Variational Autoencoders & Applications

by Student: Christos KORMARIS
Supervisor Professor: Dr. Michalis TITSIAS

Outline

- Introduction
- Machine Learning Tools & Libraries
- Theoretical Background
- Pros and Cons of VAEs
- The VAE Loss Function (ELBO)
- How VAEs Work
- The TensorBoard
- K-NN Missing Values Algorithm
- VAE Missing Values Algorithm
- Datasets
- Experimental Results

Machine Learning Tools & Libraries

Programming Language: **Python 3**
Python IDE: **JetBrains PyCharm**
Thesis Editor: **ShareLaTeX/Overleaf**

Machine Learning libraries:

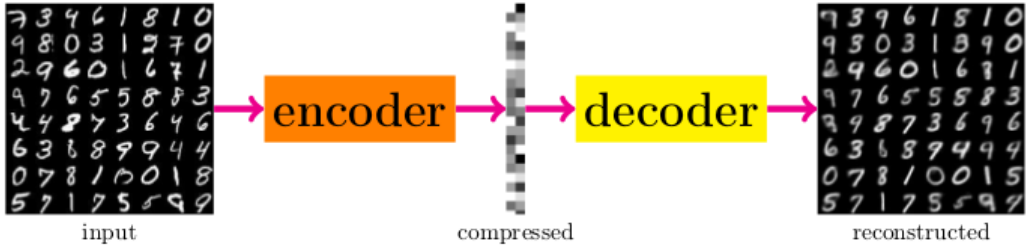
1) TensorFlow
2) PyTorch
3) Keras
4) Matplotlib
5) Scipy
6) Pandas
7) Numpy

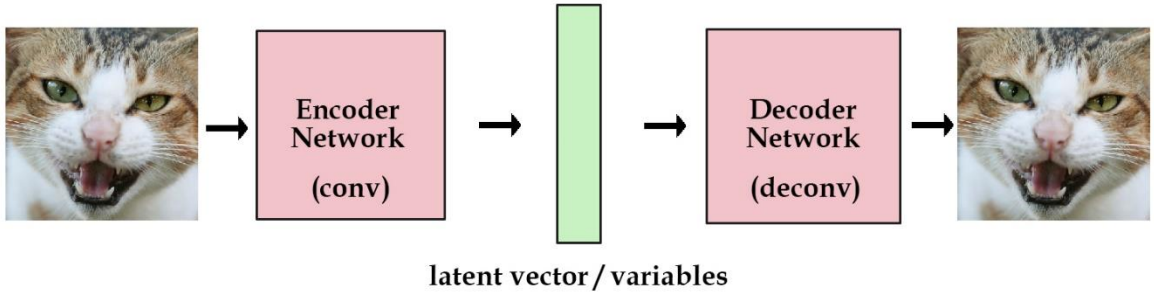
Introduction

A variational autoencoder is a method that can produce artificial data which will resemble a given dataset of real data. **Why?** For many reasons, such as to increase the size of datasets. Another usage of VAEs is dimensionality reduction.

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**. 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.
A Variational Autoencoder is a sequence of two neural network one after the other. The first neural network is called the **encoder.** The second neural network is called the **decoder**.
The encoder and the decoder are trained simultaneously.
The VAE runs dimensionality reduction on the initial data, by compressing them into latent variables.

VAEs Schema


encoder
decoder
input
compressed
reconstructed


Encoder Network
(conv)
Decoder Network
(deconv)
latent vector / variables

Variational Inference

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

$$P(Z \mid X) = \frac{P(X \mid Z) \cdot P(Z)}{P(X)} = \frac{P(X \mid Z) \cdot P(Z)}{\int P(X, Z)\, dz} \quad posterior = \frac{likelihood \cdot prior}{normalizing\ constant}$$

Where posterior: P(Z|X), likelihood: P(X|Z), prior: P(Z), normalizing constant: P(x)

We can calculate the normalizing constant (aka **evidence**) by marginalizing out the latent variables Z:

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

However, calculating this integral requires exponential time, because the distribution of the latent variables Z is continuous. We will use a method called **variational inference**. Thus, the term P(Z|X) will be named to Qφ(Z|X), when using this method.

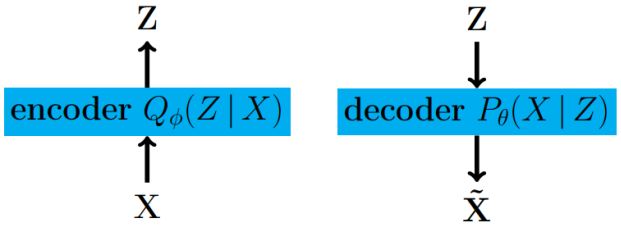
Benefits of Variational Inference & Drawback of VAEs

Benefits of Variational Inference

1) **Intractability:** The calculation of the term P(X) is intractable and so is the term P(Z|X). Thus, the Expectation-Maximization (EM) algorithm and other similar 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).

Main Drawback of VAEs

- **Blurry Images:** The reason remains to be researched.



The Variational Lower Bound (ELBO)

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

The whole term L(X, Q) is called **Variational Lower Bound** or **ELBO**.
The term E [$\log P_\theta$ (X|Z)] is called **reconstruction cost**.
The term $D_{KL}[Q_\varphi(Z)||P(Z)]$ is called **penalty** or **regularization term**.

The penalty term ensures that the explanation of the data **Q(Z|X) ≈ Q(Z)** doesn't deviate too far from the beliefs term P(Z). This penalty term helps us apply Occam's Razor to our inference model. The KL divergence is always greater or equal to 0 and so it can be omitted.

The KL-divergence formula between two distributions P and Q is the following. It is important to note that the KL-divergence is asymmetric, thus $D_{KL}[P||Q] \neq D_{KL}[Q||P]$:

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

Optimizing the ELBO

I will try to obtain an analytical formula to calculate the KL-divergence term, $D_{KL}[Q_\varphi (Z) \,||\, P (Z)]$, of the ELBO:

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

For the dimensionality of the latent variable Z parameter J=1, which means univariate Gaussian distributions, we end up with the formula:

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

The Reparameterization Trick

I want to calculate the gradient of the **mean of the ELBO** using back-propagation:

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

The calculation of this gradient reduces to the calculation of the gradient of the much simpler equation:

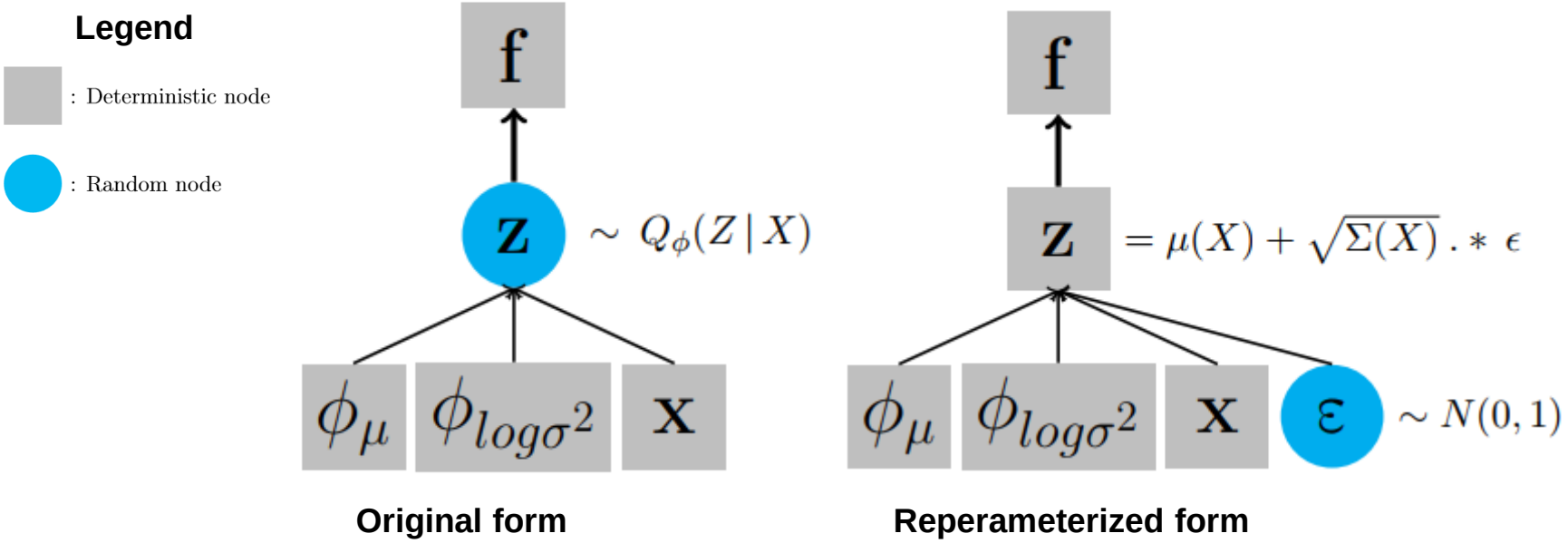
$$logP_\theta\,(X|Z) - D_{KL}[Q_\phi(Z)\,||\,P(Z)]$$

We can make this calculation possible using a method called "**reparameterization trick**". I sample Z using a linear transformation of Gaussians. We have a normal distribution **N(μ(X), Σ(X))** and **ε~N(0, I)**. The result is again a Gaussian distribution **Z~N(μ(X), Σ(X))**.

$$Z = \mu(X) + \sqrt{\Sigma(X)}\,.*\,\epsilon$$

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

The Reparameterization Trick (part 2)



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

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

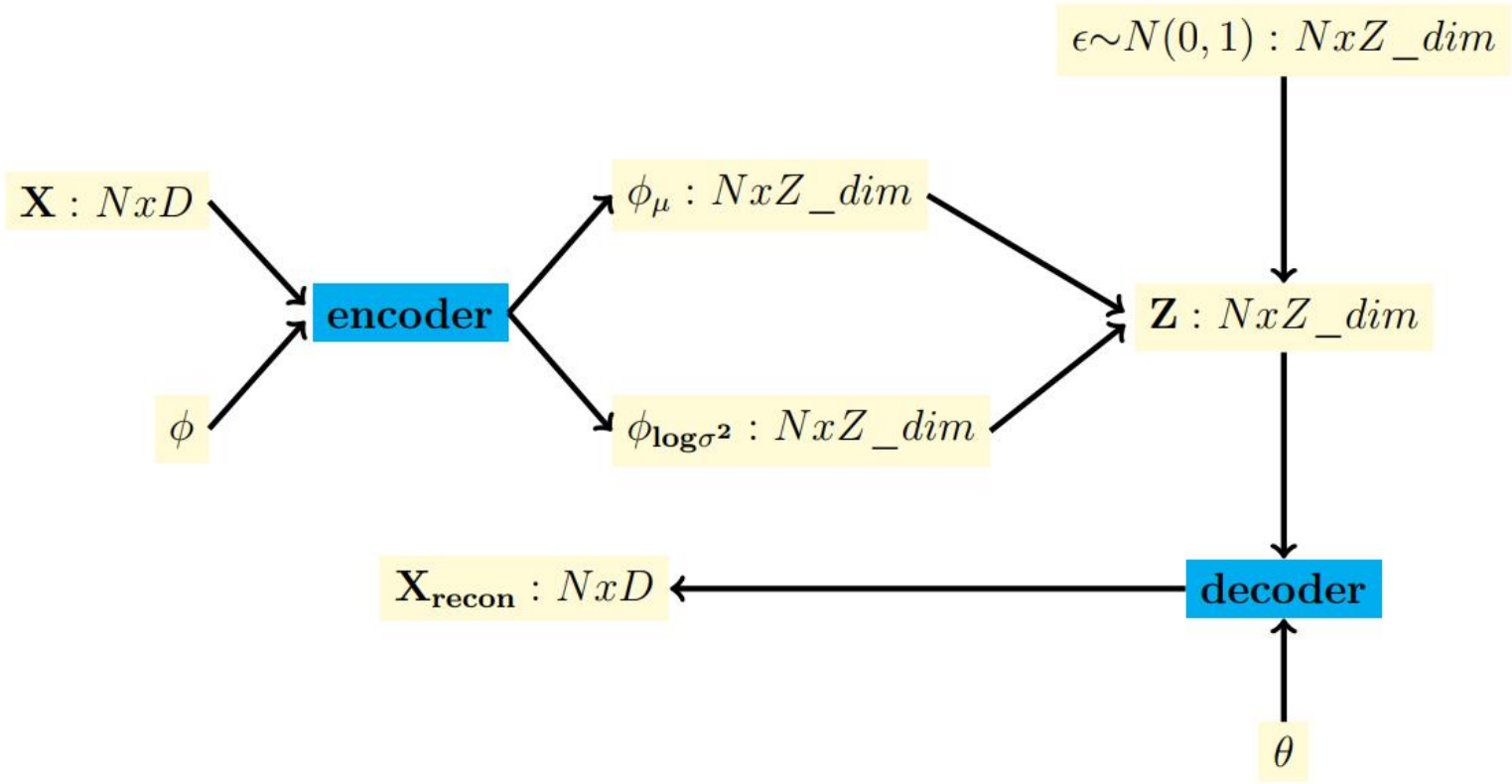
TensorFlow back-propagation

```
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)
```

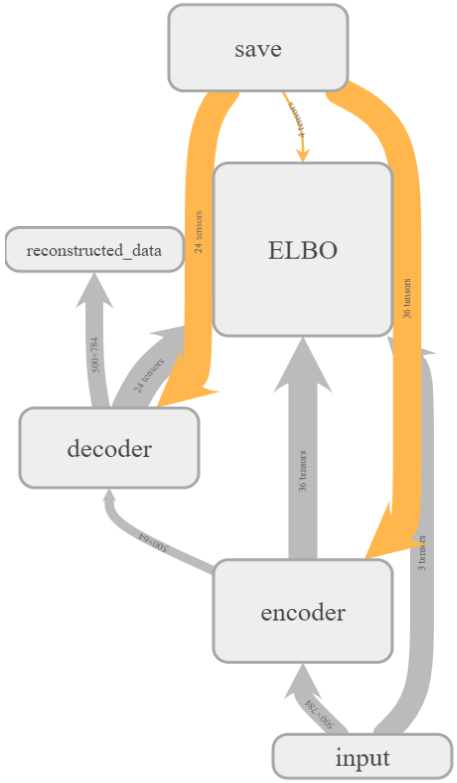
PyTorch back-propagation

```
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_()
```

The whole VAE Process



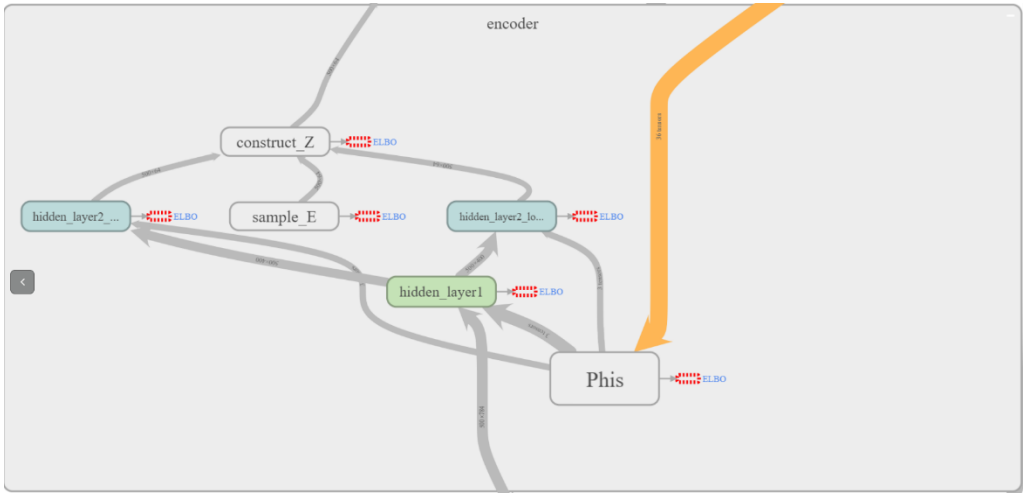
The TensorBoard



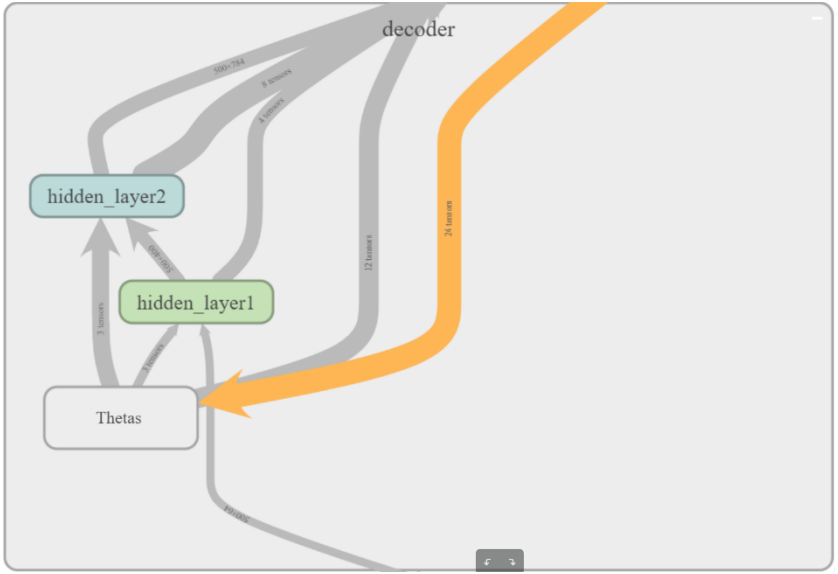
This is a visualization of the TensorFlow implementation, with TensorBoard.

The TensorBoard (part 2)

Encoder



Decoder



This is an enlarged visualization of the encoder and the decoder scopes of the TensorFlow implementation, with TensorBoard.

K-NN Missing Values Algorithm

I used a modified K-NN algorithm for regression to predict missing values in images, thus I predict missing pixels.
For each test example iteratively, I selected the K closest data (images) that the algorithm will take into consideration.
For the selection of the K closest distances I modified the data, in a way that the train data with missing values are not preferred. Then I calculated the distances using the Euclidean formula:

$$\sqrt{\sum_{j=1}^{D} (X\_test\_common_{ij} - X\_train\_common_{ij})^2}$$

The one nearest neighbor corresponds to K=1.
I have also tested the algorithm for K=3, K=10 and K=100.

K-NN Missing Values Algorithm (part 2)

Then, I distributed softmax weights depending on the distances of the K closest data. The closest example gets the biggest weight value. Each pixel of the ***k-th*** train example is being multiplied with its corresponding weight. The weight values are assigned as follows:

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

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

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

where *dk* denotes the distance from the ***k-th*** closest train example

VAE Missing Values Algorithm

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, I constructed the dataset of missing values, **X_train_missing**, from the original dataset **X_train**.

VAE Missing Values Algorithm (part 2)

After I had constructed the dataset with missing values, **X_train_missing**, I needed to construct a matrix with binary values (0 or 1), which would store the information of which parts of the original images I had replaced with missing values. I called this matrix **X_train_masked**. If a pixel in an image did not get replaced, the corresponding pixel in the X_train_masked matrix would be 1. If a pixel in an image did get replaced by the "missing value", the corresponding pixel in the **X_train_masked** matrix would be 0.
Furthermore, I needed to define the matrix that would contain the data with the predicted values of the missing pixels. I called this matrix **X_filled**. I initialized the matrix **X_filled** to be the same as **X_train_missing**.

VAE Missing Values Algorithm pseudocode

```
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
```

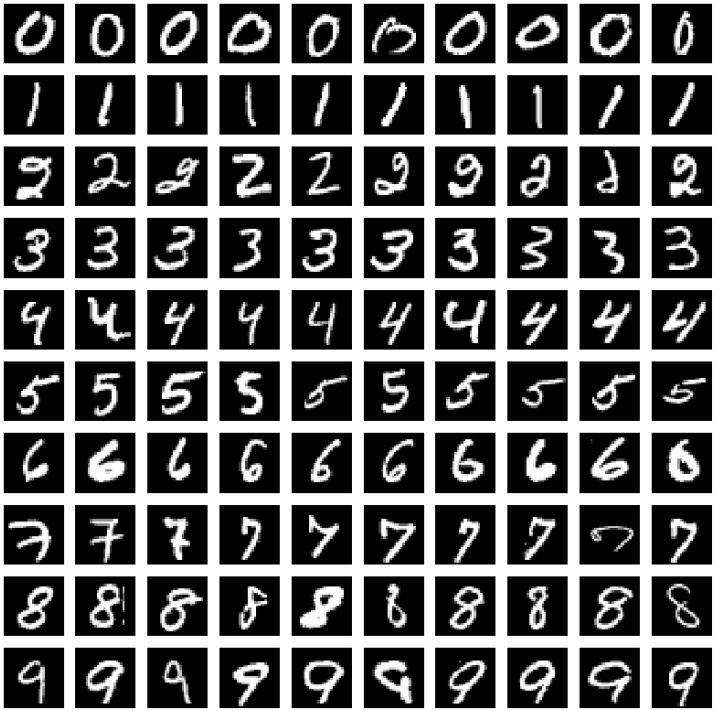
Datasets

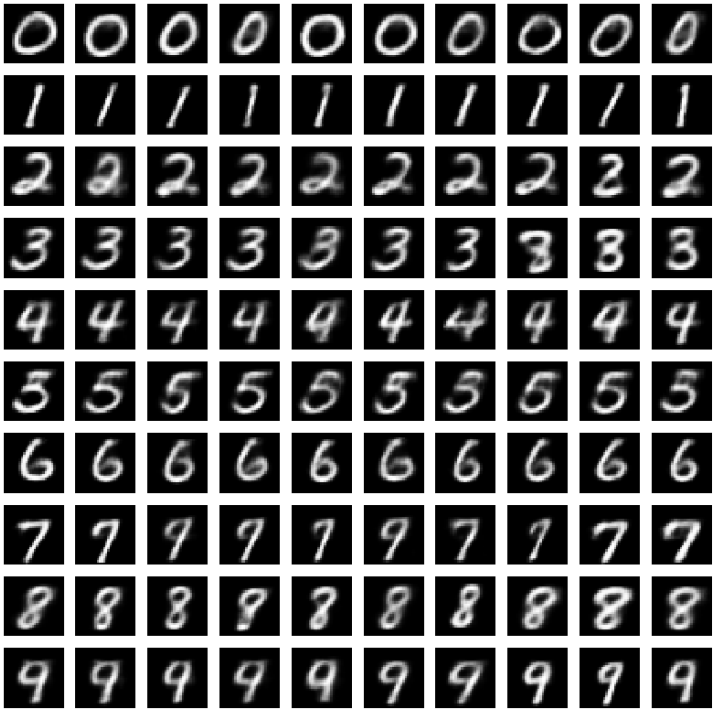
Datasets	Train examples	Test examples	Validation examples
MNIST	55000	10000	5000
Binarized MNIST	50000	10000	10000
CIFAR-10	50000	10000	-
OMNIGLOT	390 [En] / 130 [Gr]	360 [En] / 120 [Gr]	-
Cropped YALE Faces	2442	-	-
The Database of Faces	400	-	-
MovieLens 100k	90570	9430	-

Datasets

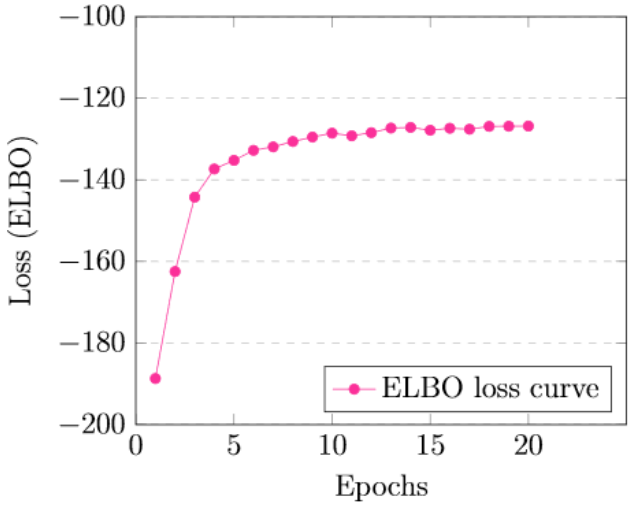
Datasets	# Classes	# Dimensions	Type of Values
MNIST	10	28x28 pixels	real values in [0, 1]
Binarized MNIST	10	28x28 pixels	{0, 1}
CIFAR-10	10	32x32x3 [RGB] / 32x32x1 [Grayscaled] pixels	real values in [0, 1]
OMNIGLOT	26 [En] / 24 [Gr]	32x32 pixels	real values in [0, 1]
Cropped YALE Faces	38	168x192 pixels	real values in [0, 255]
The Database of Faces	40	92x112 pixels	real values in [0, 255]
MovieLens 100k	943 users	1982 movies	{1, 2, 3, 4, 5}

Experimental Results, VAE on the MNIST Dataset





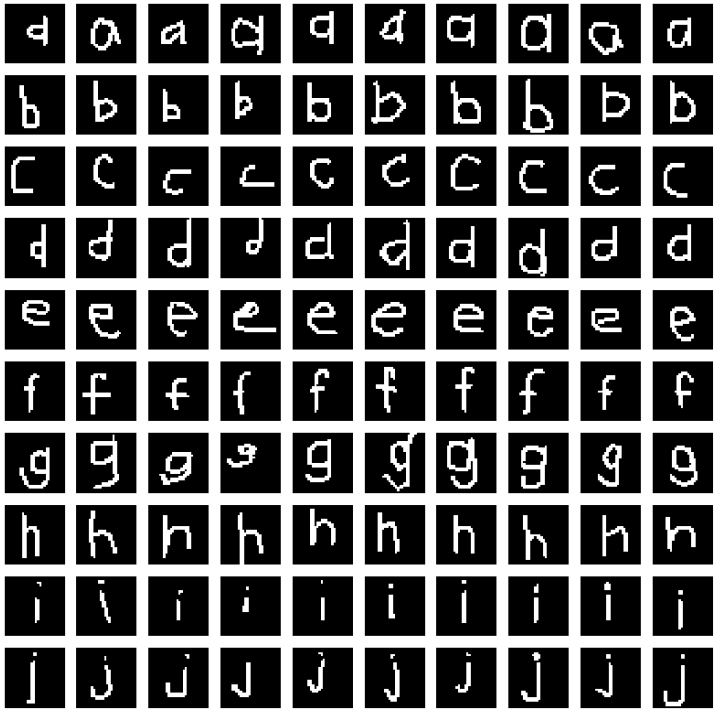
Experimental Results, VAE on the MNIST Dataset (metrics)

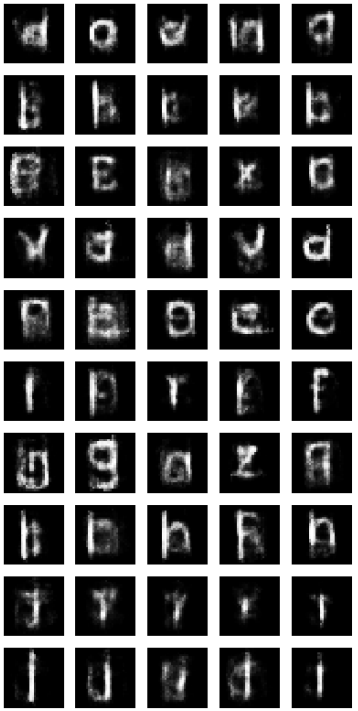


Metric	Value
root mean squared error (RMSE)	0.17121448655010216
mean absolute error (MAE)	0.07299246278044173

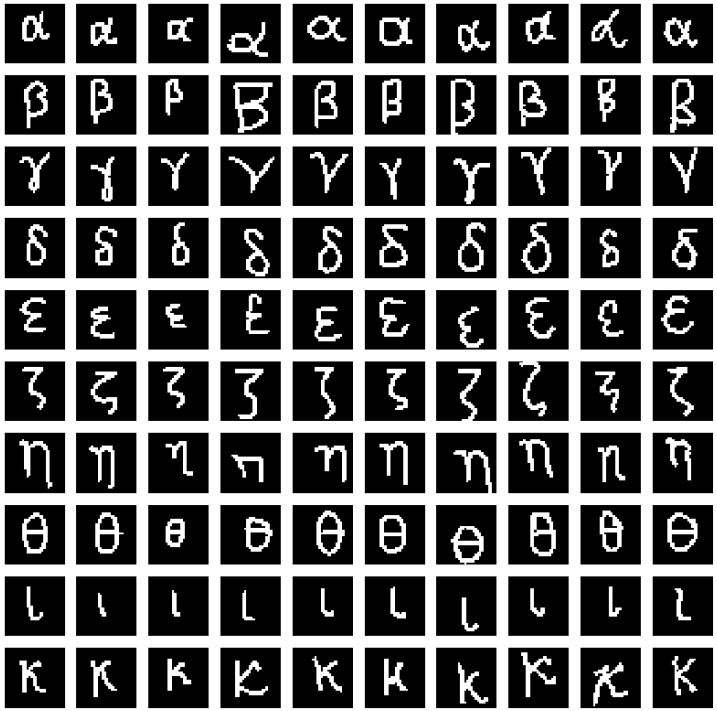
VAE on the MNIST dataset in TensorFlow.

Experimental Results, VAE on the OMNIGLOT English Dataset



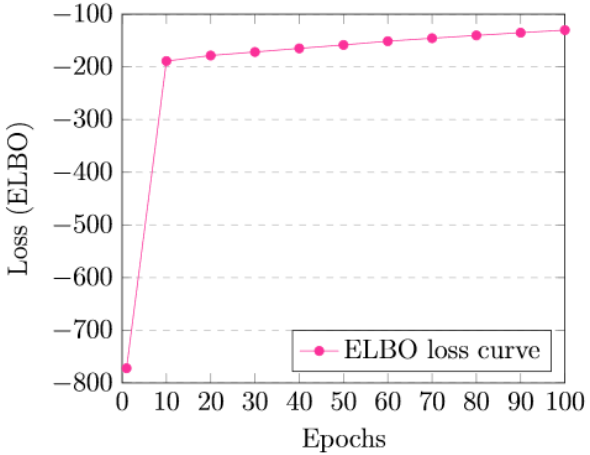


Experimental Results, VAE on the OMNIGLOT Greek Dataset



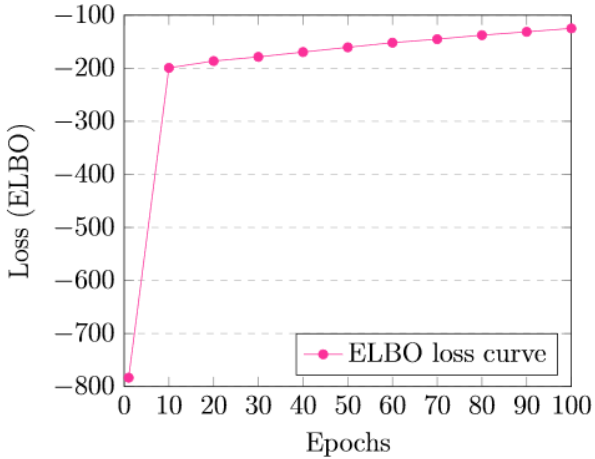
Experimental Results, VAE on the OMNIGLOT English/Greek Datasets (Metrics)

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



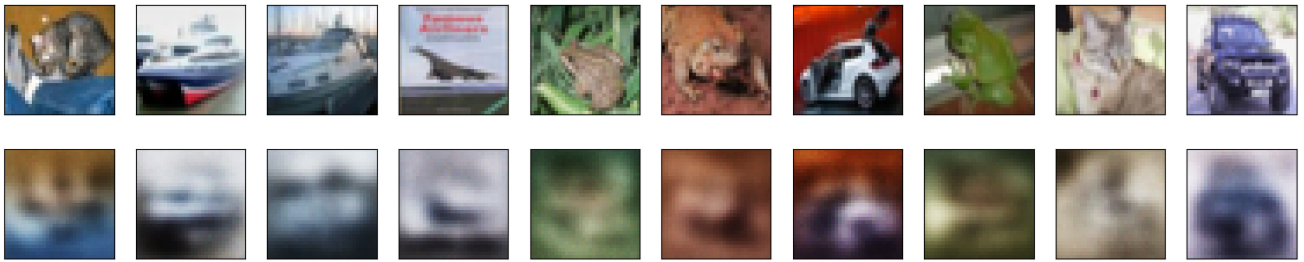
Metric	Value
root mean squared error (RMSE)	0.2134686176531402
mean absolute error (MAE)	0.09105986614619989

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

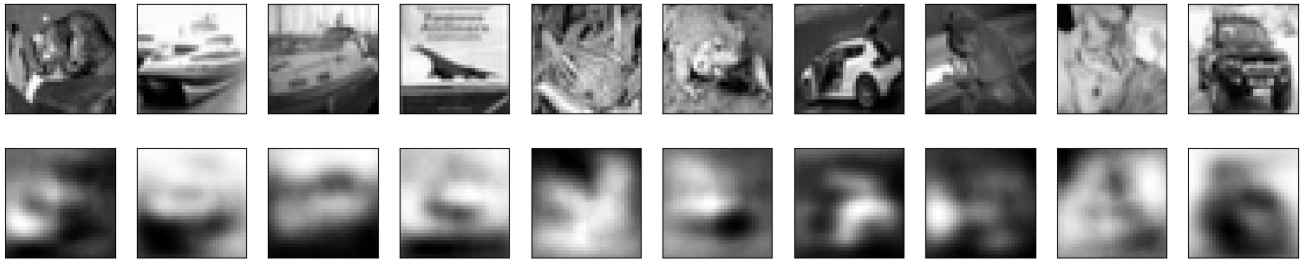


Metric	Value
root mean squared error (RMSE)	0.20157381435292054
mean absolute error (MAE)	0.20157381435292054

Experimental Results, VAE on the CIFAR-10 Dataset



CIFAR-10 RGB



CIFAR-10 Grayscale

Experimental Results, VAE on the CIFAR-10 Dataset (Metrics)

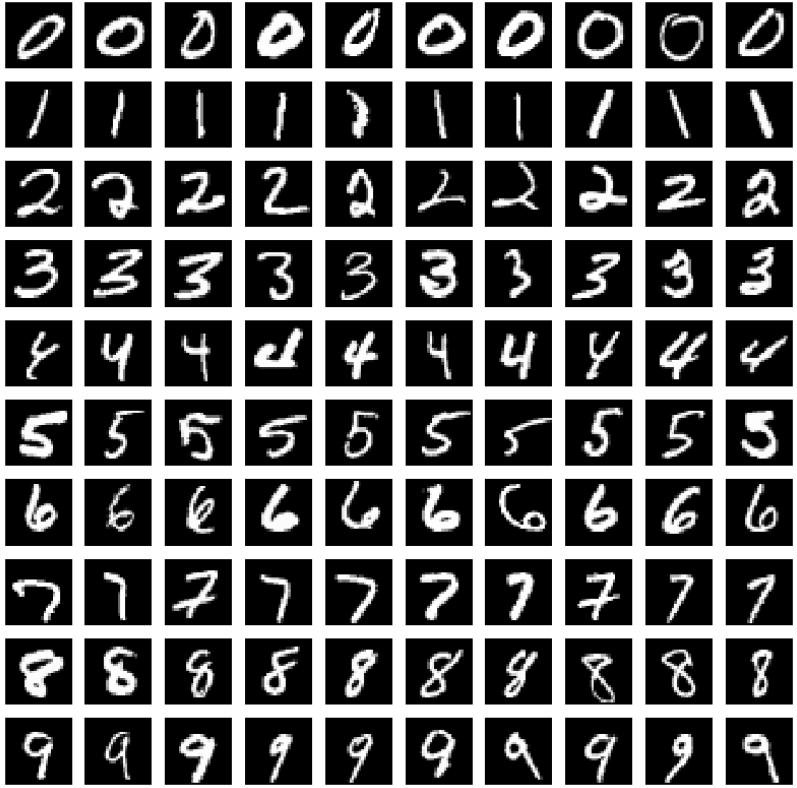
Metric	Value
last epoch loss (ELBO)	0.6198
root mean squared error (RMSE)	0.17352526
mean absolute error (MAE)	0.1368697

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

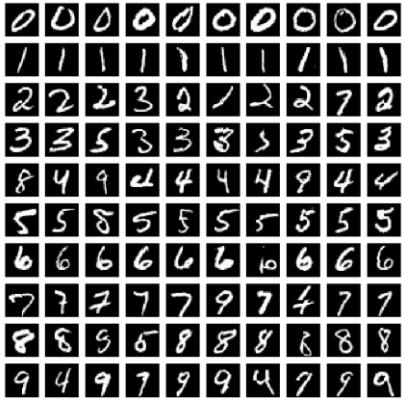
VAE on the CIFAR-10 Grayscale dataset, in Keras.

Experimental Results, Missing Values Completion Algorithm on the MNIST Dataset, Original Test Data

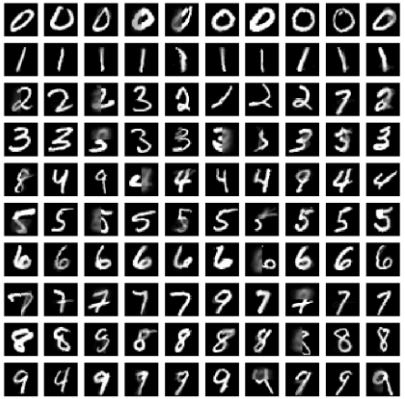


Experimental Results, Missing Values Completion Algorithms on the MNIST Dataset

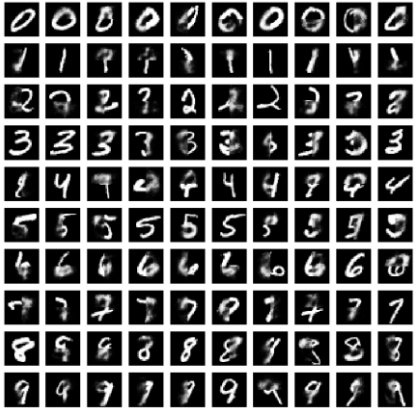
Test Data with Structured Missing Values



1-NN

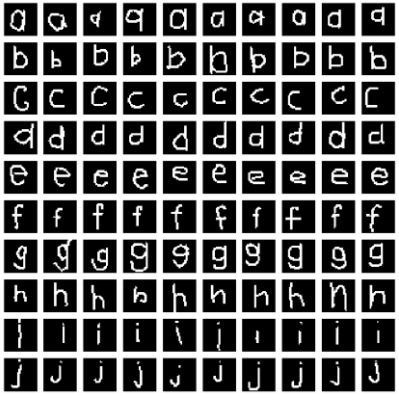


100-N

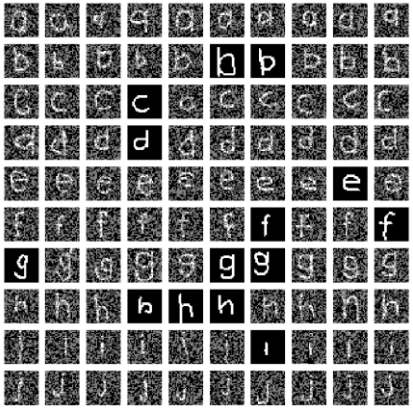


VAE in PyTorch Epoch 200

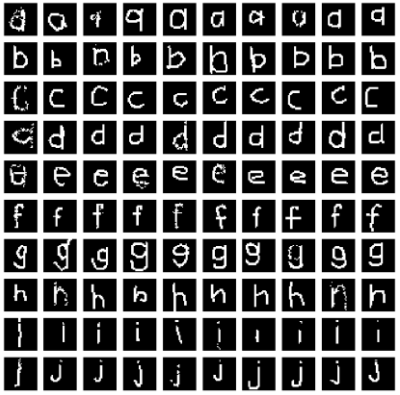
Experimental Results, Missing Values Completion Algorithms on the OMNIGLOT English Dataset



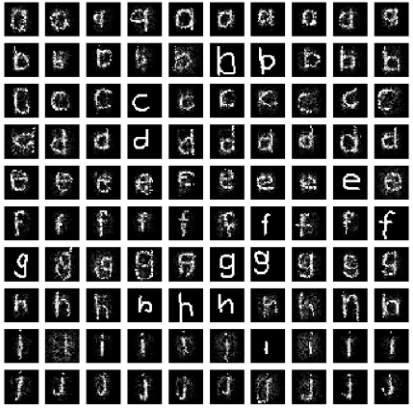
Original Data



Data with Missing at Random Values

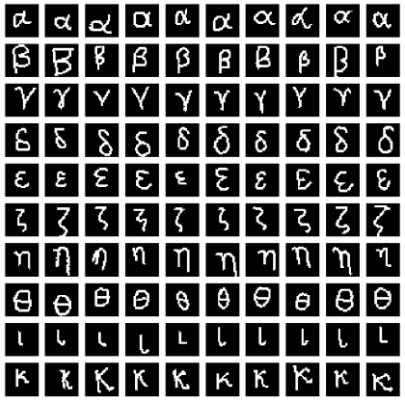


1-N

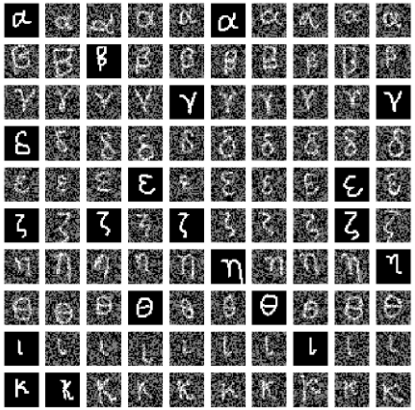


VAE in PyTorch Epoch 100

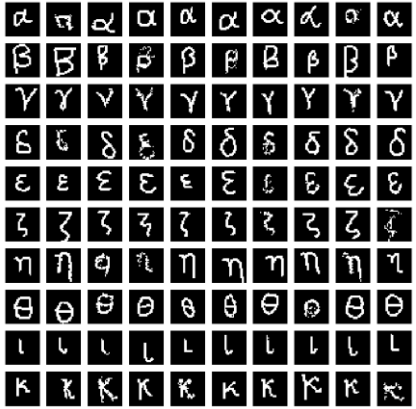
Experimental Results, Missing Values Completion Algorithms on the OMNIGLOT Greek Dataset



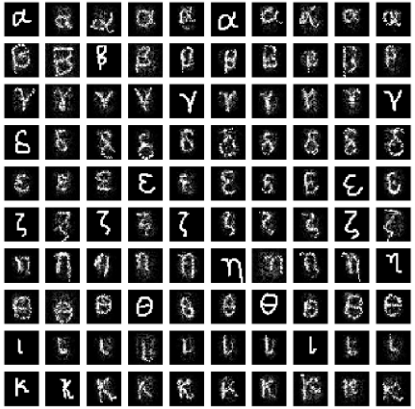
Original Data



Data with Missing at Random Values



1-N



VAE in PyTorch Epoch 100

Experimental Results, VAE Missing Values Completion Algorithms (Metrics)

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

Missing Values completion algorithms on the MNIST dataset.

Method	RMSE	MAE	Time
1-NN	0.089003	0.161780	8.64 sec
VAE	0.160895	0.050487	254.48 sec

Missing Values completion algorithms on the OMNIGLOT English dataset.

Method	RMSE	MAE	Time
1-NN	0.084689	0.1663	9.37 sec
VAE	0.162737	0.052432	245.66 sec

Missing Values completion algorithms on the OMNIGLOT Greek dataset.

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

We have run both K-NN and VAE in TensorFlow Missing Values algorithms on the MovieLens Dataset and compared the predicted ratings. The results are the following:

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

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

THANK YOU!

https://bitbucket.org/lptamenos/variational-autoencoder-thesis