STATISTICAL SCIENCE DEPARTMENT, DUKE UNIVERSITY

Auto-Encoding Variational Bayes Implementation

Yizi Zhang and Chenxi Wu
April 30, 2020

ABSTRACT

We implement a generative model for naturalistic image synthesis using variational auto-encoders (VAE). VAE is a popular approach to unsupervised learning of complicated distributions. We build the VAE model on top of standard function approximators (neural networks), and train it with Adam optimizer. In this report, we explain the theory behind VAE and describe its empirical behaviors on various simulated and real datasets. To evaluate the model performance, we further compare VAE to other generative models. Our implementation shows promising results in simulation study and application to benchmark datasets, but also expose weaknessess in dealing with more graphically complex data.

1 INTRODUCTION

This project is a Python implementation of Auto-Encoding Variational Bayes method by Kingma and Welling [1]. In statistical learning, we often encounter continuous latent variables with intractable posterior distributions and large datasets. How to perform approximate inference on the latent variables with intractable posterior has become crucial. One solution is to use Variational Bayes (VB) to approximate intractable posteriors to make inference possible. However, this method requires analytical solutions to the expectations of the posterior which could also be intractable. So Kingma and Welling [1] proposed a Stochastic Gradient Variational Bayes method called Auto-Encoding Variational Bayes (VAE) to efficiently approximate posterior inference.

2 Background

In simple terms, VAE is a generative neural network which can map the input vector in a low-dimensional space to real data in a high-dimensional space. Thus we can generate quantity of interest, such as different numbers, faces, cartoon avatars, etc.

For example, given a series of cat photos, we hope to generate a new cat photo corresponding to an n-dimensional vector we randomly input. To accomplish this task, we need to learn the connection between the known photos and vectors and use this "connection" to generate new photos according to some random input. In VAE model, this n-dimensional vector represents n invisible factors that determine the appearance of the final cat face. The known cat photos compose our training data $X_1, X_2, ..., X_n$ which are i.i.d. samples in a high-dimensional space. If we can estimate $P(X)$ based on these values, we would obtain a wonderful generative model from which we can sample as much values as we want. Indeed it is hard to realize, so we set a latent variable Z to represent the invisible factors that determine the characteristics of these photos. For each factor, there is a corresponding distribution which we assume to be a Gaussian distribution. Sampling from these distributions can recover the final cat face through a deep network. We refer to this process as "encode" (transform x into latent variable z) and "decode" (use z to reproduce x), which is shown in Figure 1.

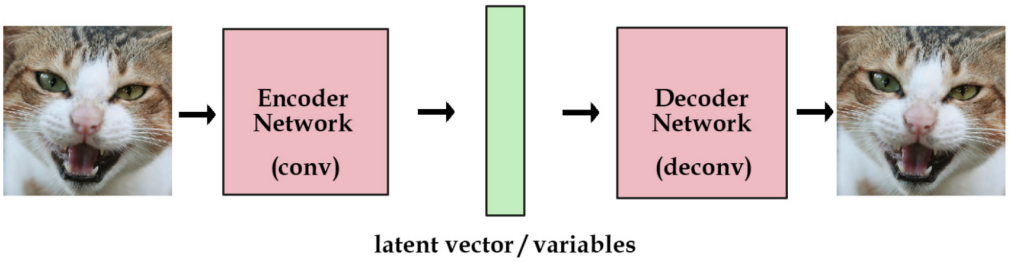


Figure 2.1: Illustration of VAE Model

3 Theory behind VAE

The process of generating data consists of two steps: first generate a value of z_i from some prior distribution $p_\theta(z)$, then generate a value x_i from some conditional distribution $p_\theta(x \mid z)$. Noting that the true parameters θ and z are unknown. In the setting above, we aim at three objectives:

- Estimate ML(maximum likelihood) or MAP(maximum a posteriori) of parameter θ.
- Approximate posterior inference of the latent variable Z given an observed value x for a choice of θ, namely $p_\theta(z \mid x)$
- Approximate marginal inference of X, namely $p(x)$.

Since the true posterior $p_\theta(z \mid x)$ is usually intractable, the author proposes a recognition model $q_\phi(z \mid x)$ to approximate the true posterior. From a coding theory perspective, we refer to $p_\theta(x \mid z)$ as decoder since it reproduce values of x given code z. In a similar manner, $q_\phi(z \mid x)$ is referred as encoder since it produces the distribution of z given the data point x.

The objective of the training process is to maximize the global log-likelihood, which is

$$max\mathscr{L} = \sum_{i=1}^{n} \log p_\theta(x_i) \tag{3.1}$$

Since data points are i.i.d. samples, maximizing $\mathscr{L}$ is equivalent to maximizing the likelihood for every individual x. The marginal log likelihood of an individual data point x_i can be defined as follows:

$$\log p_\theta(x_i) = D_{KL}(q_\phi(z \mid x_i) \| p_\theta(z \mid x_i)) + \mathscr{L}(\theta, \phi; , x_i) \tag{3.2}$$

$D_{KL}(q_\phi(z \mid x_i)$ is the KL divergence of the approximate from the true posterior, which is non-negative, So $log\, p_\theta(x_i) \geq \mathscr{L}(\theta, \phi; , x_i)$. We denote $\mathscr{L}(\theta, \phi; , x_i$ as lower bound $\mathscr{L}_b$. The aim changes into maximizing this lower bound with regard to variational parameter ϕ and generative parameter θ. $\mathscr{L}_b$ can be further decomposed as follows:

$$\begin{aligned}
\mathscr{L}(\theta, \phi; x_i) &= \int q_\phi(z \mid x_i) \log(\frac{p(z, x_i)}{q_\phi(z \mid x_i)}) dz && (3.3)\\
&= \int q_\phi(z \mid x_i) log(\frac{p(z)}{q_\phi(z \mid x_i)}) dz + \int q_\phi(z \mid x_i) \log p_\theta(x_i \mid z) dz && (3.4)\\
&= -D_{KL}(q_\phi(z \mid x_i) \| p(z)) + E_{q_\phi(z|x_i)}[\log p_\theta(x_i \mid z)] && (3.5)\\
&= L_1 + L_2 && (3.6)
\end{aligned}$$

L_1 depicts the similarity between $p_\theta(z)$ and the encoder $q_\phi(z \mid x)$. L_2 reflects the quality of the decoder $p_\theta(x \mid z)$). Note that L_1 and L_2 are independent, maximizing $\mathscr{L}_b$ is equivalent to minimizing $D_{KL}(q_\phi(z \mid x) \| p(z))$ and maximizing $E_{q_\phi(z|x)}[log[p_\theta(x \mid z))]$. In other words, $\mathscr{L}_b$ can be maximized by adjusting the encoder $q_\phi(z \mid x_i)$ and decoder $p_\theta(x_i \mid z)$.

Since it is difficult to compute the derivative of the above equation, the author introduces a reparameterization method to substitute $q_\phi(z \mid x)$. Let $z = g_\phi(\epsilon, x)$ with $\epsilon \sim p(\epsilon)$. g_ϕ is some vector-valued function parameratized by ϕ. In this way, random variable z is decomposed into two parts: a definitive part by some function of x and a random part by ϵ. Thus, the conditional distribution of z depends completely on the distribution of ϵ, i.e. $q(z_i \mid x) = p(\epsilon)$

Suppose z follows a multivariate Gaussian distribution $\mathscr{N}(z; 0, I_J)$, where J is the dimension of z. $q_\phi(z \mid x) = \mathscr{N}(z; \mu, \sigma)$. So L_1 can be rewritten as follows (See proof at Appendix B of the original paper):

$$L_1 = \frac{1}{2} \sum_{j=1}^{J} (1 + \log(\sigma_i^j)^2 - (\mu_i^j)^2 - (\sigma_i^j)^2) \tag{3.7}$$

L_2 can be approximated by Monte Carlo simulation, i.e.

$$L_2 = \frac{1}{L} \sum_{l=1}^{L} \log p_\theta(x_i \mid z_i^l) \tag{3.8}$$

Combining the two loss function together, we can obtain the following resulting estimator for data point x_i:

$$\mathscr{L}(\theta,\phi;x_i) = \frac{1}{2}\sum_{j=1}^{J}(1+\log(\sigma_i^j)^2-(\mu_i^j)^2-(\sigma_i^j)^2)+\frac{1}{L}\sum_{l=1}^{L}\log p_\theta(x_i \mid z_i^l) \tag{3.9}$$

4 Baseline Models

Kingma and Welling [1] used 2 fully connected layers for the encoder and the decoder in their paper on VAEs. Mathematically, for the encoder

$$h_z(x) = \tanh(W_1 x + b_1), \tag{4.1}$$
$$\mu_z(x) = W_2 h_z(x) + b_2, \tag{4.2}$$
$$\log\sigma_z^2(x) = W_3 h_z(x) + b_3. \tag{4.3}$$

Then z is generated according to $N(z;\mu_z(x),\mathrm{diag}(\sigma_z^2(x)))$. Afterwards, decoder is

$$h_x(z) = \tanh(W_4 z + b_4), \tag{4.4}$$
$$\mu_x(z) = W_5 h_x(z) + b_5, \tag{4.5}$$
$$\log\sigma_x^2(z) = W_6 h_x(z) + b_6. \tag{4.6}$$

In the original VAE paper [1], Kingma and Welling report results using VAEs trained using MNIST and Frey Face datasets. In this project, We implemented their model and reproduce the results on the MNIST and Frey Face dataset. In addition, we conducted a simulation study on network-valued data and applied our model to the CIFAR-10 data set. We also compared the VAE method to two other generative models.

5 Profiling and Optimization

The basic code was written in Python using package numpy. Please see section 8 for the link to the Git repository and the installation instructions of our package. We profiled the basic version of the code to find the functions or parts of code taking significant amounts of time. The profiler results of training for 1 epoch are shown below. Notice that the optimize and backward propagation functions of the VAE model are the most time-consuming.

```
      427241 function calls (421241 primitive calls) in 1281.958 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   600  810.285    1.350  810.285    1.350 vae_model.py:109(optimize)
   600  462.185    0.770 1275.299    2.125 vae_model.py:146(backward)
     1    3.678    3.678 1281.958 1281.958 vae_mnist_numpy.py:33(train)
  9000    1.774    0.000    1.774    0.000 {method 'dot' of 'numpy.ndarray'}
```

```
   600    1.327    0.002    1.349    0.002  functionals.py:12(BCE)
  1200    1.025    0.001    1.025    0.001  functionals.py:23(sigmoid)
  3600    0.734    0.000    0.881    0.000  functionals.py:45(lrelu)
  1200    0.215    0.000    0.215    0.000  functionals.py:34(relu)
```

Since the optimize and backpropagation functions involve heavy matrix multiplication and mathematical operations on multi-dimensional arrays, optimizing over unit functions is not the best way to speed up the code. Therefore, we seek the help of GPU programming, which is ideal for processing images and Tensorflow, which provides faster optimization algorithms.

5.1 GPU Programming

We tried GPU programming to reduce the total run times. We used the CuPy library, which accelerates NumPy operations with CUDA. In Table 5.1, we see how the CuPy implementation of the code is almost 100 times faster than the original Numpy version. In addition, we used the tensorflow-gpu library to further speed up the tensorflow implementation of the VAE model.

5.2 Tensorflow Implementation

To enhance the performance of the variational auto-encoder, we used the tensorflow library, which is designed for deep learning, to implement the algorithm following the same structure as the one in the NumPy implementation. Tensorflow has optimized algorithms for computing and updating the gradients, which speeds up the training of the auto-encoder networks. Table 5.1 shows that the tensorflow implementation is about 10 times faster than the CuPy version for 1 training epoch. The following empirical studies on the various data sets were done using the tenforflow implementation of the VAE model.

Table 5.1: Comparison of model implementations

Total Runtimes (in sec)	
Pure Numpy	1281.958
Numpy on GPU	31.636
Tensorflow on CPU	4.011
Tensorflow on GPU	2.368

6 Experiments

We empirically evaluated our VAE method via a simulation study and real-world applications.

6.1 Simulation study

We conducted a simulation study to evaluate the performance of the VAE model in accurately reconstructing the network-valued data. We simulated from three network structures: sparse

networks according to the model in [2], community structures under the model of [3], and small-worldness from the model [4]. We simulated 100 networks with 68 nodes for each type by sampling their edges from conditional independent Bernoulli random variables given their corresponding structure-specific edge probability. Figure 6.1 displays some example networks we generated with the three different network structures.

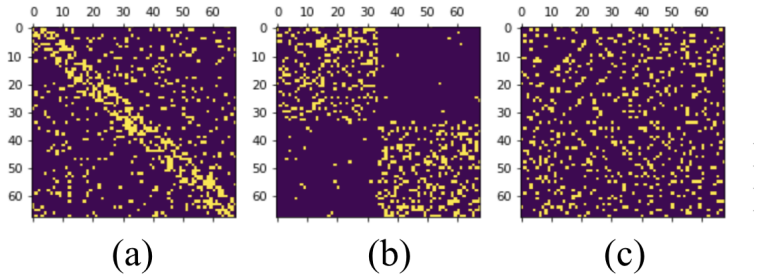


Figure 6.1: Simulated network structures: (a) small world, (b) community structure, (c) sparse graph.

We evaluated the reconstruction accuracy of the VAE model. Figure 6.2 shows the successful reconstruction of the three simulated network structures. We train the VAE model to obtain low-dimensional representations of the simulated networks. We then conducted PCA analysis on the posterior mean and plot the first two PC scores colored with its corresponding network structure category. The separation pattern of the different network structures in the 2-dimensional latent space is shown in Figure 6.2.

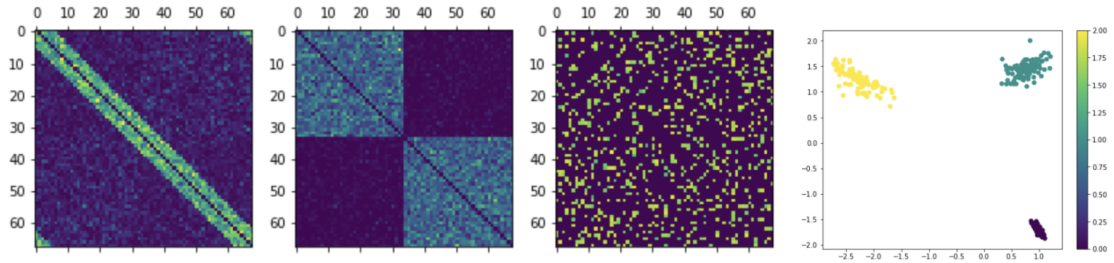


Figure 6.2: Reconstructed network structures and their first 2 PC scores of the posterior mean of $\boldsymbol{z}|\boldsymbol{x}$ corresponding to different network structures.

We summarize the computing details used in the simulation study here. We run stochastic gradient descent with Adam optimizer on the tensorflow implementation of the VAE model with learning rate 0.001 on a CPU device. We used a batch size of 100 and repeated for 200 epochs. The latent dimension is chosen to be 20. The encoder and the decoder models each has 2 hidden layers with a layer size of input → 1000 → 200 → 20 → 200 → 1000 → output. We used relu and linear function as the activation functions in the encoder model and leaky relu and sigmoid function in the decoder model.

6.2 Application to real data sets

We trained our baseline VAE model on MNIST, Frey Face and CIFAR-10 data sets using the Tensorflow implementation. Availability of GPU computing helped us explore hyper-parameters and network configurations. We were able to implement the baseline VAE model and reproduce Kingma and Welling's results on MNIST dataset. However, the reported model had poor reconstruction and generation quality for CIFAR-10. We discuss the possible reasons behind this in section 6.2.2.

6.2.1 MNIST

In this section, we present and discuss the following results obtained by training the VAE baseline model on the MNIST data set:

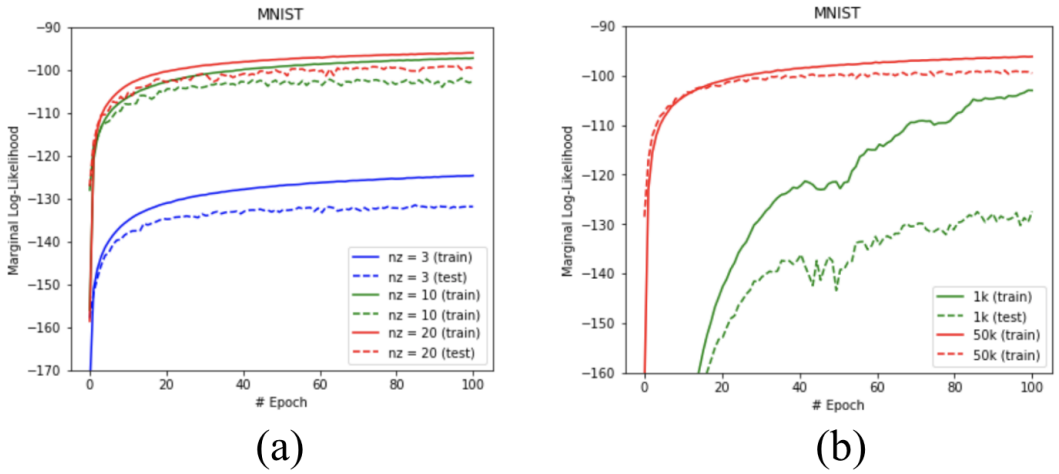
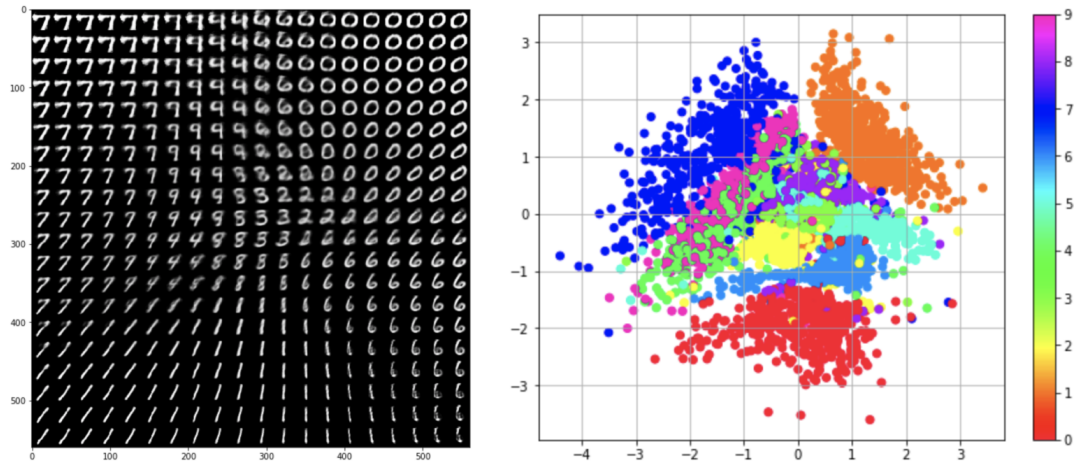


Figure 6.3: Performance of the VAE method on MNIST in terms of optimizing the lower bound, for (a) different dimensionality of latent space nz and (b) different training sample size. Vertical axis: the estimated average variational lower bound per datapoint. Horizontal axis: number of training epochs.

1. **Likelihood lower bound** We trained generative models (decoders) and corresponding encoders having 500 hidden units in case of MNIST. Figure 6.3 shows that the choice of the latent dimensionality affects the model performance. 6.3 (a) suggests that a smaller size of the latent vector restricts the model's ability to learn the data. The model's performance saturates around a latent dimension of 20 for the MNIST data set. In addition, 6.3 (b) indicates that training the VAE model with a small sample size, unsurprisingly, yields less stable performance and lower variational lower bound on the marginal log-likelihood of the data.

2. **Visualization of high-dimensional data** The VAE model is traied to learn the mapping that projects the high-dimensional data to the low-dimensional latent space. We have

seen in Figure 6.2 that the low-dimensional latent representation is able to capture the underlying structure of the data. In Figure 6.4, we again visualize the 2D latent manifolds and the separating pattern for the MNIST datasets.



(a) (b)

Figure 6.4: Learned MNIST manifold and the distribution of MNIST data in 2D latent space.

6.2.2 Frey Face and CIFAR-10

We used 200 hidden units in case of the Frey Face dataset to prevent overfitting, since it is a considerably smaller dataset. Although the generated images look pretty natural for MNIST data set, however, the results for Frey Face and CIFAR-10 are not as good. Figure 6.5 shows that our VAE model only captures the rough features of the images in the frey face dataset, while the reconstructed images for the CIFAR-10 data look blurry. Because the Frey Face data set is small including only 1965 images, the quality of the reconstructed images is better than that for CIFAR-10. The poor reconstruction quality is mainly due to the simple network structure of the baseline VAE model, since we only used fully-connected layers in the encoder and decoder models. We list some possible changes for the network that may lead to better results: (1) Adding convolutional layers to exploit the geometric patterns in the image data; (2) Considering Graph Convolutional Networks (GCNs) to incorporate the neighboring features of the pixels. Because building the current model and experimenting with training it took considerable time, we decided to conclude the project with our current results.

6.3 Comparative analysis

Various algorithms and deep learning methods have been implemented to reconstruct images, and we seek algorithms that are comparable to the VAE method. Although both the

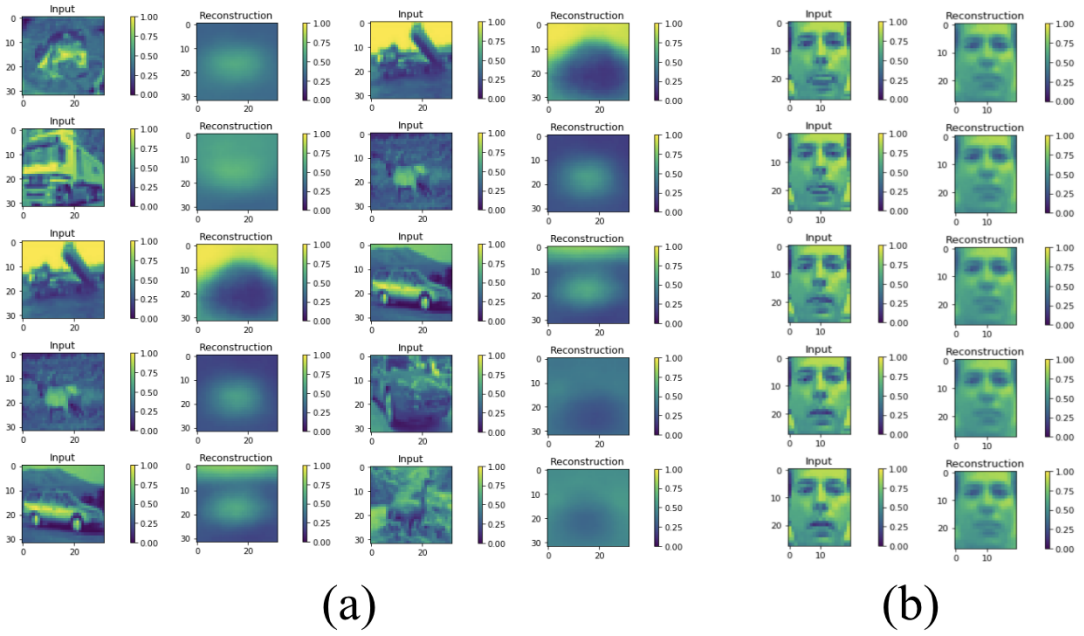


Figure 6.5: Reconstructed images from the datasets: (a) CIFAR-10 (b) Frey Face.

Convolutional Neural Networks (CNN) and Adversarial Generative Networks (GAN) are good candidate generative models, CNN works with explicit and tractable densities, while GAN takes advantage of implicit density estimation. In comparison, Restricted Boltzman Machines (RBM) [5] and Deep Belief Nets (DBN)[6] are generative models that use Markov Chain to approximate densities, and VAE uses variational inference to achieve the same goal. Thus, we compared the VAE method to the following image reconstruction algorithms on the MNIST data set. RBMs are probabilistic (stochastic) graphical models and DBNs are stacked RBMs.

1. Restricted Boltzmann Machines (RBMs)

2. Deep Belief Networks (DBNs)

Our objective is to reconstruct the given inputs (images, densities, etc.), so that the reconstructed outputs from the generative model are faithful approximations to the original inputs. Therefore, we chose the reconstructoin error as our model performance metric. The reconstruction error is defined as the binary cross entropy loss:

$$\text{BCE} = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i) + (1-y_i) \cdot \log(1-\hat{y}_i),$$

where N is the number of data points evaluated, y_i is the input data point, $\hat{y}_i$ is the reconstructed data. A larger BCE indicates a larger deviation of the reconstructed from the input and thus poor reconstruction quality. Before evaluating the performance of the algorithms, we

Figure 6.6: A visual inspection of the reconstructed MNIST digits from the VAE, RBM and DBN methods.

visually inspect the reconstructed MNIST digits from the various algorithms. We trained the three models with the same hidden-layer size (500) and 200 epochs using all 60,000 MNIST training data points. We then reconstructed 100 images plotted in 6.6 and computed their BCE measure. Figure 6.6 shows that the VAE and RBM algorithms perform better than the DBN algorithm in terms of the reconstruction quality. A closer exmination of the BCE scores suggests that RBM achieved the lowest reconstruction error, while DBN obtained the worst reconstruction results.

Table 6.1: Comparison of image reconstruction algorithms on MNIST

Average Reconstruction Error	
VAE	73.783
RBM	19.704
DBN	133.117

7 Discussion

In this project, we tried to generate natural-looking images using VAEs by training them on various datasets. With only fully connected layers, we were able to generate realistic hand-written digits from simple datasets like MNIST and good replicated networks from our simulation study. However, we have seen that the lack of complexity for processing geometric features in our network structure yields less desirable performance for the Frey Face and CIFAR-10 datasets. Furthermore, we compared our VAE model to other generative models that learn to approximate densities of the data-generating distribution.

On the implementation of neural networks, we have seen that debugging and hyper-parameter tuning a network takes lots of effort and time, even with the help of high performance computing. Furthermore, we have seen that good initialization of variables for training and good optimization algorithm is important for convergence of the algorithm, and techniques such as Xavier initialization and Adam optimizer helps our implementation.

8 Installation Instructions

The testing performed and the empirical results are in `https://github.com/yizi0511/sta_663_vae`. Our package has several dependencies. We recommend creating a virtual environment and then installing the required packages. You can install the package either locally or from TestPyPI.

```
1. Clone the Git repo: git clone https://github.com/yizi0511/sta_663_vae.git
2. Create a virtual conda environment: cd sta_663_vae; python3 -m venv vae_env
3. Activate the environment: source vae_env/bin/activate
4. Install the packages into the environment: pip install -r requirements.txt
5. Install our package: python setup.py install
6. Start using it: import vae
```

```
Install package from TestPyPI: pip install -i https://test.pypi.org/simple/ simplevae
```

9 TEAM CONTRIBUTIONS

Both of the team members contributed similar amount of efforts. They shared efforts in both the theoretical understanding and the implementation of the algorithm. TensorFlow was new to the team and initially time was devoted to experimenting with the tool. Chenxi, having more interest on the theory side, put in efforts on learning the different models in the literature. Yizi, having more interest on the implementation, spent time finding good tools to implement the various models. Both members had useful discussions on theory behind VAE and generative models, and applied the VAE model to various datasets. The only marginal difference in efforts, if any, was in time devoted to prepare the report and maintain the Github repository for the project.

REFERENCES

[1] Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. In *ICLR* (2014).

[2] Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. In *J. ACM* 24, 1 (January 1977), DOI:https://doi.org/10.1145/321992.321993

[3] Krzysztof Nowicki Tom A. B Snijders. Estimation and Prediction for Stochastic Blockstructures. In *Journal of the American Statistical Association*, (2001) 96:455, 1077-1087, DOI: 10.1198/016214501753208735

[4] Watts, D. J. Strogatz, S. H.. Collective dynamics of 'small-world' networks. In *Nature*, 393, 440–442. doi: 10.1038/30918

[5] Montufar, Guido. Restricted Boltzmann Machines: Introduction and Review In *arXiv e-prints* (2018).

[6] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *N*eural Comput. 18, 7 (July 2006), 1527–1554. DOI:https://doi.org/10.1162/neco.2006.18.7.1527