japkowicz, Hanson, & Gluck, 2000, Yoshua Bengio 2009)

4.2 The TensorBoard

For programming implementations of the variational autoencoder in TensorFlow, PyTorch and Keras see Appendix C. Open a terminal (in Unix/Linux) or a command prompt (in Windows) and run the following command:

```
tensorboard --logdir = "path_to_the_graph"
```

Listing 4.1: TensorBoard command

Then, a message should appear that refers to the following URL: http://localhost:6006. Open a browser (e.g. Firefox) and browse to that URL. TensorFlow now runs as a web app on port 6006, by default.

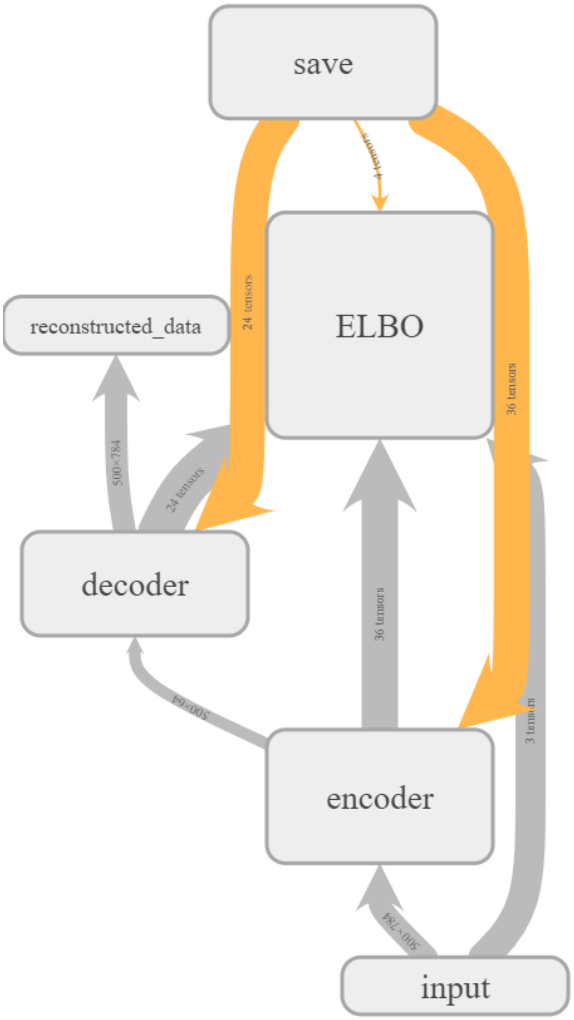


Figure 4.2: This is a visualization of the TensorFlow implementation, with TensorBoard.

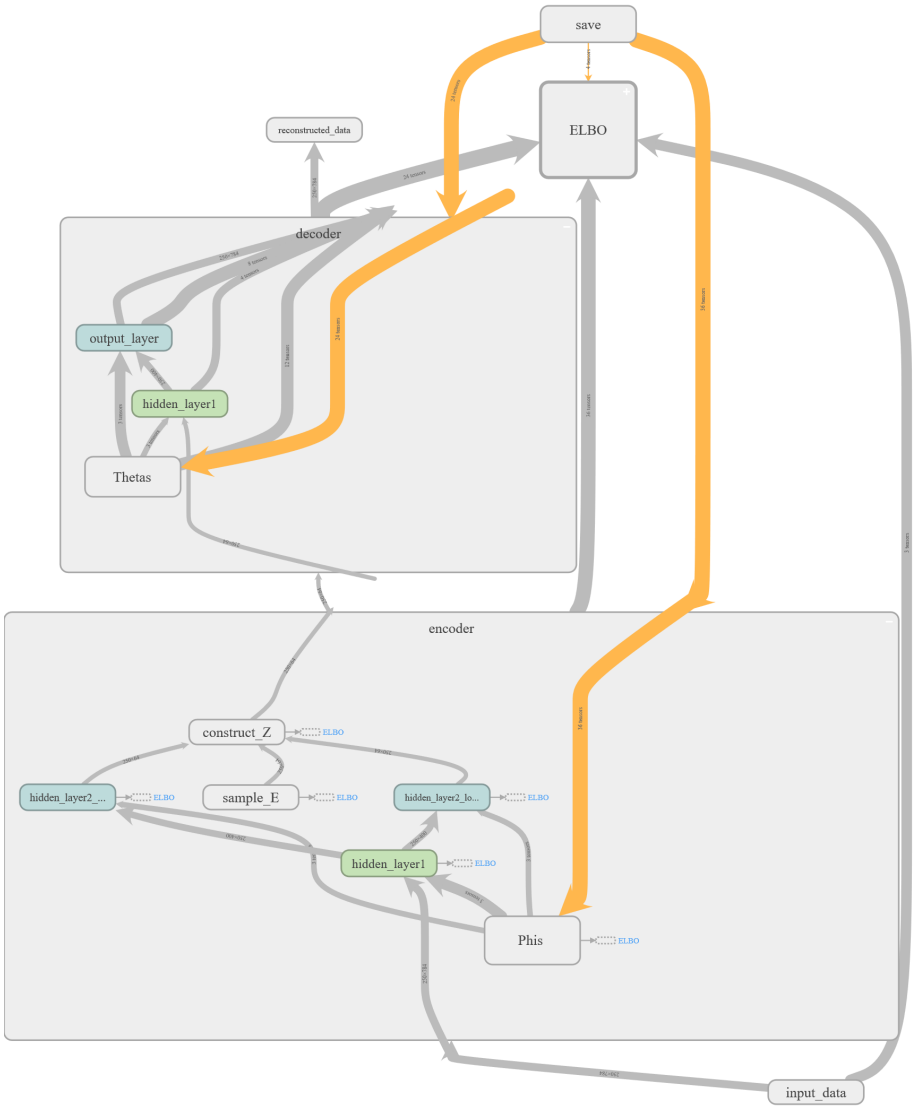


FIGURE 4.3: This is an enlarged visualization of the TensorFlow implementation, with TensorBoard.

NOTE: The port "6006" on the URL `http://localhost:6006` is "goog" upside down.

4.3 Datasets

Here is a summary of all the datasets used in this thesis:

Dataset	# TRAIN	# TEST	# VALIDATION
MNIST [5]	55000	10000	5000
Binarized MNIST	50000	10000	5000
CIFAR-10 [6]	50000	10000	-
OMNIGLOT	390 [En] / 360 [Gr]	130 [En] / 120 [Gr]	-
Cropped YALE Faces [7]	2442	-	-
ORL Face Database	400	-	-
MovieLens 100k	90570	9430	-

TABLE 4.1: Datasets details.

Dataset	# Classes	Dimensions	Type of Values
MNIST [5]	10	28x28 pixels	*real values in* $[0, 1]$
Binarized MNIST	10	28x28 pixels	$\{0, 1\}$
CIFAR-10 [6]	10	32x32x3 [RGB] / 32x32x1 [Grayscaled] pixels	*real values in* $[0, 1]$
OMNIGLOT	26 [En] / 24 [Gr]	32x32 pixels	*real values in* $[0, 1]$
Cropped YALE Faces [7]	38	168x192 pixels	*real values in* $[0, 255]$
ORL Face Database	40	92x112 pixels	*real values in* $[0, 255]$
MovieLens 100k	943 users	1682 movies	$\{1, 2, 3, 4, 5\}$

TABLE 4.2: Datasets number of classes, dimensions & type of values.

Dataset	URL
MNIST [5]	http://yann.lecun.com/exdb/mnist
Binarized MNIST	https://github.com/yburda/iwae/tree/master/datasets/BinaryMNIST
CIFAR-10 [6]	https://www.cs.toronto.edu/~kriz/cifar.html
OMNIGLOT	https://github.com/yburda/iwae/tree/master/datasets/OMNIGLOT
Cropped YALE Faces [7]	http://vision.ucsd.edu/extyaleb/CroppedYaleBZip/CroppedYale.zip
ORL Face Database	http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html
MovieLens 100k	https://grouplens.org/datasets/movielens

TABLE 4.3: Datasets links.

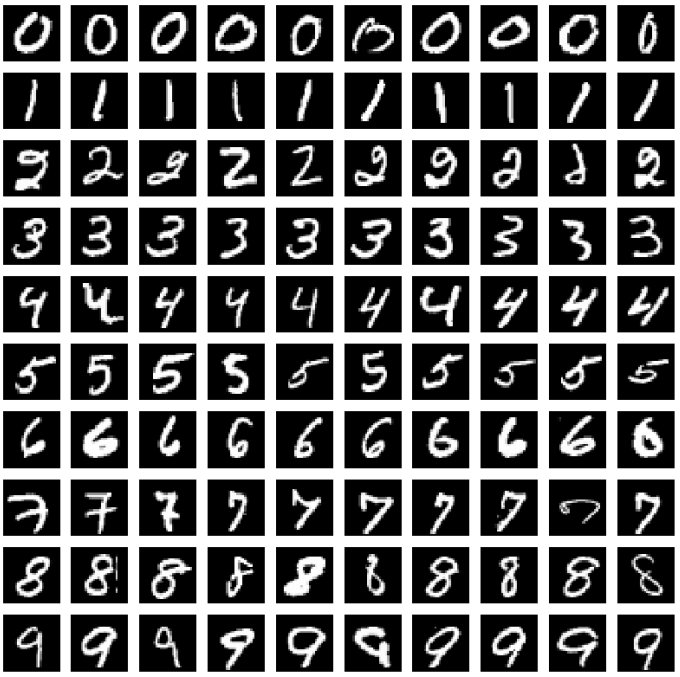
NOTES:

- The link for the YALE faces dataset may not be valid.
- ORL Face Database dataset has very few examples and therefore the results are very inaccurate. We do not recommend this dataset in our experiments. However, the dataset is promised to do well only with the K-NN missing values completion algorithm.
- All datasets that take values in the interval $[0, 255]$ are normalized to the interval $[0, 1]$, for easier calculations in the train process.

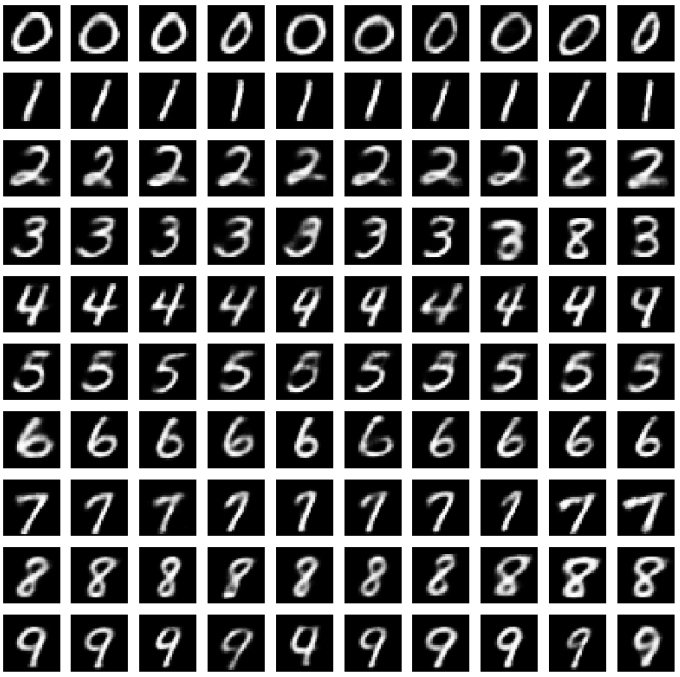
Christos Kormaris

4.4 Experiment Results

Here are the results of the VAE algorithm, using **TensorFlow**, on the **MNIST** dataset:

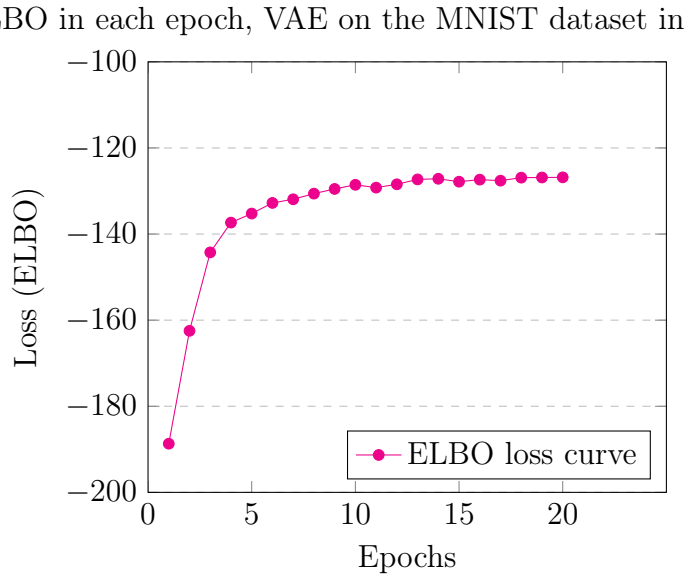


(a) Original Data



(b) VAE Epoch 20

FIGURE 4.4: MNIST VAE in TensorFlow.
root mean squared error: 0.142598

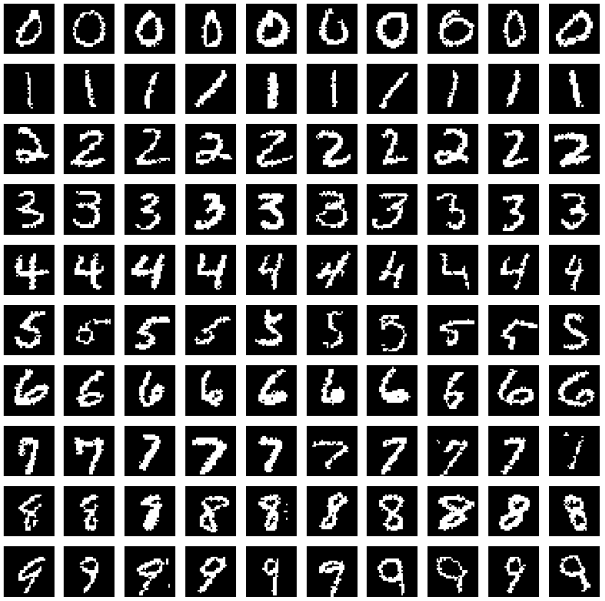


Metric	Value
last epoch loss (ELBO)	-126.83
root mean squared error (RMSE)	0.17121448655010216
mean absolute error (MAE)	0.07299246278044173

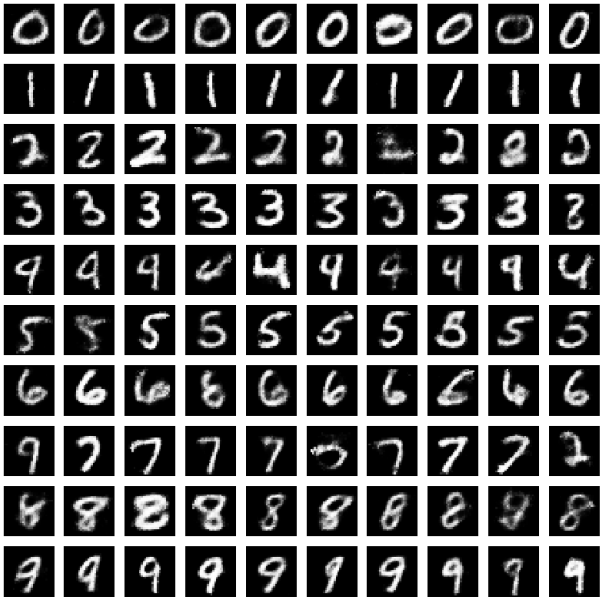
TABLE 4.4: VAE on the MNIST dataset in TensorFlow.

As explained on Chapter 3, Section 2, the ELBO is being minimized on each epoch instead of being maximized, because of using gradient descent instead of gradient ascent. We can conclude that MAE is a better metric than RMSE, because RMSE is less than 1. That means that its square (i.e. the MSE) must be smaller than the RMSE and also less than 1.

Here are the results of the VAE algorithm, using **TensorFlow**, on the **Binarized MNIST** dataset:



(a) Original Data



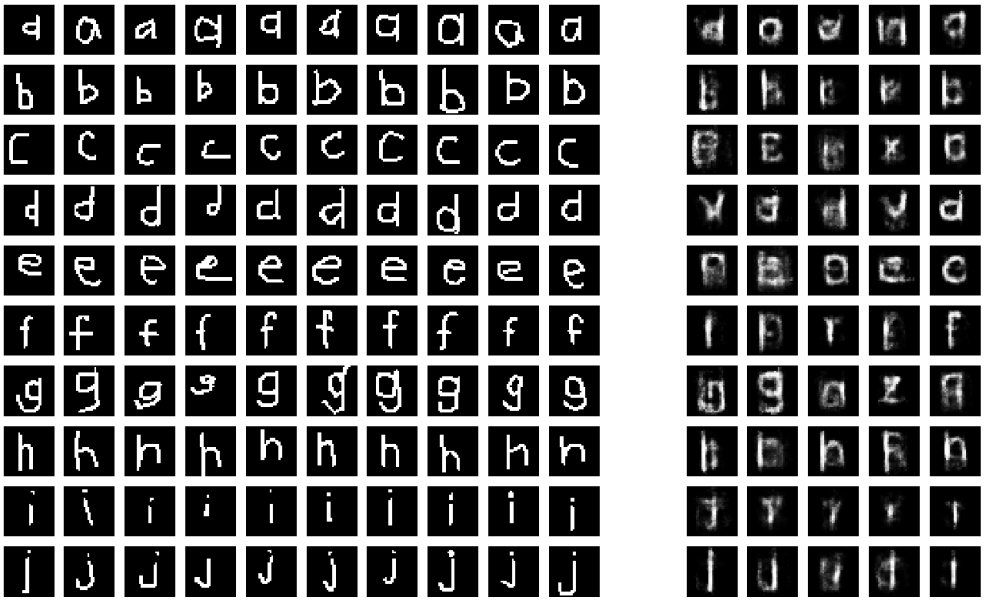
(b) VAE Epoch 50

FIGURE 4.5: MNIST VAE in TensorFlow.

Metric	Value
last epoch loss (ELBO)	-109.82
root mean squared error (RMSE)	0.186880795395
mean absolute error (MAE)	0.0685835682327

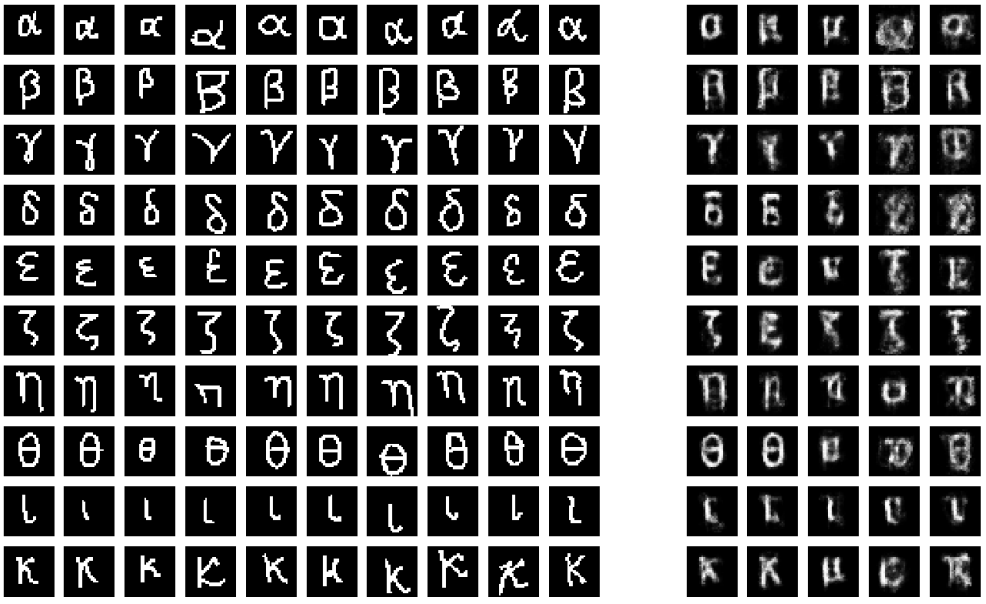
TABLE 4.5: VAE on the Binarized MNIST dataset in TensorFlow.

Here are the results of the VAE algorithm, using **PyTorch**, on the **OMNIGLOT** dataset:



(a) Original Data (b) VAE Epoch 100

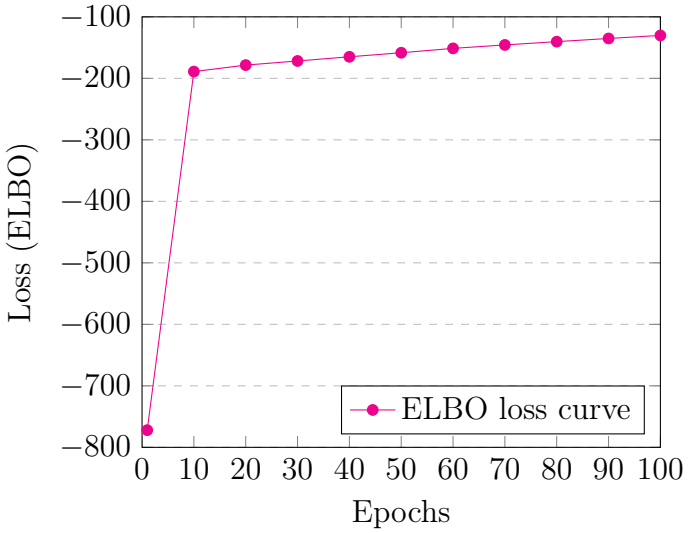
FIGURE 4.6: OMNIGLOT English alphabet, characters 1-10, original and reconstructed.



(a) Original Data (b) VAE Epoch 100

FIGURE 4.7: OMNIGLOT Greek alphabet, characters 1-10, original and reconstructed.

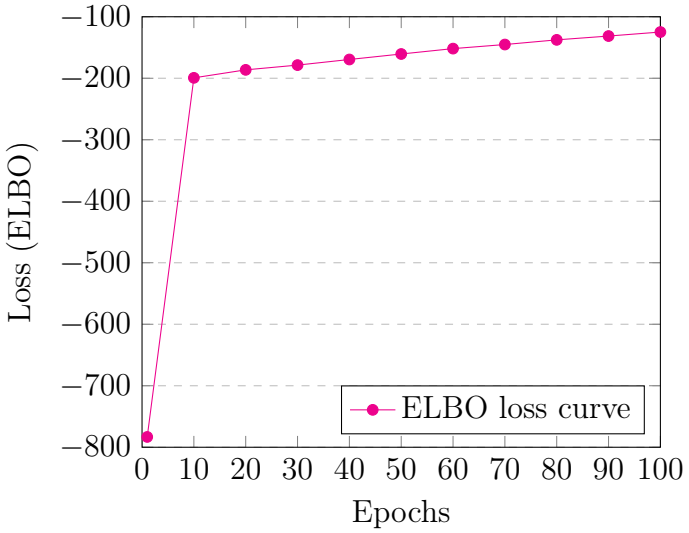
ELBO in each epoch, VAE on the OMNIGLOT English dataset, in PyTorch



Metric	Value
last epoch loss (ELBO)	-130.22
root mean squared error (RMSE)	0.2134686176531402
mean absolute error (MAE)	0.09105986614619989

TABLE 4.6: VAE on the OMNIGLOT English dataset in PyTorch.

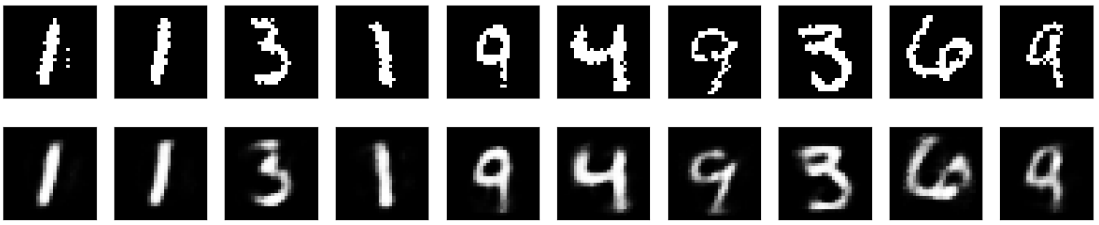
ELBO in each epoch, VAE on the OMNIGLOT Greek dataset, in PyTorch



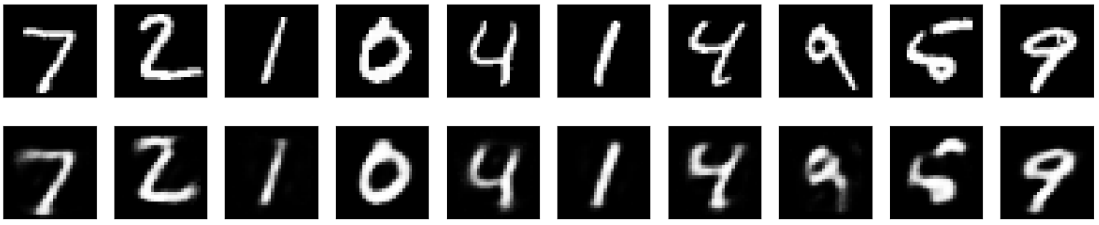
Metric	Value
last epoch loss (ELBO)	-124.85
root mean squared error (RMSE)	0.20157381435292054
mean absolute error (MAE)	0.08379960438030741

TABLE 4.7: VAE on the OMNIGLOT Greek dataset in PyTorch.

Here are the results of the VAE algorithm, using **Keras**, on the **Binarized MNIST**, **MNIST**, **CIFAR-10** & **OMNIGLOT** datasets:

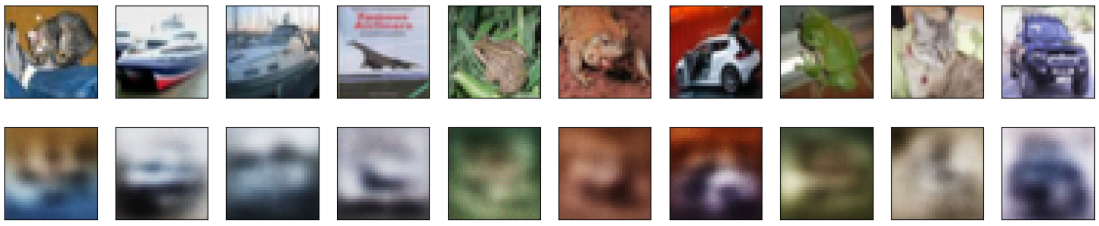


(a) Binarized MNIST

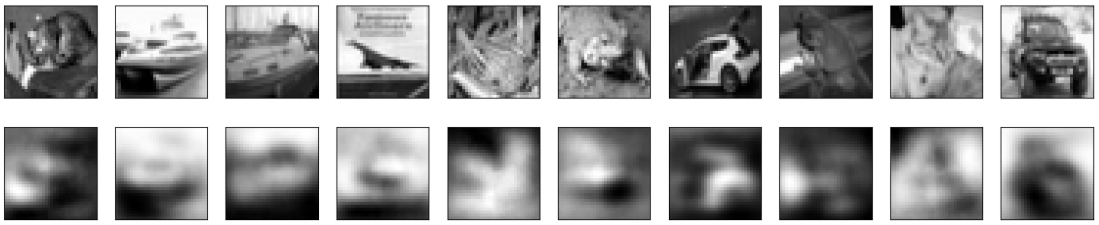


(b) MNIST

FIGURE 4.8: Binarized MNIST & (Real-valued) MNIST digits, original and reconstructed.

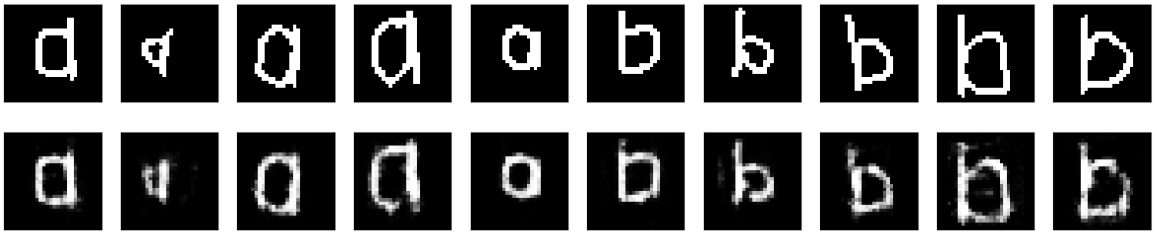


(a) CIFAR-10 RGB

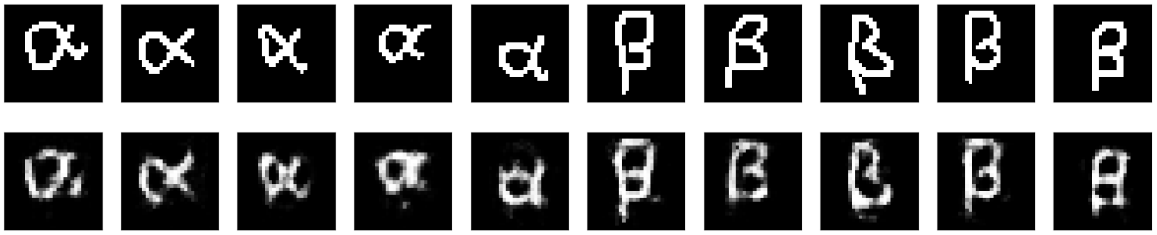


(b) CIFAR-10 Grayscale

FIGURE 4.9: CIFAR-10 images, RGB & grayscale, original and reconstructed.



(a) OMNIGLOT English



(b) OMNIGLOT Greek

FIGURE 4.10: OMNIGLOT English & Greek characters, original and reconstructed.

NOTE: The images of the CIFAR-10 RGB dataset have pixels of three channels (red, green and blue) taking real values, thus the VAE is not expected to have good results on them.

Metric	Value
last epoch loss (ELBO)	0.0635
root mean squared error (RMSE)	0.1361456340713906
mean absolute error (MAE)	0.03993093576275391

TABLE 4.8: VAE on the Binarized MNIST dataset, in Keras.

Metric	Value
last epoch loss (ELBO)	0.0040
root mean squared error (RMSE)	0.00102286
mean absolute error (MAE)	0.00063948194

TABLE 4.9: VAE on the MNIST dataset, in Keras.

Metric	Value
last epoch loss (ELBO)	0.6198
root mean squared error (RMSE)	0.17352526
mean absolute error (MAE)	0.1368697

TABLE 4.10: VAE on the CIFAR-10 RGB dataset, in Keras.

Metric	Value
last epoch loss (ELBO)	0.6166
root mean squared error (RMSE)	0.14963366
mean absolute error (MAE)	0.11517575

TABLE 4.11: VAE on the CIFAR-10 Grayscale dataset, in Keras.

Metric	Value
last epoch loss (ELBO)	0.1764
root mean squared error (RMSE)	0.23234108227525616
mean absolute error (MAE)	0.11663868188603484

TABLE 4.12: VAE on the OMNIGLOT English dataset, in Keras.

Metric	Value
last epoch loss (ELBO)	0.1840
root mean squared error (RMSE)	0.23350568189372273
mean absolute error (MAE)	0.1205574349168227

TABLE 4.13: VAE on the OMNIGLOT Greek dataset, in Keras.

OBSERVATIONS:

- The ELBO losses estimated by the Keras VAE are significantly lower than the losses delivered by the TensorFlow and PyTorch implementations.
- The images reconstructed by the VAE on the MNIST dataset are closer to the original data than ones reconstructed by the VAE on the Binarized MNIST dataset. The error metric (RMSE, MAE) as well as the ELBO for the MNIST dataset are close to 0, which means that the VAE has possibly overfit the training data. This outcome was not expected since a VAE should behave better on binary data. The explanation should be sought in the Keras backend and the algorithms it uses to estimate the loss function.
- The results on the CIFAR-10 dataset are better for the Grayscaled images rather than the RGB images.
- The results on the OMNIGLOT dataset are slightly better for the English language rather than the Greek language, mainly because of shortage in examples. There are 520 images of English characters and 480 images of Greek characters.

Chapter 5

Missing Values Completion Algorithms & Variational Autoencoders

5.1 A Simple Missing Values Completion Algorithm Using K-NN Collaborative Filtering (CF)

The traditional algorithm for a recommendation system is collaborative filtering. A very usual application for this method is the prediction of a user's rating for movies he has not rated yet, based on the similarity between his ratings on other movies and the ratings of other users. The cosine similarity metric may be used to estimate the most similar users. In this algorithm, we' ll use Euclidean distances. The **MovieLens** dataset is the most popular one among others, for cases like this. The **MNIST** dataset can also be a suitable application for the algorithm with small modifications.

We can apply the collaborative filtering method on the **MNIST** dataset, using a variation of K-NN (K Nearest Neighbors) algorithm for regression, as follows:

1. Store the MNIST dataset **X_train** of digit images in memory. The dimensions of the train data, **X_train** are NxD, where N is the number of train images and D is the number of pixels in each image. For the MNIST dataset: $D = 784$. Also, store the test data on a variable, **X_test**.
2. Modify the train and the test data by introducing missing values. We have thought of two ways to choose to construct missing values. One way is to to replace the right, left, top and bottom part pixels or no pixels at all in each image, with values that indicate that the pixels are missing. A second way is to select pixels at random from each image and replace them with the missing value. If the pixels take values in the range $[0, 1]$ (1 for being a black pixel and 0 for being a white pixel), a good choice for the value to represent the missing data would be *0.5*, which corresponds to a gray color pixel.
3. For the test examples, select the **K** closest data (images) that the algorithm will take into consideration. The one nearest neighbor corresponds to **K=1**.
4. For each test example calculate the Euclidean distances to every train example, find the **K** closest ones and store the indices of the K closest train examples in a variable.

However, prefer not to find closest neighbors with missing pixels. To surpass this obstacle, we propose the following procedure:
Make the train data pixels missing, where the current test instance has missing pixels. Store the result in the variable called **X_train_common**. Thus, the difference **X_train_common - X_test_common**, in the indices where the test instance pixels are missing, will be 0. The variable **X_test_common** will be built next.

```
X_test_i  # the current test instance
s = np.where(X_test_i == missing_value)
X_train_common = np.array(X_train)
X_train_common[:, s] = missing_value
```

LISTING 5.1: X_train_common

Repeat the test instance Ntrain times and make the test data values equal to the mean of all train examples, where the current train instance every time has missing values. Store the result on the variable called **X_test_common**.

```
Ntrain  # number of train examples
D  # number of pixels (aka dimensions)
X_test_i  # the current test instance
X_test_common = np.zeros((Ntrain, D))
mean_values = np.mean(X_train, axis=0)  # 1 x D array
for k in range(Ntrain):
    X_test_common[k, :] = X_test_i
    s = np.where(X_train[k, :] == missing_value)
    if len(s[0]) != 0:
        X_test_common[k, s] = mean_values[s]
```

LISTING 5.2: X_test_common

5. Extract the data (images) whose indices correspond to the K closest train examples indices, from the train data **X_train** and store the result to a variable.
6. There are two solutions proposed to proceed further.
   - We can use weight coefficients, depending on the distance between the test example and the K closest train examples. The closest example gets the biggest weight value. Each pixel of the *k-th* train example is being multiplied with its corresponding weight w_k. The weight values are assigned as follows:

$$\mathbf{w_k} = \mathbf{softmax}(-\mathbf{d_k}) = \frac{\mathbf{e^{-d_k}}}{\sum_{\mathbf{i=1}}^{\mathbf{K}} \mathbf{e^{-d_i}}},$$

$$\text{where: } w_1 \geq w_2 \geq w_3 \geq ... \geq w_K,$$

$$\text{and } w_1 + w_2 + w_3 + ... + w_K = \sum_{j=1}^{K} w_j = \sum_{j=1}^{K} \frac{e^{-d_j}}{\sum_{i=1}^{K} e^{-d_i}} = 1$$

where $d_k$denotes the distance from the $k-th$closest train example

- Alternatively, we can take the sum across all rows of the resulting extracted data variable (each row representing a train example). After having calculated the sum of the K closest train examples on each pixel, we can divide the vector of the sums with **K**. The result will be an average prediction on the values of each pixel of the current test example. This method would be the same as the previous one, if we had assigned to the weights equal values: $w_k = \frac{1}{K}$, for each k.

7. Assign the values of the predicted values vector, only to the corresponding indices with missing values of the test example.
8. Repeat steps 3-7 for all test examples.
9. Calculate the root mean squared error (RMSE) between the predicted test data and the real test data, in order to estimate the efficiency of the algorithm, in the following manner:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{N}\sum_{j=1}^{D}\left(X_{i,j} - \tilde{X}_{i,j}\right)^2}$$

where X are the original test data and $\tilde{X}$ are the test data with the predicted missing values. The closer this value is to 0, the more efficient the algorithm is.

The overall procedure of the pixels prediction in matrix notation is represented by the following formula:

$$closest\_data_{ij} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & \dots & p_{1D} \\ p_{21} & p_{22} & p_{23} & \dots & p_{2D} \\ & \dots\dots\dots\dots & & & \\ p_{k1} & p_{k2} & p_{k3} & \dots & p_{kD} \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ . \\ w_K \end{bmatrix} = \begin{bmatrix} p_{11}\cdot w_1 & p_{12}\cdot w_1 & \dots & p_{1D}\cdot w_1 \\ p_{21}\cdot w_2 & p_{22}\cdot w_2 & \dots & p_{2D}\cdot w_2 \\ & \dots\dots\dots\dots\dots & & \\ p_{K1}\cdot w_K & p_{K2}\cdot w_K & \dots & p_{KD\cdot w_K} \end{bmatrix}$$

*where **p** stands for pixel, **K** is the number of the closest train data to take into consideration and D is the number of pixels.* Then, the vector of the predicted pixels of the current text example will be the following:

$$predicted\_pixels = \sum_{i=1}^{K} closest\_data_{ij}$$

From this vector we assign the values only to the missing pixels of the test example.
To sum up, the K-NN algorithm is mostly used for classification. However, we have hereby shown how we can alter the algorithm to work for regression purposes as well, such as the prediction of real values for pixels.

Christos Kormaris

5.2 A Proposed Missing Values Completion Algorithm Using Variational Autoencoders

We can use variational autoencoders to predict missing data on training images. The main thought is to modify the VAE algorithm to keep intact the original non-missing values and change only the parts with missing values, on each iteration.

First, we construct the dataset of missing values, $X\_train\_missing$, from the original dataset X_train. We leave $\frac{1}{5}$ of the dataset as is. From the remaining $\frac{4}{5}$ data, we replace the half part of each image, with a value we'll call "missing value". The "missing value" could be set equal to the mean of the interval $[lowest\_value, highest\_value]$, where $lowest\_value$ and $highest\_value$ are the lowest and highest values that appear on the images of the dataset, respectively. For instance, this interval in the MNIST dataset is $[0, 1]$, thus the "missing value" is set equal to 0.5. The part that we choose to erase and replace with missing values could be either the top, bottom, left or right half of the image. Another option is to pick pixels at random, specifically half from each image and replace them with missing values. Now that we have constructed the dataset with missing values, $X\_train\_missing$, we need to construct a matrix with binary values (0 or 1), which will store the information of which parts of the original images we replaced with missing values. We'll call this matrix X_train_masked. If a pixel in an image did not get replaced, the corresponding pixel in the X_train_masked matrix will be 1. If a pixel in an image did get replaced by the "missing value", the corresponding pixel in the X_train_masked matrix will be 0. Furthermore, we need to define the matrix that will contain the data with the predicted values of the missing pixels. We'll call this matrix X_filled. We initialize the matrix X_filled to be the same as $X\_train\_missing$.

Finally, we run a modified version of the VAE algorithm. Like we mentioned earlier, on each iteration, we must replace the pixels only where the missing values existed, with the predicted ones. At the same time we must maintain intact the pixels with non-missing values. For that purpose, the matrix X_train_masked will come in handy.

Here's the pseudocode for the whole process we described above:

```
for epoch = 0 to epochs - 1
    iterations = N / batch_size

    for i = 0 to iterations - 1
        start_index = i * batch_size
        end_index = (i + 1) * batch_size

        // fetch the batch data, labels and masked batch data
        batch_data = X_filled(start_index:end_index, :)
        batch_labels = y_train(start_index:end_index)
        masked_batch_data = X_train_masked(start_index:end_index, :)

        // train the batch data using the VAE process
        cur_samples = train(batch_data, params)

        // ".*" denotes element-wise multiplication
        // The "cur_samples" will take values from the "batch_data"
        // where the pixels are observed (with masked values=1)
        // and will keep intact its values from the VAE training
        // where the pixels are missing (with masked values=0).
        cur_samples = masked_batch_data .* batch_data +
                      (1 - masked_batch_data) .* cur_samples
        X_filled(start_index:end_index, :) = cur_samples
```

LISTING 5.3: pseudocode for the VAE missing values completion algorithm

5.3 Experiment Results on K-NN Missing Values Completion Algorithms & VAE Missing Values Completion Algorithms

After testing the collaborative filtering algorithm on the **MNIST** dataset with missing values, for various values of K, along with the VAE missing values completion algorithm in PyTorch, we have ended up with the following results. As we are about to see, some test images of the digits *3*, *5*, *8* were reconstructed with the other half of images of the digits *3* or *5* or *5*. This mishap is somewhat logical, since the digits *3*, *5* and *8* resemble each other.

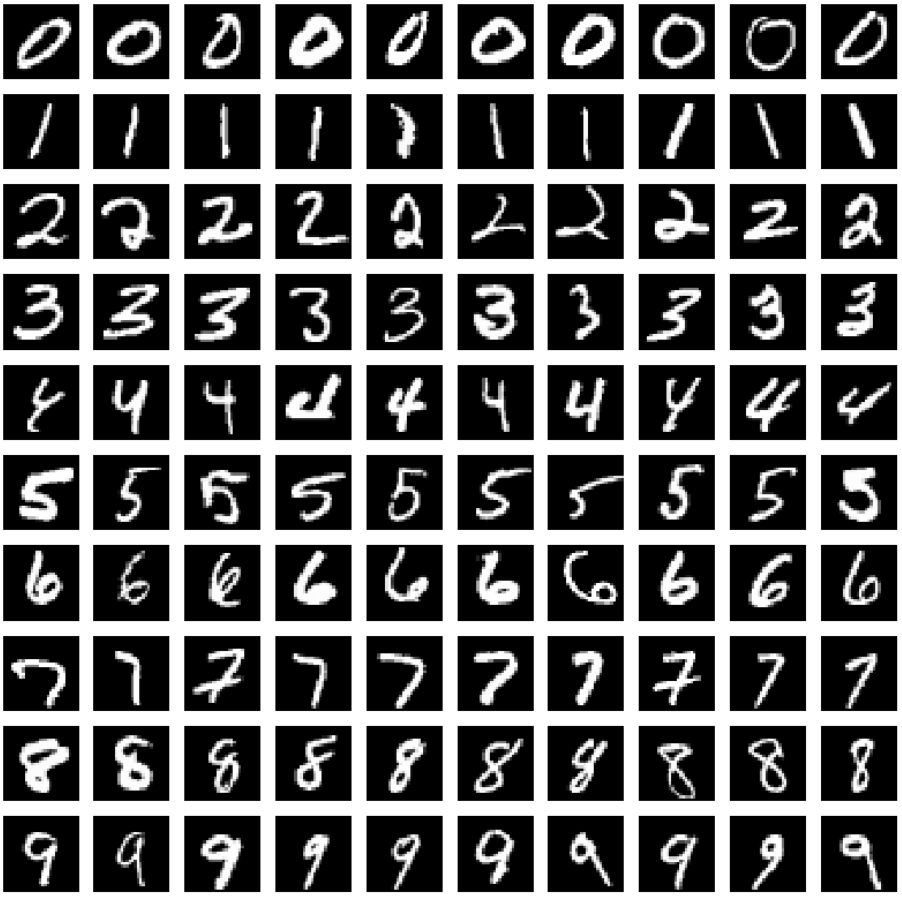
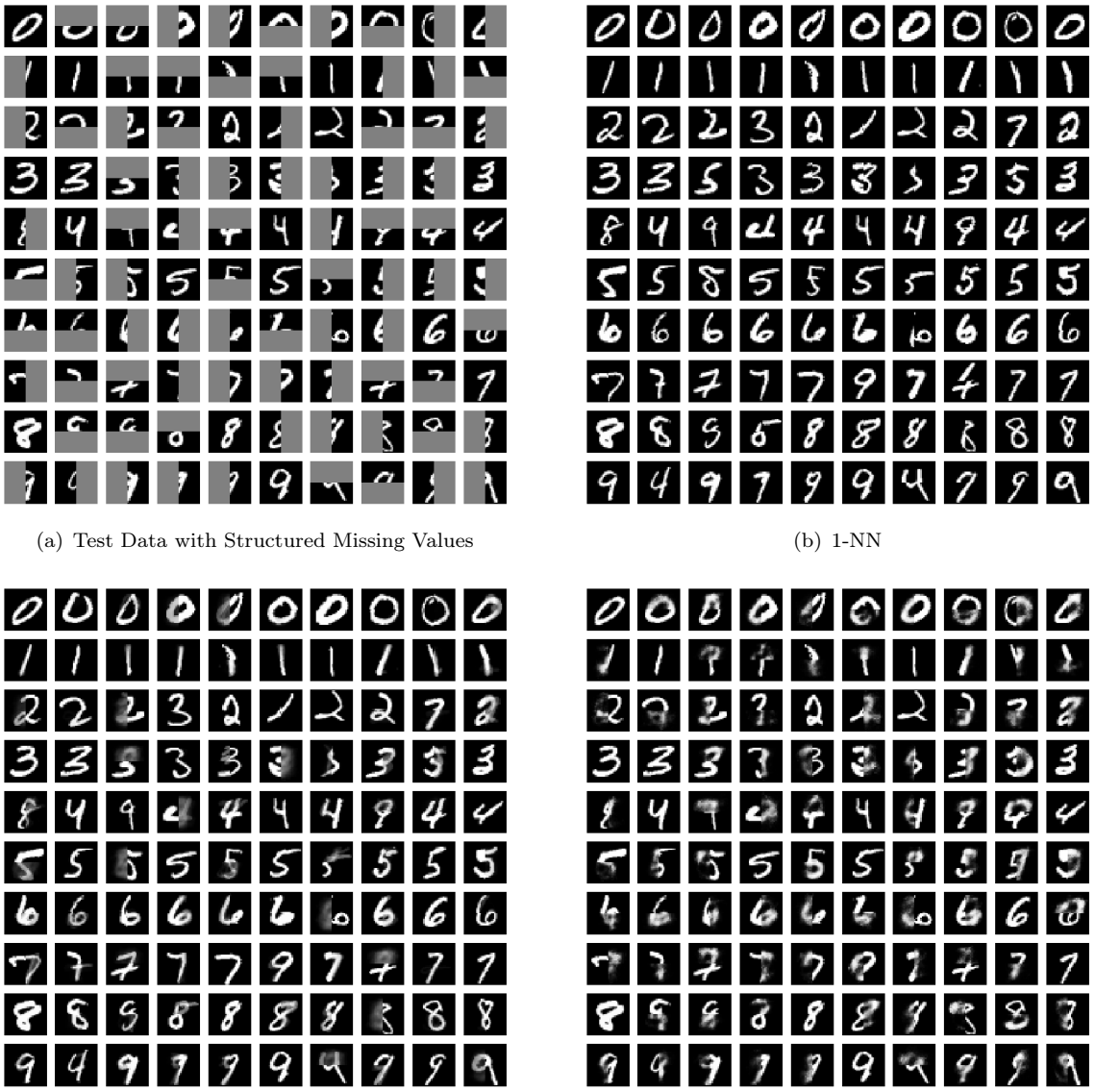


FIGURE 5.1: Original MNIST Test Data



(a) Test Data with Structured Missing Values

(b) 1-NN

(c) 100-NN

(d) VAE in PyTorch Epoch 200

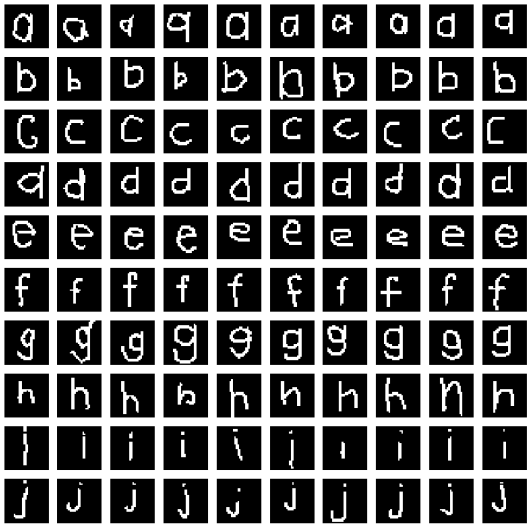
FIGURE 5.2: MNIST **1-NN**, **100-NN** & **VAE** Missing values algorithms.

The lowest root mean squared error (RMSE) and the lowest mean absolute error (MAE) was achieved for **K=100**, but the algorithm was the most time consuming compared to the others.

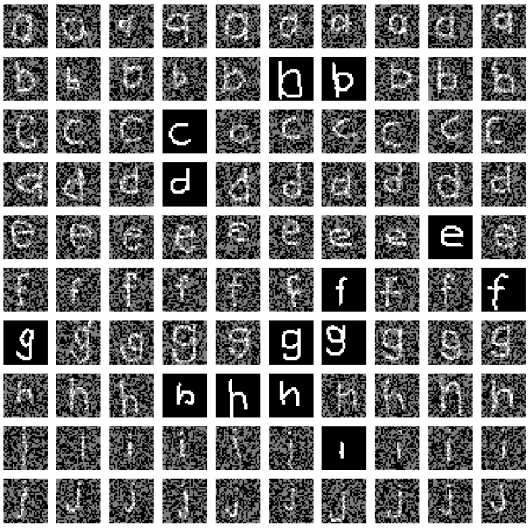
Method	RMSE	MAE	Time
1-NN	0.171569	0.0406822	190.26 sec
100-NN	0.148223	0.0396628	233.19 sec
VAE	0.168031	0.0499518	178.09 sec

TABLE 5.1: Missing values completion algorithms on the MNIST dataset.

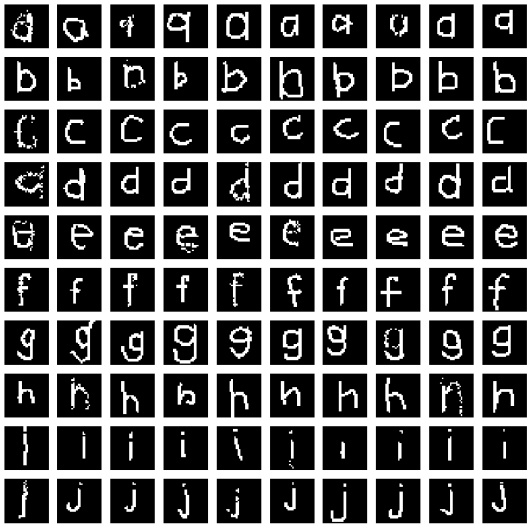
Here are the results of the Missing Values completion algorithms, using **PyTorch**, on the **OMNIGLOT** English dataset.



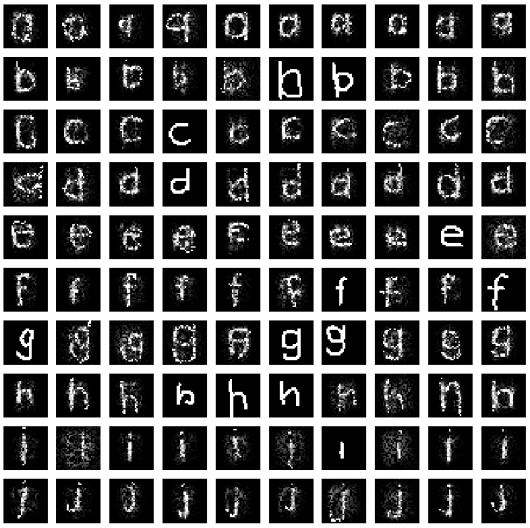
(a) Original Data



(b) Data with Missing at Random Values



(c) 1-NN



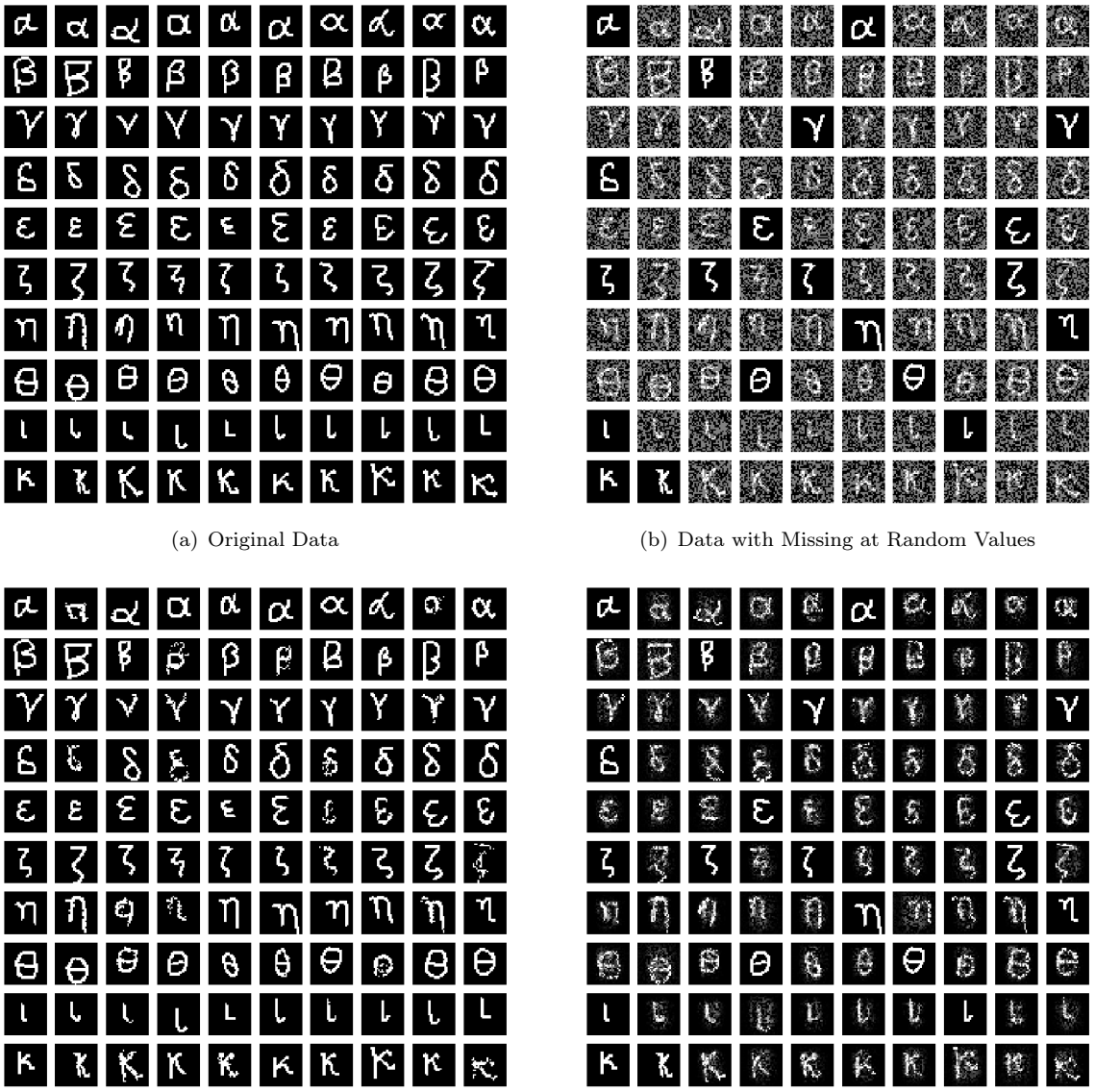
(d) VAE in PyTorch Epoch 100

FIGURE 5.3: **1-NN** & **VAE** Missing values algorithms on OMNIGLOT English alphabet, characters 1-10, original, missing at random and reconstructed.

Method	RMSE	MAE	Time
1-NN	0.0890025270986	0.00792144982993	8.64 sec
VAE	0.162737553068	0.0524327812139	245.66 sec

TABLE 5.2: Missing values completion algorithms on the OMNIGLOT English dataset.

Here are the results of the Missing Values completion algorithms, using **PyTorch**, on the **OMNIGLOT** Greek dataset.



(a) Original Data

(b) Data with Missing at Random Values

(c) 1-NN

(d) VAE in PyTorch Epoch 100

FIGURE 5.4: **1-NN** & **VAE** Missing values algorithm on OMNIGLOT Greek alphabet, characters 1-10, original, missing at random and reconstructed.

Method	RMSE	MAE	Time
1-NN	0.0846893853663	0.00717229199372	9.37 sec
VAE	0.166289735254	0.0536736838285	258.4 sec

TABLE 5.3: Missing values completion algorithms on the OMNIGLOT Greek dataset.

NOTE: The time metric of the K-NN algorithms shows the duration for the distances and predictions calculations, while the time metric of the VAE algorithms shows the durations for the VAE training loops.

5.4 K-NN vs VAE Missing Values Completion Algorithm on the MovieLens Dataset

We have run both K-NN and VAE in TensorFlow Missing Values completion algorithms on the MovieLens 100k dataset, **ua set** and compared the predicted ratings. The results are demonstrated below:

Metric	**Value**
root mean squared error (RMSE)	0.0803184360998
mean absolute error (MAE)	0.0209178842034
match percentage	91.2514516501 %

Method	**Mean Rating**
VAE in TensorFlow	1.60717332671
K-NN Missing Values algorithm	1.61095209334

TABLE 5.4: Comparison between 10-NN and VAE in TensorFlow missing values completion algorithms.

From this table, judging from the error metrics, we conclude that the predictions of the two algorithms for the ratings of the users are very close. In fact, we rounded up the ratings to the closest integer and we realized that the results are about identical 91%. Finally, the mean rating for the VAE algorithm in TensorFlow was about 1.61, whereas the mean rating for the K-NN missing values completion algorithm was about 1.6. The ratings are rounded to take integer values in the range $[1, 5]$ and the value that indicates that the user has not rated a movie is 0.

Chapter 6

Variational Autoencoders & Missing Values Completion Algorithms GUI

A graphical user interface (GUI) has been implemented for the project of this thesis, using Python 3 and the **Tkinter** library.

First, browse to the directory **"vaes_gui"** of the thesis project and install the Python dependency libraries, by typing:

```
pip install -r depencencies.txt
```

Listing 6.1: command for installing Python dependencies

To run the GUI from the terminal, type:

```
python vaes_gui.py
```

Listing 6.2: command for running the GUI

To create an executable file for the GUI (".exe"), which you can run anytime from a Windows environment, type:

```
pip install pyinstaller
pyinstaller vaes_gui.spec
```

Listing 6.3: command for creating an executable file for the GUI

Then, download all the datasets from the URLs in the file **"datasets_urls.txt"** and move them to the newly created **"dist"** folder. Inside, there should be a folder with the name **"vaes_gui"**, which contains the executable file **"vaes_gui.exe"**.

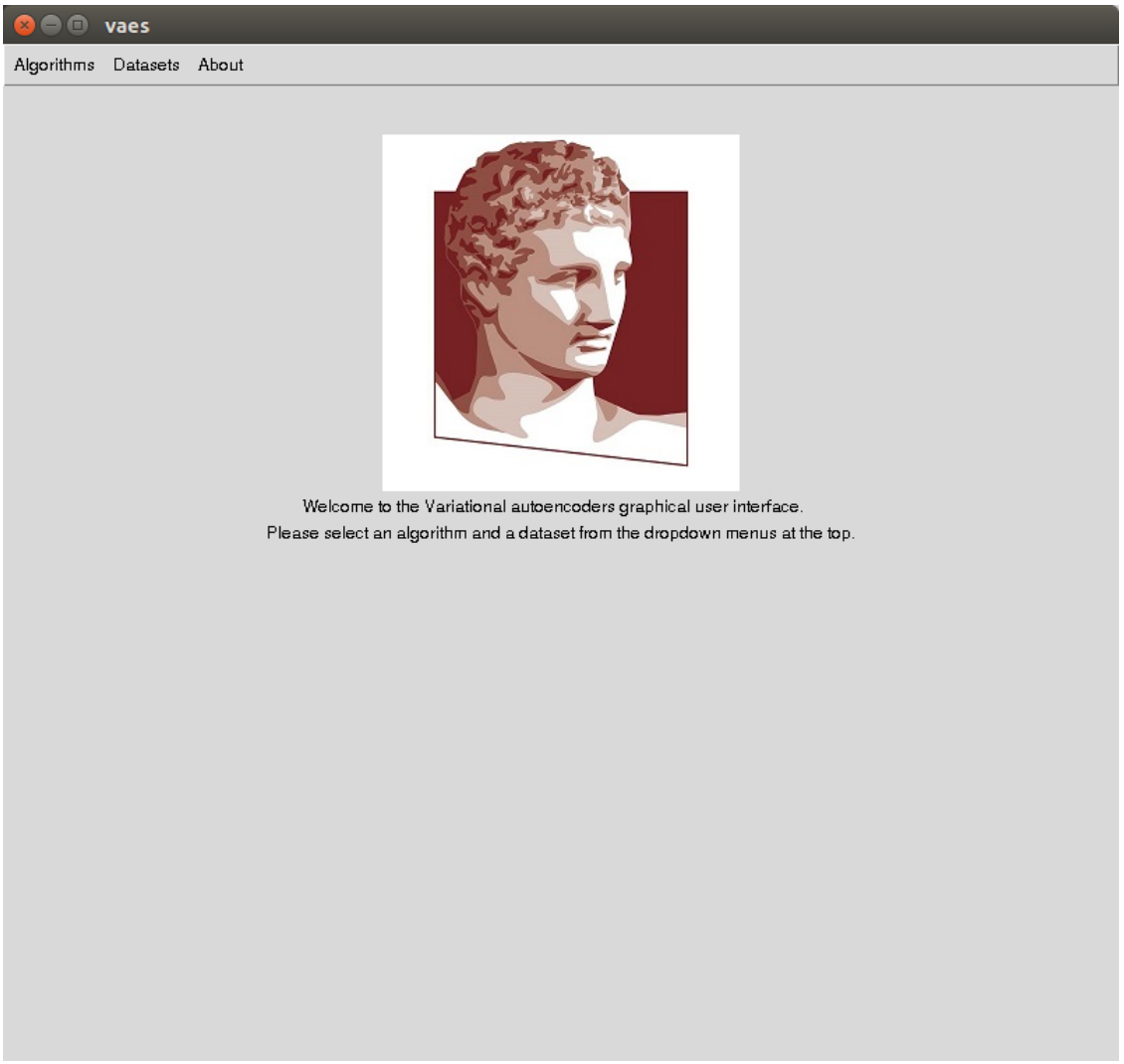
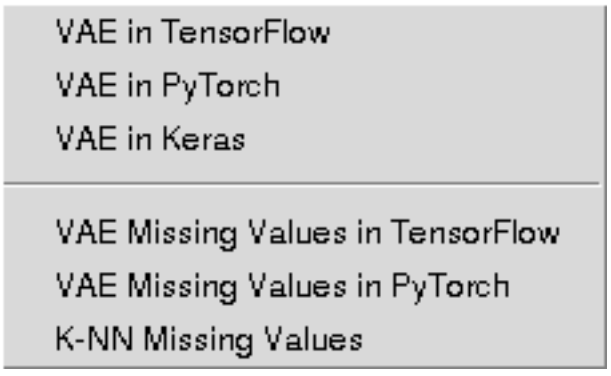
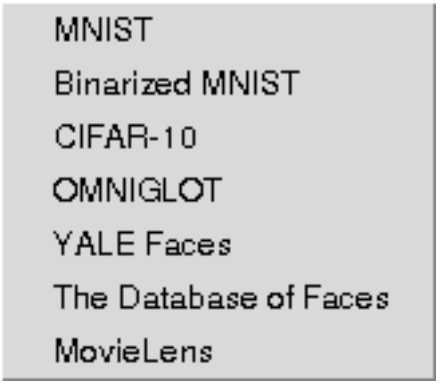


FIGURE 6.1: GUI Welcome page.



(a) GUI Algorithms dropdown menu.



(b) GUI Datasets dropdown menu.

FIGURE 6.2: Dropdown menus.

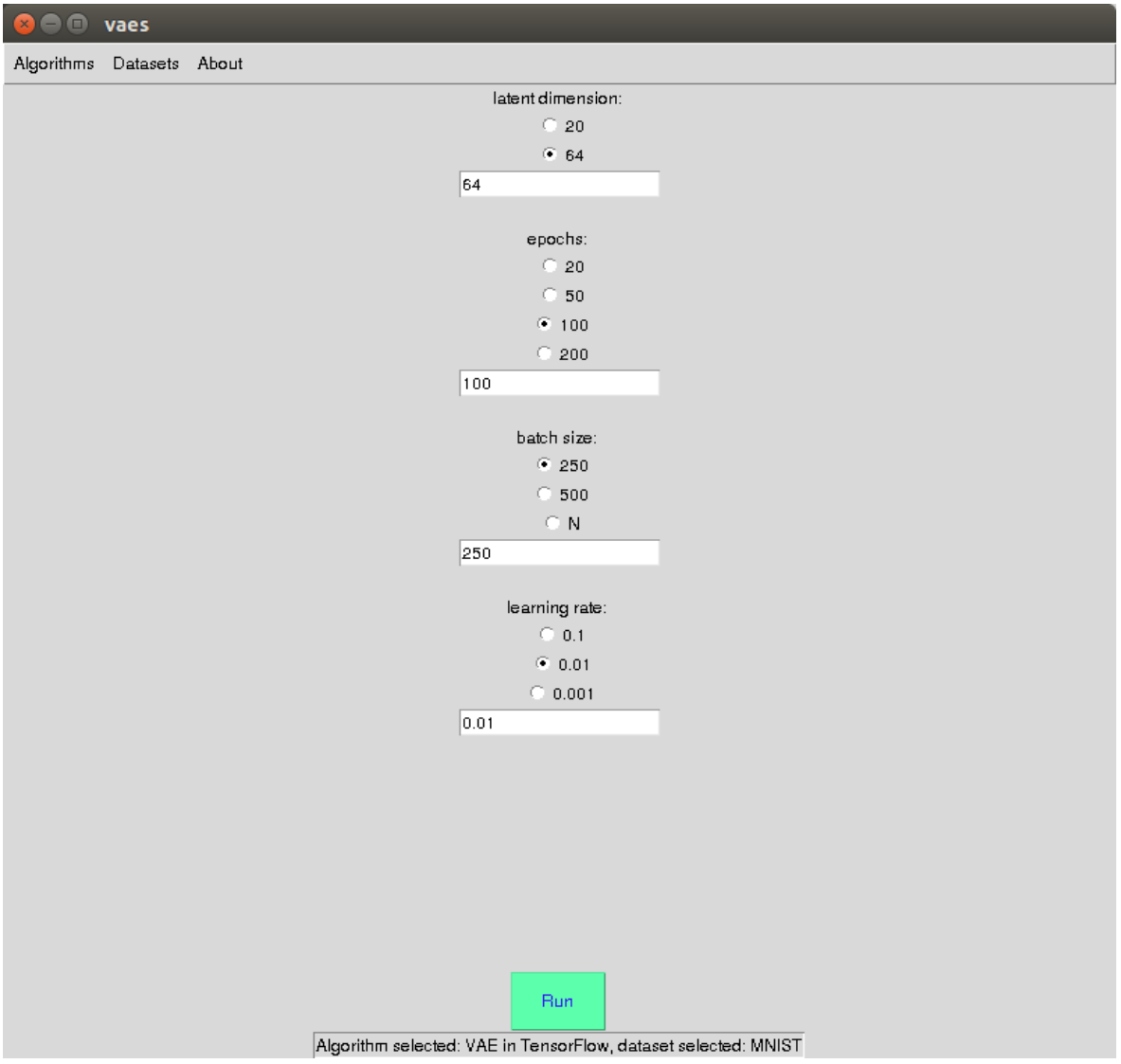


FIGURE 6.3: GUI VAE in TensorFlow, MNIST dataset.

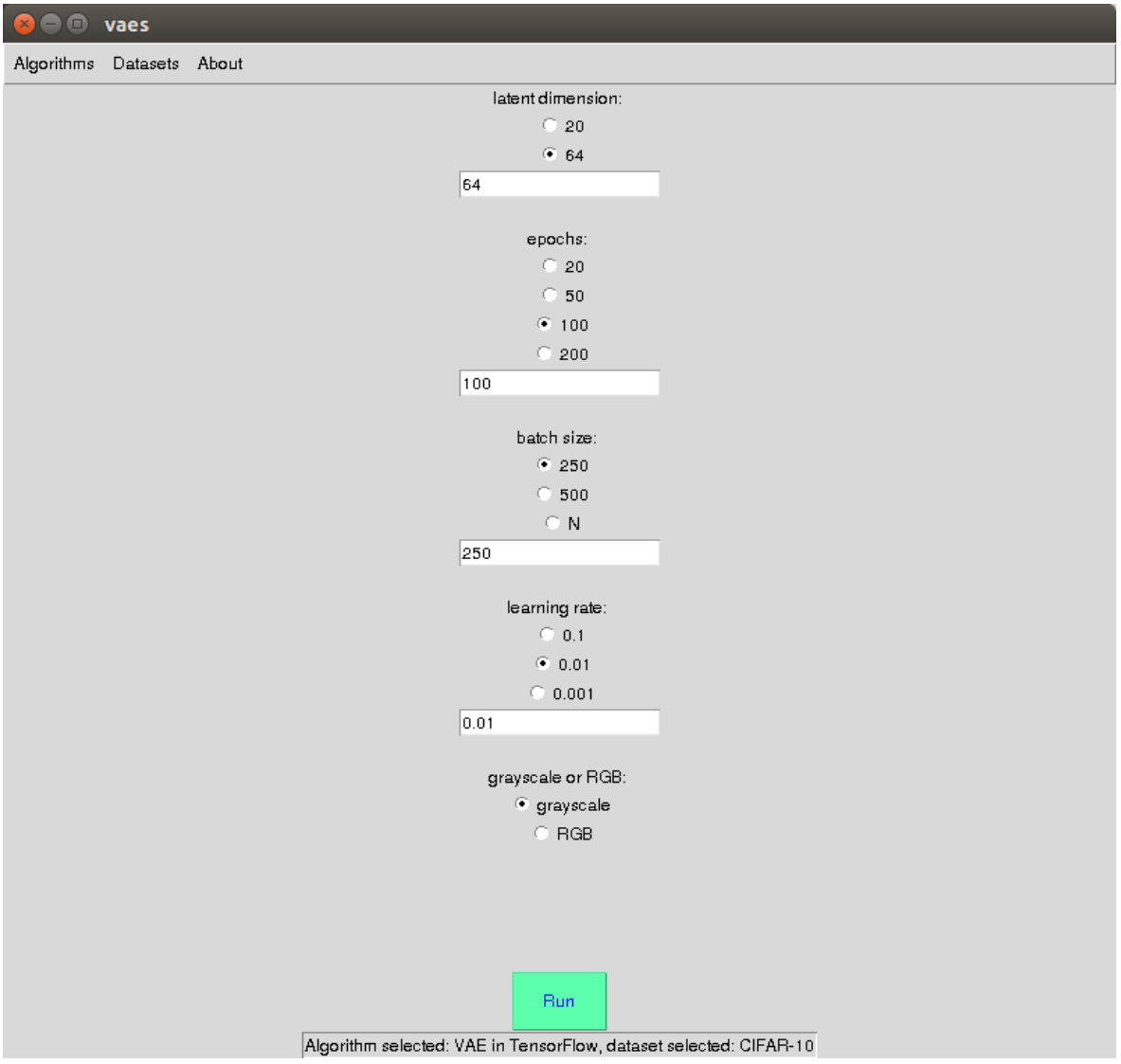


FIGURE 6.4: GUI VAE in TensorFlow, CIFAR-10 dataset.

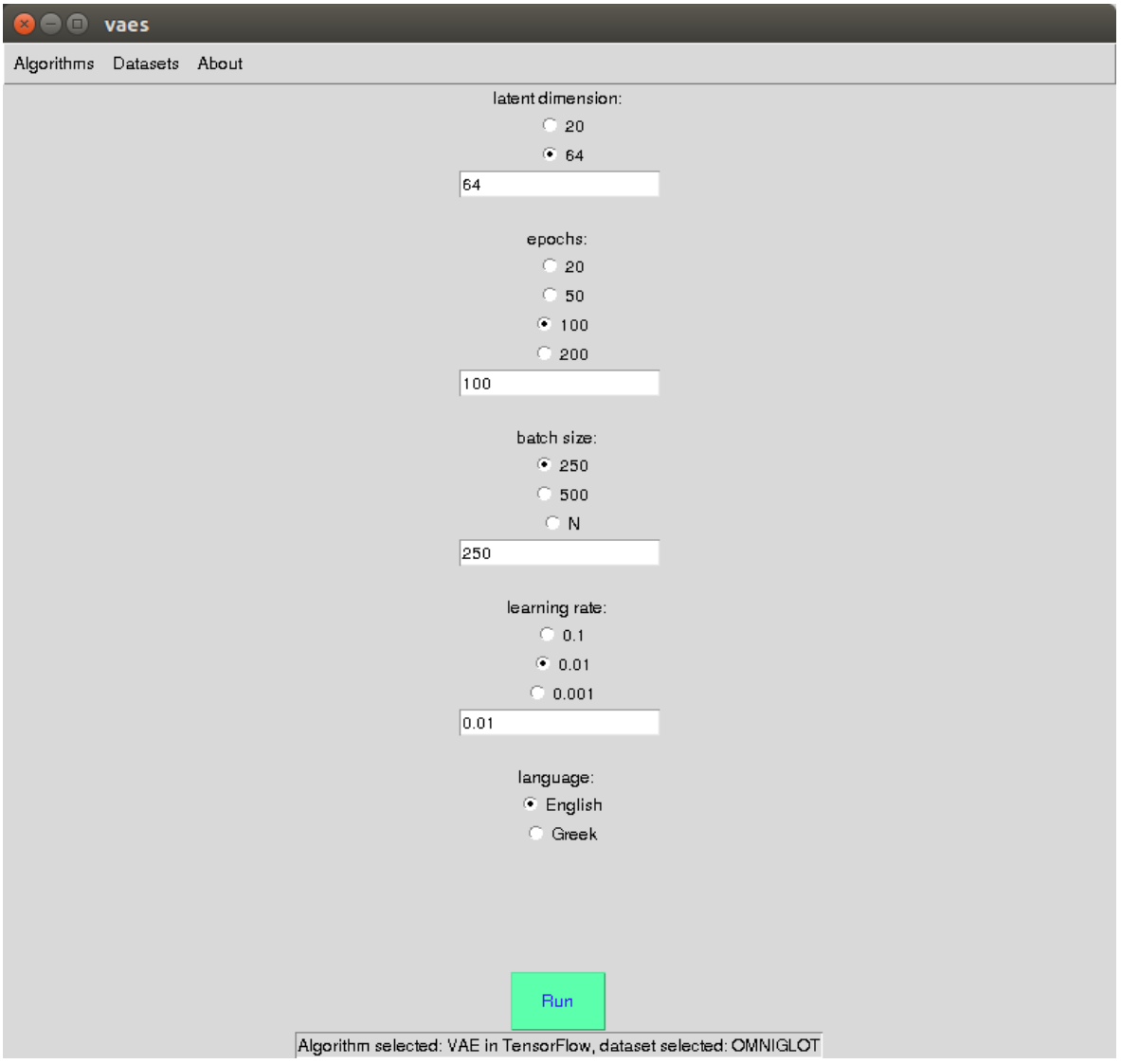


FIGURE 6.5: GUI VAE in TensorFlow, OMNIGLOT dataset.

FIGURE 6.6: GUI K-NN Missing Values algorithm, MNIST dataset.

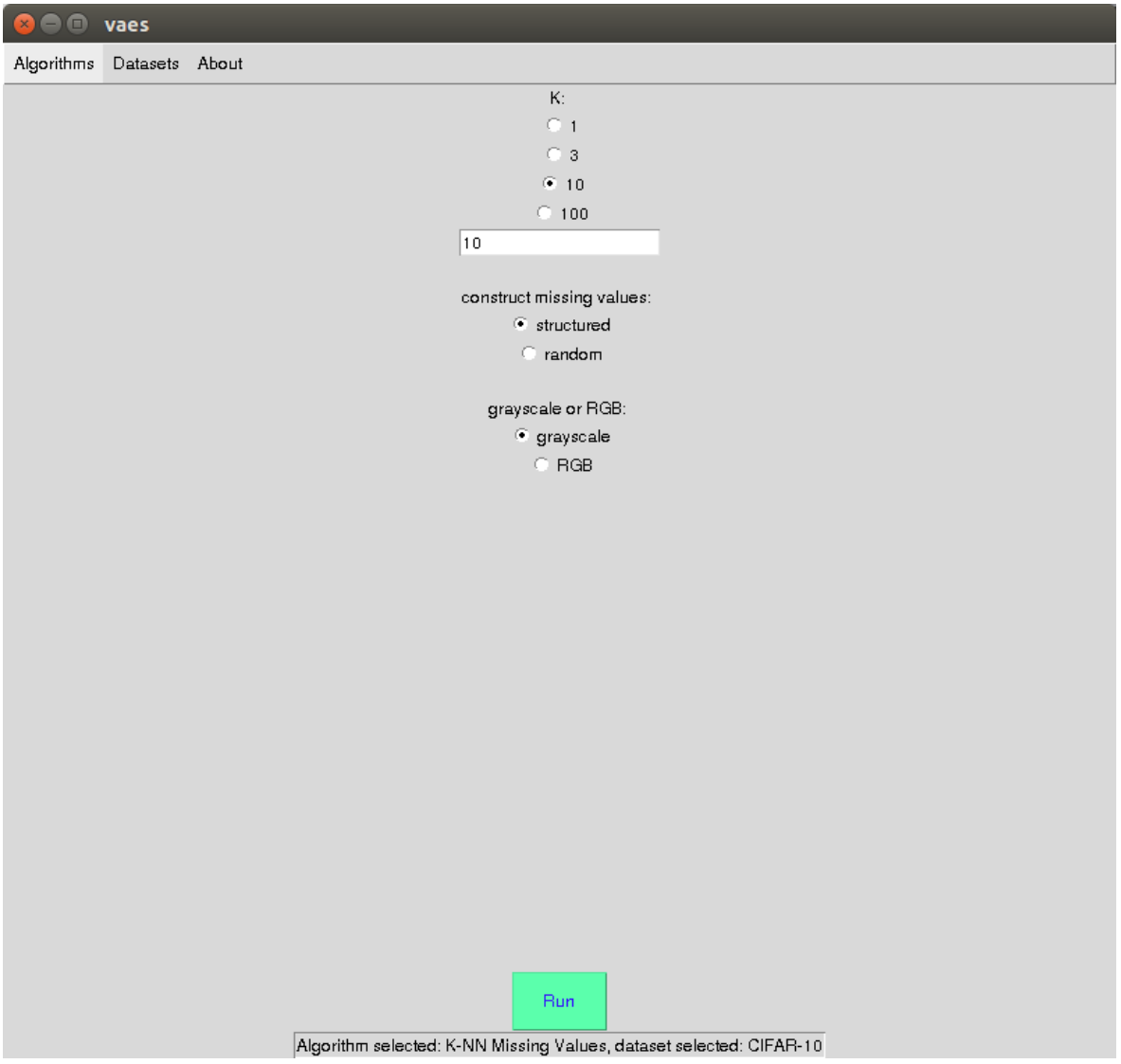


FIGURE 6.7: GUI K-NN Missing Values algorithm, CIFAR-10 dataset.

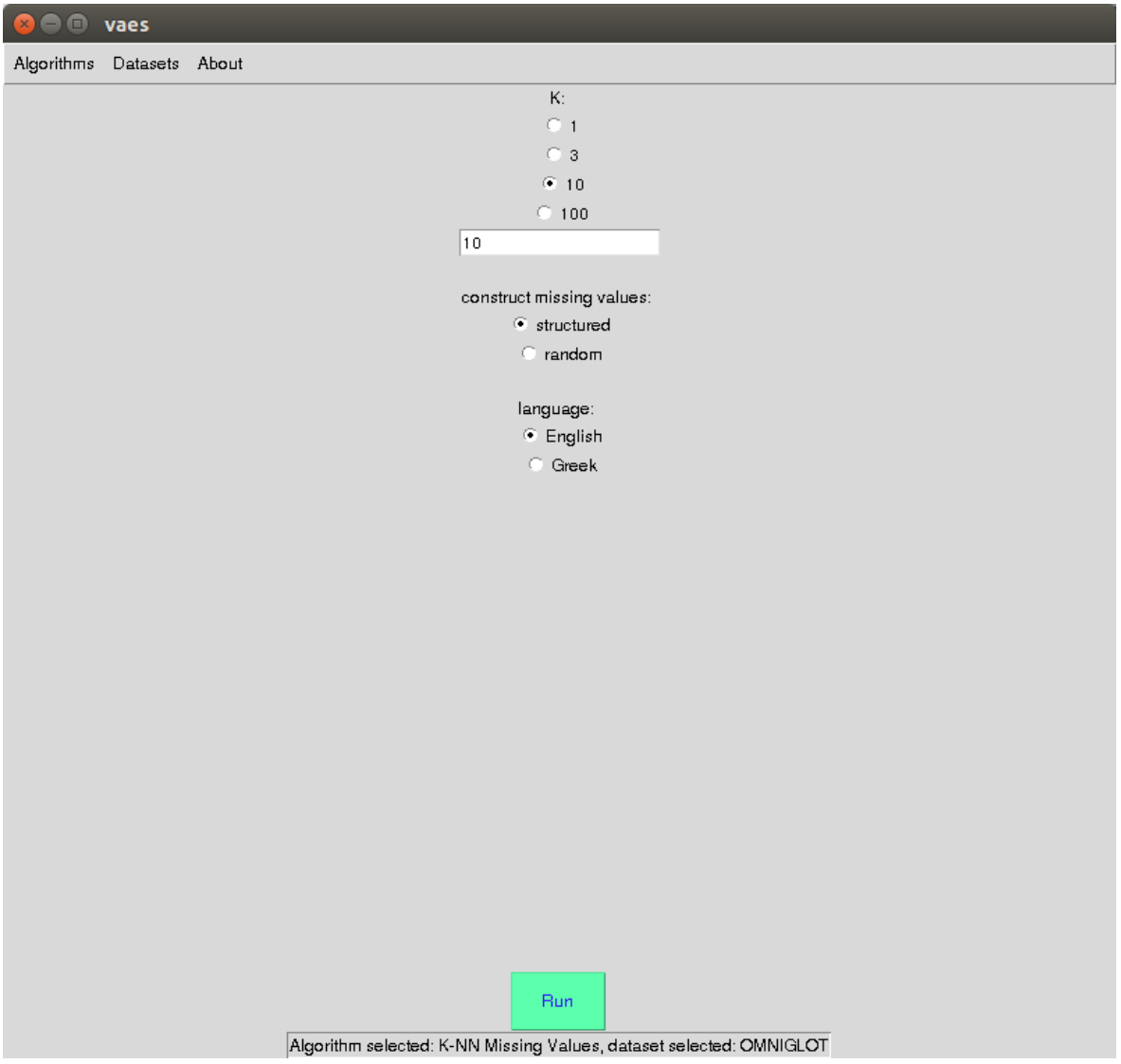


FIGURE 6.8: GUI K-NN Missing Values algorithm, OMNIGLOT dataset.

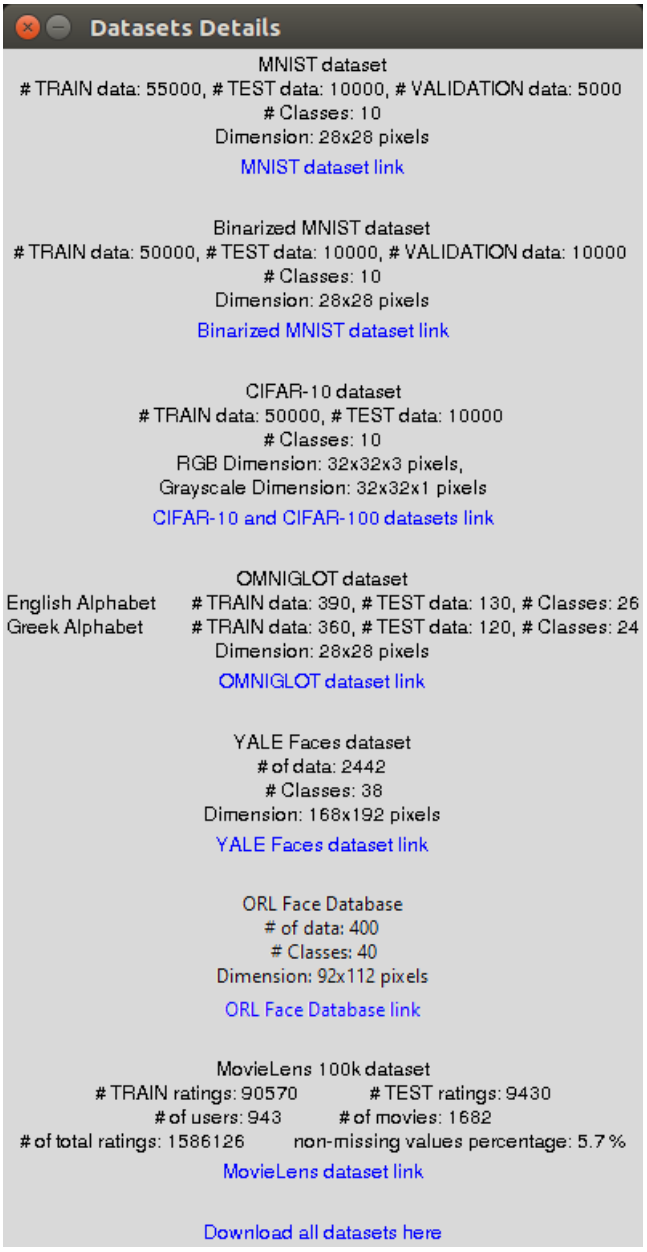


FIGURE 6.9:
GUI datasets details.

FIGURE 6.10: GUI About.

Chapter 7

Further Applications of Variational Autoencoders

(Aaron Courville, Ian Goodfellow, Yoshua Bengio 2016. [4])

Autoencoders have been successfully applied to: **1) dimensionality reduction** and **2) information retrieval** tasks. Dimensionality reduction was one of the first applications of representation learning and deep learning.

Lower-dimensional representations can improve performance on many tasks, such as classification. Models of smaller spaces consume less memory and runtime. The hints provided by the mapping to the lower-dimensional space aid generalization.

One task that benefits even more than usual from dimensionality reduction is **information retrieval**. Information retrieval is the task of finding entries in a database that resemble a query entry. This task derives the usual benefits from dimensionality reduction that other tasks do, but also derives the additional benefit that search can become extremely efficient in certain kinds of low dimensional spaces. Specifically, if we train the dimensionality reduction algorithm to produce a code that is lowdimensional and binary, then we can store all database entries in a hash table mapping binary code vectors to entries. This hash table allows us to perform information retrieval by returning all database entries that have the same binary code as the query. We can also search over slightly less similar entries very efficiently, just by flipping individual bits from the encoding of the query. This approach to information retrieval via dimensionality reduction and binarization is called **semantic hashing**.

Appendix A

Background Theory

A.1 Bayes' Rule

Bayes rule for conditional probabilities:

$$P(A|B) \cdot P(B) = P(B|A) \cdot P(A) \Rightarrow$$

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

The following equation also exists:

$$P(A|B) = \frac{P(A,B)}{P(B)}$$

A.2 Softmax Function

Let X be an input vector or a matrix. The softmax function constructs weights for each element of the input X. The element with the biggest value will get the weight with the biggest value. Also, the sum of all the weights must be equal to 1. Thus, the softmax function is given from the following formula:

$$\mathbf{w_i} = \mathbf{softmax}(\mathbf{X}) = \frac{\mathbf{e^{X_i}}}{\sum_{\mathbf{j=1}}^{\mathbf{N}} \mathbf{e^{X_j}}},$$

$$\text{and } w_1 + w_2 + w_3 + ... + w_N = \sum_{i=1}^{N} w_i = \sum_{i=1}^{N} \frac{e^{X_i}}{\sum_{j=1}^{N} e^{X_j}} = \frac{\sum_{i=1}^{N} e^{X_i}}{\sum_{j=1}^{N} e^{X_j}} = 1$$

A.3 Entropy

Information entropy is defined as the average amount of information produced by a stochastic source of data.

The formula of the information entropy of a distribution P is the following:

$$H(P) = -\sum_{x} P(x) \cdot log_b P(x)$$

where b is the base of the logarithm used

Information entropy is typically measured in bits (alternatively called **"Shannons"**) for *b=2* or sometimes in "natural units" (**nats**) for *b=e* (Euler's constant), or decimal digits (called "dits", "bans", or "hartleys") for *b=10*. The unit of the measurement depends on the base of the logarithm that is used to define the entropy.

A.4 Cross-Entropy

Let Q be an "unnatural" probability distribution and P be the "true" distribution over the same underlying set of events. The cross entropy between the two probability distributions P and Q measures the average number of bits needed to identify an event drawn from the set.

The formula of the cross entropy of two discrete distributions P and Q is the following:

$$H(P, Q) = -\sum_{x} P(x) \cdot log_b Q(x)$$

Likewise, the formula of the cross entropy of two continuous distributions P and Q is the following:

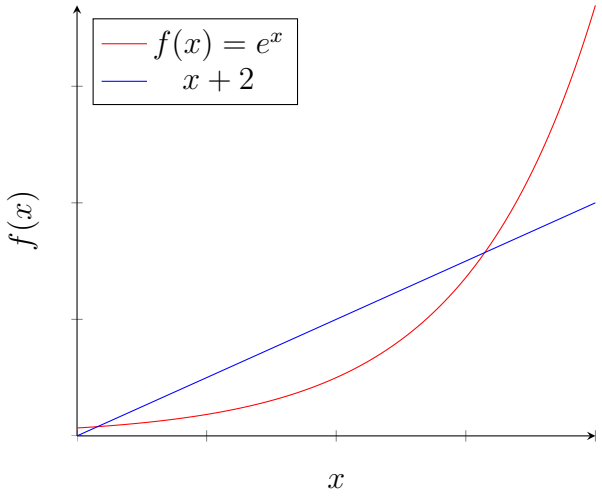
$$H(P, Q) = -\int_{-\infty}^{+\infty} P(x) \cdot log_b Q(x)\, dx \Rightarrow$$
$$H(P, Q) = -E_P[-logQ(x)]$$

where b is the base of the logarithm used

A.5 Jensen's Inequality

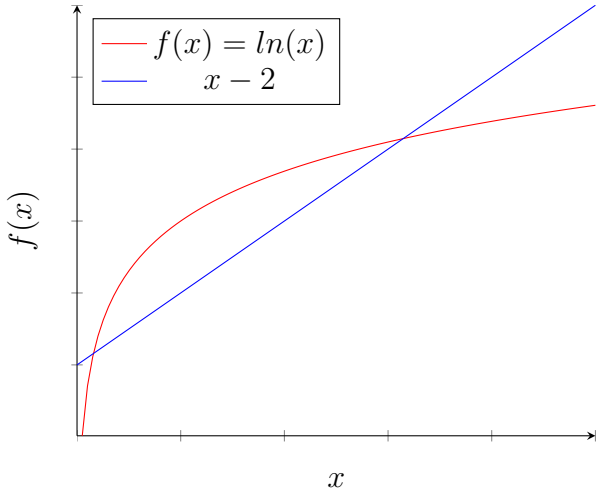
Jensen's inequality, named after the Danish mathematician Johan Jensen, relates the value of a convex (or concave) function of an integral to the integral of the convex function. It generalizes the statement that a secant line of a convex function lies above the graph.

$f(E[X]) \leq E[f(X)]$, where f is a convex function



In a similar manner, a secant line of a concave function lies below the graph.

$f(E[X]) \geq E[f(X)]$, where f is a concave function



A.6 Kullback-Leibler (KL) Divergence

(Aaron Courville, Ian Goodfellow, Yoshua Bengio 2016. [4])

If we have two separate probability distributions $P(x)$ and $Q(x)$ over the same random variable x, we can measure how different these two distributions are using the Kullback-Leibler (KL) divergence:

$$D_{KL}[P \,||\, Q] = E_{X\sim P}[log\frac{P(X)}{Q(X)}] = E_{X\sim P}[logP(X) - logQ(X)]$$

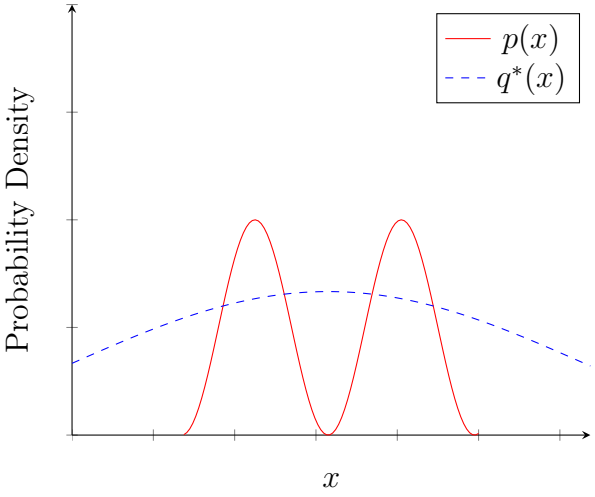
In the case of discrete variables, it is the extra amount of information (measured in bits if we use the base 2 logarithm, but in machine learning we usually use nats and the natural logarithm) needed to send a message containing symbols drawn from probability distribution P, when we use a code that was designed to minimize the length of messages drawn from probability distribution Q. The KL divergence has many useful properties, most notably that it is non-negative. The KL divergence is 0 if and only if P and Q are the same distribution in the case of discrete variables, or equal "almost everywhere" in the case of continuous variables. Because the KL divergence is non-negative and measures the difference between two distributions, it is often conceptualized as measuring some sort of distance between these distributions. However, it is not a true distance measure because it is not symmetric: $D_{KL}(P \,||\, Q) \neq D_{KL}(Q \,||\, P)$ for some P and Q. This asymmetry means that there are important consequences to the choice of whether to use $D_{KL}(P \,||\, Q)$ or $D_{KL}(Q \,||\, P)$. A quantity that is closely related to the KL divergence is the cross-entropy $H(P, Q) = H(P) + D_{KL}(P \,||\, Q)$, which is similar to the KL divergence but lacking the term on the left:

$$H(P, Q) = -E_{X\sim P}\, log(Q(X))$$

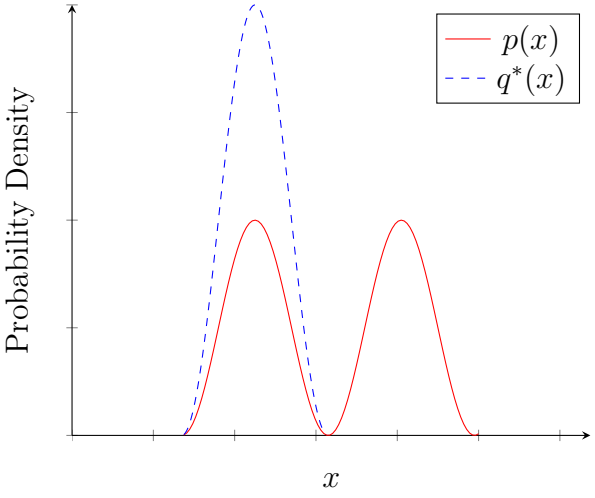
Minimizing the cross-entropy with respect to Q is equivalent to minimizing the KL divergence, because Q does not participate in the omitted term. When computing many of these quantities, it is common to encounter expressions of the form $0 \cdot log\, 0$. By convention, in the context of information theory, we treat these expressions as $\lim_{x\to 0} x \cdot log\, x = 0$.

The KL divergence is asymmetric:

$$q^* = argmin_q D_{KL}(p \,||\, q)$$



$$q^* = argmin_q D_{KL}(q \,||\, p)$$



A.7 Mean Absolute Error (MAE)

$$\frac{\sum_{i=1}^{N} |X_i - X_{recon\_i}|}{N}$$

A.8 Mean Squared Error (MSE)

$$\frac{\sum_{i=1}^{N} (X_i - X_{recon\_i})^2}{N}$$

A.9 Occam's Razor

The principle of Occam's Razor (aka Ockham's razor) states that among competing hypotheses, the one with the fewest assumptions should be selected. The idea is attributed to William of Ockham (1287–1347), who was an English Franciscan friar, scholastic philosopher, and theologian. The principle in Latin is stated as **"Pluralitas non est ponenda sine neccesitate"**, which translates in English as: **"Plurality is not to be posited without necessity"**.

A.10 Root Mean Squared Error (RMSE)

$$\sqrt{\frac{\sum_{i=1}^{N} (X_i - X_{recon\_i})^2}{N}}$$

A.11 Update Rules of Neural Network Weights

These are the generalized updates rules used in back-propagation.

- **Batch gradient descent**
  Update rule for the encoder parameters:

$$W = W - \eta \cdot \frac{\partial E}{\partial W}$$

- **Stochastic gradient descent**
  Update rule for the encoder parameters:

$$for\ each\ x_i : W = W - \eta \cdot \frac{\partial E_i}{\partial W}$$

- **Batch gradient ascent**
  Update rule for the encoder parameters:

$$W = W + \eta \cdot \frac{\partial E}{\partial W}$$

- **Stochastic gradient ascent**
  Update rule for the encoder parameters:

$$for\ each\ x_i : W = W + \eta \cdot \frac{\partial E_i}{\partial W}$$

where W are the weights of the Neural Network and E is the loss function to be maximized (gradient ascent) or minimized (gradient descent).

Appendix B

Probability Distributions

B.1 Bernoulli Distribution

(Christopher M. Bishop 2006. [8]) This is the distribution for a single binary variable $x \in 0, 1$ representing, for example, the result of flipping a coin. It is governed by a single continuous parameter $\mu \in [0, 1]$ that represents the probability of $x = 1$.

$$Bern(x|\mu) = \mu^x \cdot (1 - \mu)^{1-x}$$

$$E[x] = \mu$$

$$VAR[x] = \mu \cdot (1 - \mu)$$

$$mode[x] = \begin{cases} 1, \ if \ \mu >= 0.5 \\ 0, \ otherwise \end{cases}$$

$$H[x] = -\mu \cdot ln\mu - (1 - \mu) \cdot ln(1 - \mu)$$

The Bernoulli is a special case of the binomial distribution for the case of a single observation. Its conjugate prior for μ is the beta distribution.

pmf	Mean	Variance
$\begin{cases} q = (1 - p), \ for \ k = 0 \\ p, \ for \ k = 1 \end{cases}$	p	$p \cdot (1 - p) = p \cdot q$

TABLE B.1: Bernoulli distribution, probability mass function (pmf), mean & variance.

B.2 Binomial Distribution

(Christopher M. Bishop 2006. [8]) The binomial distribution gives the probability of observing m occurrences of $x = 1$ in a set of N samples from a Bernoulli distribution, where the probability of observing $x = 1$ is $\mu \in [0, 1]$.

$$Bin(k|N, \mu) = \binom{N}{k} \cdot \mu^k \cdot (1 - \mu)^{N-k}$$

$$E[k] = N \cdot \mu$$

$$VAR[k] = N \cdot \mu \cdot (1 - \mu)$$

$$mode[k] = \lfloor (N + 1) \cdot \mu \rfloor$$

where $\lfloor (N + 1) \cdot \mu \rfloor$ denotes the largest integer that is less than or equal to $(N + 1) \cdot \mu$, and the quantity:

$$\binom{N}{k} = \frac{N!}{k!(N - k)!}$$

denotes the number of ways of choosing m objects out of a total of N identical objects.

Here $m!$, pronounced *'factorial m'*, denotes the product $m \times (m - 1) \times \ldots, \times 2 \times 1$. The particular case of the binomial distribution for $N = 1$ is known as the Bernoulli distribution, and for large N the binomial distribution is approximately Gaussian. The conjugate prior for μ is the beta distribution.

pmf	**Mean**	**Variance**
$\binom{n}{k} = \frac{n!}{k!(n-k)!}$	$n \cdot p$	$n \cdot p \cdot (1 - p)$

TABLE B.2: Binomial distribution, probability mass function (pmf), mean & variance.

B.3 Gaussian (or Normal) Distribution

(Christopher M. Bishop 2006. [8]) The Gaussian is the most widely used distribution for continuous variables. It is also known as the normal distribution. In the case of a single variable $x \in (-\infty, \infty)$ it is governed by two parameters, the mean $\mu \in (-\infty, \infty)$ and the variance $\sigma^2 > 0$.

$$N(x|\mu, \sigma^2) = \frac{1}{\sqrt{2 \cdot \pi\sigma^2}} \cdot e^{-\frac{1}{2 \cdot \sigma^2} \cdot (x-\mu)^2}$$

$$E[x] = \mu$$

$$VAR[x] = \sigma^2$$

$$mode[x] = \mu$$

$$H[x] = \frac{1}{2} \cdot ln\sigma^2 + \frac{1}{2} \cdot (1 + ln(2 \cdot \pi))$$

The inverse of the variance $\tau = \frac{1}{\sigma^2}$ is called the precision, and the square root of the variance Σ is called the standard deviation. The conjugate prior for μ is the Gaussian, and the conjugate prior for τ is the gamma distribution. If both μ and τ are unknown, their joint conjugate prior is the Gaussian-gamma distribution. For a D-dimensional vector x, the Gaussian is governed by a D-dimensional mean vector μ and a $D \times D$ covariance matrix Σ that must be symmetric and positive-definite.

$$N(x|\mu, \Sigma) = \frac{1}{(2 \cdot \pi)^{D/2}} \cdot \frac{1}{\sqrt{|\Sigma|}} \cdot \mathrm{e}^{-\frac{1}{2} \cdot (x-\mu)^T \cdot \Sigma^{-1} \cdot (x-\mu)}$$

$$E[x] = \mu$$

$$Cov[x] = \Sigma$$

$$mode[x] = \mu$$

$$H[x] = \frac{1}{2} \cdot ln|\Sigma| + \frac{D}{2} \cdot (1 + ln(2 \cdot \pi))$$

The inverse of the covariance matrix $\Lambda = \Sigma^{-1}$ is the precision matrix, which is also symmetric and positive definite. By the central limit theorem, averages of random variables tend to a Gaussian and the sum of two Gaussian variables is again Gaussian. The Gaussian is the distribution that maximizes the entropy for a given variance (or covariance). Any linear transformation of a Gaussian random variable is again Gaussian. The marginal distribution of a multivariate Gaussian with respect to a subset of the variables is itself Gaussian, and

similarly the conditional distribution is also Gaussian. If we have a marginal Gaussian distribution for x and a conditional Gaussian distribution for y given x in the form:

$$P(x) = N(x|\mu, \Lambda^{-1)}$$

$$P(y|x) = N(y|A \cdot x + b, L^{-1})$$

then the marginal distribution of y, and the conditional distribution of x given y, are given by:

$$P(y) = N(y|A \cdot \mu + b, L^{-1} + A \cdot \Lambda^{-1} \cdot A^T)$$

$$P(x|y) = N(x|\Sigma A^T \cdot L(y - b) + \Lambda \cdot \mu, \Sigma)$$

where:

$$\Sigma = (\Lambda + A^T \cdot L \cdot A)^{-1}.$$

If we have a joint Gaussian distribution $N(x|\mu, \Sigma)$ with $\Lambda = \Sigma^{-1}$ and we define the following partitions:

$$x = \begin{bmatrix} x_a \\ x_b \end{bmatrix}, \mu = \begin{bmatrix} \mu_a \\ \mu_b \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{bmatrix}, \Lambda = \begin{bmatrix} \Lambda_{aa} & \Lambda_{ab} \\ \Lambda_{ba} & \Lambda_{bb} \end{bmatrix}$$

then the conditional distribution $P(x_a|x_b)$ is given by:

$$P(x_a|x_b) = N(x|\mu_{a|b}, \Lambda_{aa}^{-1})$$

$$\mu_{a|b} = \mu_a - \Lambda_{aa}^{-1} \cdot \Lambda ab \cdot (x_b - \mu_b)$$

and the marginal distribution $P(x_a)$ is given by:

$$P(x_a) = N(x_a|\mu_a, \Sigma_a).$$

pmf	Mean	Variance
$\frac{1}{\sqrt{2\cdot\pi\sigma^2}}\cdot e^{-\frac{1}{2\cdot\sigma^2}\cdot(x-\mu)^2}$	μ	σ^2

TABLE B.3: Gaussian distribution, probability mass function (pmf), mean & variance.

B.4 Law of Large Numbers (L.L.N.)

(Σταύρος Τουμπής, Γιάννης Κοντογιάννης 2011. [9]) The 1^{st} fundamental theorem of the probability theory is the Law of Large Number (L.L.N.).

Definition

Let $X_i, i = 1, 2, \ldots$ be a sequence of independent random variables with the same distribution, mean value $E(X) = \mu$ and variance $VAR(X) = \sigma^2$. The L.L.N. which makes sure that the probability the following empirical mean:

$$\bar{X}_N \approx \frac{1}{N}\cdot\sum_{i=1}^{N} X_i$$

will have a deviation from μ greater than ϵ tends to 0. Hence:

$$\lim_{N\to\infty} P(|\bar{X}_N - \mu| > \epsilon) = 0$$

Furthermore, the upper bound of this limit is proven to be:

$$\lim_{N\to\infty} P(|\bar{X}_N - \mu| > \epsilon) \leq \frac{\sigma^2}{N\cdot\epsilon^2}$$

where: $E(\bar{X}_N = \mu)$ and $Var(\bar{X}_N) = \frac{\sigma^2}{N}$, for every N

B.5 Central Limit Theorem (C.L.T.)

(Σταύρος Τουμπής, Γιάννης Κοντογιάννης 2011. [9]) The 2^{nd} fundamental theorem of the probability theory is the Central Limit Theorem (C.L.T.). Through this theorem we can estimate with greater precision the limit $\lim_{N\to\infty} P(|\bar{X}_N - \mu| > \epsilon)$. Concretely, the C.L.T. tells us that under certain circumstances, the distribution of the empirical mean value $\bar{X}_N$ can be estimated with great precision by the Gaussian (aka Normal) distribution, with parameters μ and $\frac{\sigma^2}{N}$, where σ^2 is the variance of the random variables X_i.

Definition

Let there be a sequence of independent random variables $X_1, X_2, \ldots$ that follow the same distribution and therefore they have the same mean $\mu = E(X_i)$ and the same variance $\sigma^2 = VAR(X_i)$. Let there be the following regularized sum:

$$\bar{S}_N = \frac{(\frac{1}{N} \cdot \sum_{i=1}^{N} X_i - \mu)}{\sigma \cdot \sqrt{\frac{1}{N}}} = \frac{1}{\sigma \cdot \sqrt{N}} \cdot \sum_{i=1}^{N} (X_i - \mu) = \frac{(\sum_{i=1}^{N} X_i) - N \cdot \mu}{\sigma \cdot \sqrt{N}}$$

In addition, let there be $a, b \in \mathbb{R}, a < b$. The following formulae hold, for $N \to \infty$:

- $P(a \leq \bar{S}_N \leq b) \to \Phi(b) - \Phi(a)$
- $P(\bar{S}_N \leq b) \to \Phi(b)$
- $P(\bar{S}_N \geq a) \to 1 - \Phi(a)$

where $\Phi(x)$ is the Gaussian (or Normal) distribution

Appendix C

Programming Implementations of Variational Autoencoders in Python

C.1 VAE Implementation in TensorFlow

Here's an implementation of a VAE algorithm, in TensorFlow:

```
import numpy as np

X_train  # matrix with the original data
y_train  # labels of the matrix "X_train"
N # the number of training examples
input_dim  # the input dimensionality D
hidden_encoder_dim  # number of neurons in the encoder
hidden_decoder_dim  # number of neurons in the decoder
latent_dim  # Z_dim
epochs  # the number of total epochs for the training
batch_size  # the number of data trained in each iteration
learning_rate  # the rate of learning "eta" for the optimization algorithm
log_dir  # the location where the tensorboard graph will be saved

x, loss_summ, apply_updates, summary_op, saver, elbo, x_recon_samples =
    vae(batch_size, input_dim, hidden_encoder_dim, hidden_decoder_dim, latent_dim,
    lr)

X_recon = np.zeros((N, input_dim))

with tf.Session() as sess:
    summary_writer = tf.summary.FileWriter(log_dir, graph=sess.graph)

    for epoch in range(epochs):
        iterations = int(N / batch_size)

        for i in range(iterations):
            start_index = i * batch_size
            end_index = (i + 1) * batch_size

            batch_data = X_train[start_index:end_index, :]
```

```
31              batch_labels = y_train[start_index:end_index]
32
33              feed_dict = {x: batch_data}
34              loss_str, _, summary_str, cur_elbo, cur_samples =
35                  sess.run([loss_summ, apply_updates, summary_op,
36                  elbo, x_recon_samples], feed_dict=feed_dict)
37
38              X_recon[start_index:end_index, :] = cur_samples
39
40              summary_writer.add_summary(loss_str, epoch)
41              summary_writer.add_summary(summary_str, epoch)
```

Listing C.1: VAE Main Loop in TensorFlow

Here's the implementation of the "vae" function of the TensorFlow implementation:

```
import tensorflow as tf

# HELPER FUNCTIONS #

def initialize_weight_variable(shape, name=''):
    initial = tf.truncated_normal(shape, stddev=0.001,
                name='truncated_normal_' + name)
    return tf.Variable(initial, name=name)


def initialize_bias_variable(shape, name=''):
    initial = tf.constant(0., shape=shape)
    return tf.Variable(initial, name=name)

#####

def vae(batch_size, input_dim, hidden_encoder_dim, hidden_decoder_dim,
                latent_dim, lr=0.01):
    # Input placeholder
    with tf.name_scope('input_data'):
        x = tf.placeholder('float', [batch_size, input_dim], name='X')

    # ====== Q(Z|X) = Q(Z) - Encoder NN ====== #

    # The encoder is a neural network with 2 hidden layers.
    with tf.name_scope('encoder'):
        with tf.name_scope('Phis'):
            # phi1: M1 x D
            phi1 = initialize_weight_variable([hidden_encoder_dim,
                        input_dim], name='phi1')
            # bias_phi1: 1 x M1
            bias_phi1 = initialize_bias_variable([hidden_encoder_dim],
                        name='bias_phi1')

            # phi_mu: Z_dim x M1
            phi_mu = initialize_weight_variable([latent_dim,
                        hidden_encoder_dim], name='phi_mu')
            # bias_phi_mu: 1 x Z_dim
            bias_phi_mu = initialize_bias_variable([latent_dim],
                                name='bias_phi_mu')

            # phi_logvar: Z_dim x M1
            phi_logvar = initialize_weight_variable([latent_dim,
                            hidden_encoder_dim], name='phi_logvar')
            # bias_phi_logvar: 1 X Z_dim
            bias_phi_logvar = initialize_bias_variable([latent_dim],
```

```
                                    name='bias_phi_logvar')

        with tf.name_scope('hidden_layer1'):
            # Hidden layer 1 activation function of the encoder
            # hidden_layer_encoder: N x M1
            # RELU
            hidden_layer_encoder = tf.nn.relu(tf.nn.bias_add(tf.matmul(
                                    x, tf.transpose(phi1)),
                                    bias_phi1))

        with tf.name_scope('hidden_layer2_mu'):
            # mu_encoder: N x Z_dim
            mu_encoder = tf.nn.bias_add(tf.matmul(hidden_layer_encoder,
                            tf.transpose(phi_mu)), bias_phi_mu)

        with tf.name_scope('hidden_layer2_logvar'):
            # the log sigma^2 of the encoder
            # logvar_encoder: N x Z_dim
            logvar_encoder = tf.nn.bias_add(tf.matmul(hidden_layer_encoder,
                                tf.transpose(phi_logvar)),
                                bias_phi_logvar)

        with tf.name_scope('sample_E'):
            # Sample epsilon
            # epsilon: N x Z_dim
            epsilon = tf.random_normal((batch_size, latent_dim),
                        mean=0.0, stddev=1.0, name='epsilon')

        with tf.name_scope('construct_Z'):
            # Sample epsilon from the Gaussian distribution. #
            # std_encoder: N x Z_dim
            std_encoder = tf.exp(logvar_encoder / 2)
            # z: N x Z_dim
            z = tf.add(mu_encoder, tf.multiply(std_encoder, epsilon),
                    name='Z')

    # ===== P(X|Z) - Decoder NN ===== #

    # The encoder is a neural network with 2 hidden layers.
    with tf.name_scope('decoder'):
        with tf.name_scope('Thetas'):
            # theta1: M2 x Z_dim
            theta1 = initialize_weight_variable([hidden_decoder_dim,
                        latent_dim], name='theta1')
            # bias_theta1: 1 x M2
            bias_theta1 = initialize_bias_variable([hidden_decoder_dim],
                        name='bias_theta1')

```

```
95              # theta2: D x M2
96              theta2 = initialize_weight_variable([input_dim,
97                          hidden_decoder_dim], name='theta2')
98              # bias_theta2: 1 x D
99              bias_theta2 = initialize_bias_variable([input_dim],
100                         name='bias_theta2')
101
102         with tf.name_scope('hidden_layer1'):
103             # Hidden layer 1 activation function of the decoder
104             # hidden_layer_decoder: N x M2
105             # RELU
106             hidden_layer_decoder = tf.nn.relu(tf.nn.bias_add(tf.matmul(z,
107                         tf.transpose(theta1)), bias_theta1))
108
109         with tf.name_scope('output_layer'):
110             # x_hat: N x D
111             x_hat = tf.nn.bias_add(tf.matmul(hidden_layer_decoder,
112                         tf.transpose(theta2)), bias_theta2)
113
114     with tf.name_scope('reconstructed_data'):
115         # X_recon_samples: N x D, reconstructed data
116         x_recon_samples = tf.nn.sigmoid(x_hat, name='X_recon_samples')
117
118     # ======= TRAINING ========== #
119
120     with tf.name_scope('ELBO'):
121         with tf.name_scope('reconstruction_cost'):
122             # log P(X)
123             # reconstruction_cost: N x 1
124             reconstruction_cost = tf.reduce_sum(tf.nn.
125                     sigmoid_cross_entropy_with_logits(logits=x_hat,
126                     labels=x), reduction_indices=1)
127
128         with tf.name_scope('KL_divergence'):
129             # KLD: N x 1
130             KLD = 0.5 * tf.reduce_sum(1 + logvar_encoder -
131                     tf.square(mu_encoder) - tf.exp(logvar_encoder),
132                     reduction_indices=1)
133
134         elbo = tf.reduce_mean(reconstruction_cost - KLD,
135                     name='lower_bound')
136
137         loss_summ = tf.summary.scalar('ELBO', elbo)
138
139         phis = [phi1, bias_phi1,
140                 phi_mu, bias_phi_mu,
141                 phi_logvar, bias_phi_logvar]
142         thetas = [theta1, bias_theta1,
```

```
143                     theta2, bias_theta2]
144         var_list = phis + thetas
145 
146         # Adam Optimizer (WORKS BEST!) #
147         grads_and_vars = tf.train.AdamOptimizer(learning_rate=lr). \
148                          compute_gradients(loss=elbo, var_list=var_list)
149         apply_updates = tf.train.AdamOptimizer(learning_rate=lr). \
150                         apply_gradients(grads_and_vars=grads_and_vars)
151 
152     # add op for merging summary
153     summary_op = tf.summary.merge_all()
154 
155     # add Saver ops
156     saver = tf.train.Saver()
157 
158     return x, loss_summ, apply_updates, summary_op, saver,
159             elbo, x_recon_samples
```

LISTING C.2: VAE Function in TensorFlow

NOTE: For optimizer of the gradients we have used the Adam optimizer (tf.train.AdamOptimizer), because it brings the best results. All the optimizers provided by the TensorFlow framework are the following:

- tf.train.GradientDescentOptimizer
- tf.train.AdadeltaOptimizer
- tf.train.AdagradOptimizer
- tf.train.AdagradDAOptimizer
- tf.train.MomentumOptimizer
- **tf.train.AdamOptimizer**
- tf.train.FtrlOptimizer
- tf.train.ProximalGradientDescentOptimizer
- tf.train.ProximalAdagradOptimizer
- tf.train.RMSPropOptimizer

C.2 VAE Implementation in PyTorch

Here's an implementation of a VAE algorithm, in PyTorch:

```
import numpy as np

X_train  # matrix with the original data
y_train  # labels of the matrix "X_train"
N  # the number of training examples
input_dim  # the input dimensionality D
hidden_encoder_dim  # number of neurons in the encoder
hidden_decoder_dim  # number of neurons in the decoder
latent_dim  # Z_dim
epochs  # the number of total epochs for the training
batch_size  # the number of data trained in each iteration
learning_rate  # the rate of learning "eta" for the optimization algorithm

# initialize_weights:
# A function that initializes the weights of the encoder and
# the decoder neural networks.
params # the weights of the encoder and the decoder neural networks,
       # that are updated on each iteration
solver # the optimizer used for training (e.g SGD)
params, solver = initialize_weights(input_dim, hidden_encoder_dim, \
                           hidden_decoder_dim, latent_dim, lr)

for epoch in range(epochs):
    iterations = int(N / batch_size)

    for i in range(iterations):
        start_index = i * batch_size
        end_index = (i + 1) * batch_size

        batch_data = X_train[start_index:end_index, :]
        batch_labels = y_train[start_index:end_index]

        cur_samples, cur_elbo = train(batch_data, batch_size, \
                                      latent_dim, params, solver)

        X_train[start_index:end_index, :] = cur_samples
```

Listing C.3: VAE Main Loop in PyTorch

Here's the implementation of the "initialize_weights" function of the PyTorch implementation:

```
import numpy as np
import torch
import torch.optim as optim
from torch.autograd import Variable


def xavier_init(size):
    in_dim = size[0]
    xavier_stddev = 1. / np.sqrt(in_dim / 2.)
    return Variable(torch.randn(*size) * xavier_stddev,
                    requires_grad=True)


def initialize_weights(X_dim, hidden_encoder_dim, hidden_decoder_dim,
                       Z_dim, lr=0.01):

    # ====== Encoder Parameters: Phis ====== #

    # M1 x D
    phi1 = xavier_init(size=[hidden_encoder_dim, X_dim])
    # 1 x M1
    bias_phi1 = Variable(torch.zeros(hidden_encoder_dim),
                         requires_grad=True)

    # Z_dim x M1
    phi_mu = xavier_init(size=[Z_dim, hidden_encoder_dim])
    # 1 x Z_dim
    bias_phi_mu = Variable(torch.zeros(Z_dim),
                           requires_grad=True)

    # Z_dim x M1
    phi_logvar = xavier_init(size=[Z_dim, hidden_encoder_dim])
    # 1 x Z_dim
    bias_phi_logvar = Variable(torch.zeros(Z_dim),
                               requires_grad=True)

    # ====== Decoder Parameters: Thetas ====== #

    # M2 x Z_dim
    theta1 = xavier_init(size=[hidden_decoder_dim, Z_dim])
    # 1 x M2
    bias_theta1 = Variable(torch.zeros(hidden_decoder_dim),
                           requires_grad=True)

    # D x M2
```

```
46      theta2 = xavier_init(size=[X_dim, hidden_decoder_dim])
47      # 1 x D
48      bias_theta2 = Variable(torch.zeros(X_dim),
49                             requires_grad=True)
50
51      # ====== TRAINING ======
52
53      phis = [phi1, bias_phi1,
54              phi_mu, bias_phi_mu,
55              phi_logvar, bias_phi_logvar]
56      thetas = [theta1, bias_theta1,
57              theta2, bias_theta2]
58      params = phis + thetas
59
60      solver = optim.Adam(params, lr=lr)
61
62      return params, solver
```

LISTING C.4: Initialization of VAE Weights in PyTorch

NOTE: For optimizer of the gradients we have used the Adam optimizer (torch.optim.Adam), because it brings the best results. All the optimizers provided by the PyTorch framework are the following:

- torch.optim.Adadelta
- torch.optim.Adagrad
- **torch.optim.Adam**
- torch.optim.SparseAdam
- torch.optim.AdaMax
- torch.optim.ASGD
- torch.optim.LBFGS
- torch.optim.RMSprop
- torch.optim.Rprop
- torch.optim.SGD

Here's the implementation of the "train" function of the PyTorch implementation:

```
import torch
import torch.nn.functional as nn
from torch.autograd import Variable


# @: denotes matrix multiplication
def train(x, mb_size, Z_dim, params, solver):
    x = Variable(torch.from_numpy(x).float())

    # Phis
    phi1 = params[0]  # M1 x D
    bias_phi1 = params[1]  # 1 x M1
    phi_mu = params[2]  # Z_dim x M1
    bias_phi_mu = params[3]  # 1 x Z_dim
    phi_logvar = params[4]  # Z_dim x M1
    bias_phi_logvar = params[5]  # 1 x Z_dim

    # Thetas
    theta1 = params[6]  # M2 x Z_dim
    bias_theta1 = params[7]  # 1 x M2
    theta2 = params[8]  # D x M2
    bias_theta2 = params[9]  # 1 x D

    # ======= Q(Z|X) = Q(Z) - Encoder NN ======= #

    hidden_layer_encoder = nn.relu(x @ torch.transpose(phi1, 0, 1) +
                           bias_phi1.repeat(mb_size, 1))
    mu_encoder = hidden_layer_encoder @ torch.transpose(phi_mu, 0, 1) +
                 bias_phi_mu.repeat(hidden_layer_encoder.size(0), 1)
    logvar_encoder = hidden_layer_encoder @ torch.transpose(phi_logvar, 0, 1)
                + bias_phi_logvar.repeat(hidden_layer_encoder.size(0), 1)

    # Sample epsilon from the Gaussian distribution. #
    epsilon = Variable(torch.randn(mb_size, Z_dim))
    # std_encoder: N x Z_dim
    std_encoder = torch.exp(logvar_encoder / 2)
    # Sample the latent variables Z. #
    z = mu_encoder + std_encoder * epsilon

    # ======= P(X|Z) - Decoder NN ======= #

    hidden_layer_decoder = nn.relu(z @ torch.transpose(theta1, 0, 1) +
                           bias_theta1.repeat(z.size(0), 1))
    x_hat = hidden_layer_decoder @ torch.transpose(theta2, 0, 1) +
            bias_theta2.repeat(hidden_layer_decoder.size(0), 1)
    x_recon_samples = nn.sigmoid(x_hat)
```

Christos Kormaris

```
47
48      # Loss #
49      recon_loss = nn.binary_cross_entropy(x_recon_samples,
50                   x, size_average=False) / mb_size
51      kl_loss = torch.mean(0.5 * torch.sum(1 + logvar_encoder
52                - mu_encoder ** 2 - torch.exp(logvar_encoder), 1))
53      elbo_loss = recon_loss - kl_loss
54
55      # Backward #
56      elbo_loss.backward()
57
58      # Update #
59      solver.step()
60
61      # Housekeeping #
62      for p in params:
63          p.grad.data.zero_()
64
65      # convert to numpy data type and return
66      x_recon_samples = x_recon_samples.data.numpy()
67      elbo_loss = elbo_loss.data.numpy()[0]
68
69      return x_recon_samples, elbo_loss
```

Listing C.5: VAE Train Function in PyTorch

C.3 VAE Implementation in Keras

Here's an implementation of a VAE algorithm, in Keras (using TensorFlow as backend):

```
import numpy as np
from keras.callbacks import TensorBoard
from keras.layers import Input, Dense
from keras.models import Model

X_train  # matrix with the original train data
X_test  # matrix with the test data
X_valid  # matrix with the cross-validation data
input_dim  # the input dimensionality D
latent_dim  # Z_dim
epochs  # the number of total epochs for the training
batch_size # the number of data trained in each iteration
log_dir # the location where the tensorboard graph will be saved

# this is our input placeholder
input_img = Input(shape=(input_dim,))

# 'encoded' is the encoded representation of the input
encoded = Dense(latent_dim, activation='relu')(input_img)
# 'decoded' is the lossy reconstruction of the input
decoded = Dense(input_dim, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)

# ====== Q(Z|X) = Q(Z) - Encoder NN ====== #

# this model maps an input to its encoded representation
encoder = Model(input_img, encoded)

# ====== P(X|Z) - Decoder NN ====== #

# create a placeholder for an encoded latent_dim input
encoded_input = Input(shape=(latent_dim,))
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# create the decoder model
decoder = Model(encoded_input, decoder_layer(encoded_input))

# compile the model
# optimizer: Adadelta
# loss function: binary crossentropy loss
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

Christos Kormaris

```
44
45 # fit the model
46 autoencoder.fit(X_train, X_train,
47                 epochs=epochs,
48                 batch_size=batch_size,
49                 shuffle=True,
50                 validation_data=(X_valid, X_valid),
51                 callbacks=[TensorBoard(log_dir=log_dir,
52                 histogram_freq=0, write_graph=False)])
53
54 # encode and decode some digits
55 # note that we take them from the *test* set
56 encoded_imgs = encoder.predict(X_test)
57 decoded_imgs = decoder.predict(encoded_imgs)
```

Listing C.6: VAE Main Loop in Keras

C.4 K-NN Missing Values Completion Algorithm in Python

Here's the full code for the K-NN missing values completion algorithm in Python:

```
X_train_missing  # TRAIN data with missing values
X_test_missing  # TEST data with missing values
K = 1
missing_value = 0.5

X_test_predicted = kNNMatrixCompletion(X_train_missing,
                    X_test_missing, K, missing_value)

# Customized Euclidean distance function.
def sqdist(X_train, X_test_instance, missing_value):
    Ntrain = X_train.shape[0]
    D = X_test.shape[1]

    X_train_common = np.array(X_train)
    s = np.where(X_test_instance == missing_value)
    X_train_common[:, s] = missing_value

    X_test_common = np.zeros((Ntrain, D))
    mean_values = np.mean(X_train, axis=0)  # 1 x D array
    for k in range(Ntrain):
        X_test_common[k, :] = X_test_instance
        s = np.where(X_train[k, :] == missing_value)
        if len(s[0]) != 0:
            X_test_common[k, s] = mean_values[s]

    dist = np.matrix(np.sqrt(np.sum(np.square(X_train_common - X_test_common),
            axis=1))).T
    return dist

def kNNMatrixCompletion(X_train, X_test, K, missing_value,
                        use_softmax_weights=True, binarize=False):
    Ntest = X_test.shape[0]

    X_test_predicted = np.array(X_test)
    for i in range(Ntest):
        print('data%i' % i)
        X_test_i = np.squeeze(np.array(X_test[i, :]))

        distances = sqdist(X_train, X_test_i, missing_value)

        # sort the distances in ascending order
        # and store the indices in a variable
        closest_data_indices = \
```

Christos Kormaris

```
            (np.argsort(distances, axis=0)).tolist()

        # select the top k indices
        closest_k_data_indices = closest_data_indices[:K]
        closest_k_data = \
            np.squeeze(X_train[closest_k_data_indices, :])

        closest_k_data_distances = \
                np.squeeze(distances[closest_k_data_indices, :]).T

        if use_softmax_weights:
            # distribute weights, in the following manner:
            # wk = e^(-k) / sum_{i=1}^{K} e^(-i)
            # such that: w1 + w2 + ... wK = 1
            a = 1
            weights = softmax(-a * closest_k_data_distances)
        else:
            # ALTERNATIVE: all weights equal to 1 / K
            weights = np.ones((K, 1)) / float(K)

        closest_k_data = np.multiply(closest_k_data, weights)

        # sum across all rows for each column, returns a column vector
        predicted_pixels = np.sum(closest_k_data, axis=0).T

        X_test_predicted[i, np.where(X_test_i == missing_value)] = \
            predicted_pixels[np.where(X_test_i == missing_value)].T

    if binarize:
        X_test_predicted = np.round(X_test_predicted + 0.3)

    return X_test_predicted
```

Listing C.7: K-NN missing values completion algorithm in Python

C.5 VAE Missing Values Completion Algorithm in PyTorch

Here's the full code for the VAE missing values completion algorithm in Python, using **PyTorch**.

```
import numpy as np

X_train  # matrix with the original data
y_train  # labels of the matrix "X_train"
X_train_missing  # matrix with missing values
X_train_masked  # matrix with 0s where the pixel are missing and
                # 1s where the pixel are not missing
X_filled = np.array(X_train_missing)  # the final matrix of predicted pixels
N  # the number of training examples
input_dim  # the input dimensionality D
hidden_encoder_dim  # number of neurons in the encoder
hidden_decoder_dim  # number of neurons in the decoder
latent_dim  # Z_dim
epochs  # the number of total epochs for the training
batch_size  # the number of data trained in each iteration
learning_rate  # the rate of learning "eta" for the optimization algorithm

# initialize_weights:
# initializes the weights of the encoder and the decoder.
# params: the weights of the encoder and the decoder
# solver: the optimizer used for training (e.g SGD)
params, solver = initialize_weights(input_dim, hidden_encoder_dim,
                    hidden_decoder_dim, latent_dim, lr)

for epoch in range(epochs):
    iterations = int(N / batch_size)

    for i in range(iterations):
        start_index = i * batch_size
        end_index = (i + 1) * batch_size

        batch_data = X_filled[start_index:end_index, :]
        batch_labels = y_train[start_index:end_index]
        masked_batch_data = X_train_masked[start_index:end_index, :]

        cur_samples, cur_elbo = train(batch_data, batch_size,
                                      latent_dim, params, solver)

        cur_samples = np.multiply(masked_batch_data, batch_data) +
                      np.multiply(1 - masked_batch_data, cur_samples)
        X_filled[start_index:end_index, :] = cur_samples
```

Listing C.8: VAE missing values completion algorithm in PyTorch

NOTES:

- The functions "initialize_weights" and "train" are the same ones used inside this Appendix.
- The implementation for VAE missing values completion algorithm in Python, using **TensorFlow**, is not included because it is similar to the PyTorch implementation.

Bibliography

[1] Yoshua Bengio. Learning deep architectures for ai. *Foundations and Trends® in Machine Learning*, 2(1):1–127, 2009. ISSN 1935-8237. doi: 10.1561/2200000006. http://dx.doi.org/10.1561/2200000006.

[2] Carl Doersch. Tutorial on variational autoencoders. *Variational Autoencoders*, June 2016. https://arxiv.org/abs/1606.05908.

[3] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *Variational Autoencoders*, December 2013. https://arxiv.org/abs/1312.6114.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. *Deep Learning*, 2016. http://www.deeplearningbook.org.

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

[6] Alex Krizhevsky. Learning multiple layers of features from tiny images. *Machine Learning*, pages 34–37, April 2009.

[7] A.S. Georghiades, P.N. Belhumeur, and D.J. Kriegman. From few to many: Illumination cone models for face recognition under variable lighting and pose. *IEEE Trans. Pattern Anal. Mach. Intelligence*, 23(6):643–660, 2001.

[8] Christopher M. Bishop. Pattern recognition and machine learning. *Machine Learning*, 2006.

[9] Ioannis Kontoyiannis and Stavros Toumpis. Academic notes for the course of "probabilities". *Athens University of Economics & Business*, 2011.

Links

- Google Images https://images.google.com/
- Analytics Vidhya,
  An Introduction to Implementing Neural Networks using TensorFlow
  https://www.analyticsvidhya.com/blog/2016/10/an-introduction-to-implementing-neural-networks-using-tensorflow/
- Arxiv Insights YouTube channel,
  Variational Autoencoders
  https://www.youtube.com/watch?v=9zKuYvjFFS8
- Augustinus Kristiadi's Blog, Many flavors of Autoencoder
  https://wiseodd.github.io/techblog/2016/12/03/autoencoders/
- Christopher Bourez's blog,
  Symbolic computing and deep learning tutorial with Tensorflow/Theano: learn basic commands of 2 libraries for the price of 1
  http://christopher5106.github.io/big/data/2016/03/06/symbolic-computing-and-deep-learning-tutorial-on-theano-and-google-tensorflow.html
- Edureka YouTube channel,
  Introduction To TensorFlow | Deep Learning Using TensorFlow | TensorFlow Tutorial | Edureka
  https://www.youtube.com/watch?v=uh2Fh6df7Lg&t=1562s
- Fast Forward Labs Blog,

  1. Introducing Variational Autoencoders (in Prose and Code)
     http://blog.fastforwardlabs.com/2016/08/12/introducing-variational-autoencoders-in-prose-and.html

  2. Under the Hood of the Variational Autoencoder (in Prose and Code)
     http://blog.fastforwardlabs.com/2016/08/22/under-the-hood-of-the-variational-autoencoder-in.html

- Jaan Altosaar, Tutorial - What is a Variational Autoencoder?
  https://jaan.io/what-is-variational-autoencoder-vae-tutorial/

- kevin frans, Variational Autoencoders Explained
  http://kvfrans.com/variational-autoencoders-explained/
- Nando de Freitas YouTube channel,
  Deep Learning Lecture 14: Karol Gregor on Variational Autoencoders and Image Generation
  https://www.youtube.com/watch?v=P78QYjWh5sM
- Siraj Raval YouTube channel,
  How to Generate Images - Intro to Deep Learning #14
  https://www.youtube.com/watch?v=3-UDwk1U77s
- studio otoro, Generating Large Images from Latent Vectors
  http://blog.otoro.net/2016/04/01/generating-large-images-from-latent-vectors/
- The Keras Blog, Building Autoencoders in Keras
  https://blog.keras.io/building-autoencoders-in-keras.html
- videolectures.net,
  Building Machines that Imagine and Reason: Principles and Applications of Deep Generative Models
  http://videolectures.net/deeplearning2016_mohamed_generative_models/
- videolectures.net,
  Variational Autoencoder and Extensions
  http://videolectures.net/deeplearning2015_courville_autoencoder_extension/
- Xitong Yang's Blog, Understanding the Variational Lower Bound
  https://xyang35.github.io/2017/04/14/variational-lower-bound/
- wiseodd GitHub, Generative Models
  https://github.com/wiseodd/generative-models

- Wikipedia,

1. Autoencoder
   https://en.wikipedia.org/wiki/Autoencoder
2. Bayes' theorem
   https://en.wikipedia.org/wiki/Bayes'_theorem
3. Softmax function
   https://en.wikipedia.org/wiki/Softmax_function
4. Cross entropy
   https://en.wikipedia.org/wiki/Cross_entropy
5. Entropy (information theory)
   https://en.wikipedia.org/wiki/Entropy_(information_theory)
6. Jensen's inequality
   https://en.wikipedia.org/wiki/Jensen's_inequality
7. Mean absolute error
   https://en.wikipedia.org/wiki/Mean_absolute_error
8. Mean squared error
   https://en.wikipedia.org/wiki/Mean_squared_error
9. Kullback–Leibler divergence
   https://en.wikipedia.org/wiki/Kullback-Leibler_divergence
10. Occam's Razor
    https://en.wikipedia.org/wiki/Occam's_razor
11. Root-mean-square error
    https://en.wikipedia.org/wiki/Root-mean-square_deviation
12. TensorFlow
    https://en.wikipedia.org/wiki/TensorFlow
13. Variational Bayesian methods
    https://en.wikipedia.org/wiki/Variational_Bayesian_methods