Auto-Encoding Variational Bayes

Qiaohui Lin & Boya Xu

Abstract

In case of high-dimensional image data with intractable latent variable, we introduce the Auto-Encoding Variational Bayes (AEVB) algorithm. AEVB is based on Auto-Encoding (a feedforward Multiple-Layer Perceptron algorithm for data compression) but with its optimization objective function replaced by Variational Bayes approach and implemented by stochastic gradient descent (SGVB estimator). SGVB estimator achieves efficient approximation of the intractable posterior and likelihood inside Auto-Encoding and results in a successful dimension reduction AEVB algorithm for large image datasets.

We apply AEVB to the example of binary dataset MNIST and continuous variable dataset FreyFaces and achieves parameter estimation, dimension reduction and image reconstruction. We also compare different gradient methods inside AEVB to optimize the inference. Our code is in package: https://github.com/ql75/AEVB-Lin-Xu-2018.

1 Background

The Variantional Bayesian (VB) approach is to optimize an approximation to the intractable posterior. However, the common mean-field approach requires analytical solutions of expectation with respect to the approximate posterior which is also intractable in many cases. In this report, we introduce the Stochastic Gradient Variantional Bayes (SGVB) estimator which is used for an efficient approximation of the posterior based on a reparameterizatio of the variational lower bound. The optimization process uses standard stochastic gradient ascent techniques.

To deal with image data, we propose the Auto-Encoding VB (AEVB) algorithm. AEVB is essentially a Auto-Encoding (feedforward Multiple-Layer Perceptron for data compression algrithom) with its optimization objective function replaced by Variational Bayes approach and implented by stochastic gradient descent. Using SGVB estimator in original Auto-Encoding algorithm optimizes a recognition model that allows us to efficiently approximate the posterior inference with ancestral sampling, which results in an efficient learning of the model parameters without costly iterative inference schemes (such as MCMC) per datapoint.

We refer to the paper of Kingma and Welling (2013) for the algorithm. We first describe the setting and the goals of the paper.

The dataset $X = \{x^{(i)}\}_{i=1}^{N}$ consists of N i.i.d samples of some continuous or discrete variable x. $z^{(i)}$ is a latent variable which $x^{(i)}$ depends on. The problem is:

- The likelihood of $p_\theta(x|z)$ has a complicated form and thus the posterior of z,$p_\theta(z|x)$ is intractable. The marginal likelihood of x, $\int p_\theta(z)p_\theta(x|z)$ is intractable in this case too.
- The dataset of x can be very large and high dimensional, parameter estimation for θ using usual MCMC and EM could be too slow. A dimension reduction is preferrable.

In this scenerio, we want to achieve:

- Efficient approximation of parameter θ.
- Efficient approximation of likelihood of latent variable z and original high dimensional variable x.

2 Algorithm

2.1 Variational Bayes and SGVB Estimators

We use the variational approximation $q_\phi(z|x)$ to approximate the intractable true posterior $p_\theta(z|x)$. The approximation parameter ϕ is learned jointly with θ in later steps. The likelihood of $X = \{x^{(i)}\}_{i=1}^{N}$ $\log p_\theta(x^{(1)}, ..., x^{(N)}) = \sum_{i=1}^{N} \log p_\theta(x^{(i)})$ can be rewritten as:

$$\log p_\theta(x^{(i)}) = D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z|x^{(i)})) + L(\theta, \phi; x^{(i)}) \quad (1)$$

where the first RHS term is the KL divergence of the approximate from the true posterior and it is non-negative. The second RHS term $L(\theta, \phi; x^{(i)})$ is the variational lower bound on the marginal likelihood of the datapoint i. Thus we have:

$$\log p_\theta(x^{(i)}) \geq L(\theta, \phi; x^{(i)}) = E_{q_\phi(z|x)}[-\log q_\phi(z|x) + \log p_\theta(x, z)] \tag{2}$$

$$= -D_{KL}(q_\phi(z|x^{(i)}))||p_\theta(z|x^{(i)})) + E_{q_\phi(z|x^{(i)})}[p_\theta(x^{(i)}|z)] \tag{3}$$

Instead of optimizing intractable $\log p_\theta(x^{(i)})$, we equivalently optimize the lower bound $L(\theta, \phi; x^{(i)})$ w.r.t both the variational parameters ϕ and generative parameters θ.

With any chosen $q_\phi(z|x)$, we reparameterize the random variable $\widetilde{z} \sim q_\phi(z|x)$, with an auxiliary noise ϵ:

$$\widetilde{z} = g_\phi(\epsilon, x), \epsilon \sim p(\epsilon) \tag{4}$$

Here we can form Monte Carlo estimates of expectations of some function $f(z)$ w.r.t $q_\phi(z|x)$ as follows:

$$E_{q_\phi(z|x^{(i)})}[f(z)] = E_{p(\epsilon)}[f(g_\phi(\epsilon, x^{(i)}))] \simeq \frac{1}{L}\sum_{l=1}^{L} f(g_\phi(\epsilon^{(l)}, x^{(i)})) \tag{5}$$

$$\epsilon^{(l)} \sim p(\epsilon) \tag{6}$$

We apply this technique to the variational lower bound and yield the generic SGVB estimator $\widetilde{L}^A(\theta, \phi; x^{(i)}) \simeq L(\theta, \phi; x^{(i)})$:

$$\widetilde{L}^A(\theta, \phi; x^{(i)}) = \frac{1}{L}\sum_{l=1}^{L} \log p_\theta(x^{(i)}, z^{(i,l)}) - \log q_\phi(z^{(i,l)}|x^{(i)}) \tag{7}$$

$$z^{(i,l)} = g_\phi(\epsilon^{(i,l)}, x^{(i)}) \tag{8}$$

When $D_{KL}(q_\phi(z|x^{(i)})||p_\theta(z|x^{(i)}))$ can be analytically integrated, we only need to estimate the expected reconstruction error $E_{q_\phi(z|x^{(i)})}[p_\theta(x^{(i)}|z)]$ by sampling. Thus we have the second version of the SGVB estimator $\widetilde{L}^B(\theta, \phi; x^{(i)}) \simeq L(\theta, \phi; x^{(i)})$:

$$\widetilde{L}^B(\theta,\phi;x^{(i)}) = -D_{KL}(q_\phi(z|x^{(i)}))||p_\theta(z|x^{(i)})) + \frac{1}{L}\sum_{l=1}^{L}\log p_\theta(x^{(i)}|z^{(i,l)}) \quad (9)$$

$$z^{(i,l)} = g_\phi(\epsilon^{(i,l)}, x^{(i)}) \quad (10)$$

The KL-divergence term can be interpreted as regularizing ϕ and $\widetilde{L}^B(\theta,\phi;x^{(i)})$ typically has less variance than $\widetilde{L}^A(\theta,\phi;x^{(i)})$. In this case, since KL-Divergence is analytical, we use the $\widetilde{L}^B$ as our SGVB estimator.

2.2 Auto-Encoding

Auto-encoding is widely used for dimension reduction and data compression for images. We first recall the traditional autoencoder architecture:

$$\phi : X \to F \quad (11)$$

$$\psi : F \to X \quad (12)$$

$$\phi, \psi = \arg\min ||X - (\psi \circ \phi)X||^2 \quad (13)$$

In the simplest case with only one hidden layer, the first stage (encoder stage) of an autoencoding takes the input $x \in R^d$ that is then mapped to the latent code $z \in R^k$.

$$\mathbf{z} = f_\phi(\mathbf{W}\mathbf{x} + \mathbf{b}) \quad (14)$$

where f_ϕ is an element-wise activation function such as a sigmoid function or a rectified linear unit. $\mathbf{W}$ is the weight matrix and $\mathbf{b}$ is the bias vector. ϕ is the parameter we want to estimate depending on W and b

Second, the decoder stage of the auto-encoding maps $\mathbf{z} \in R^k$ to the reconstruction $\mathbf{x}' \in R^d$ as of the same shape as $\mathbf{x}$:

$$\mathbf{x}' = f'_\theta(\mathbf{W}'\mathbf{z} + \mathbf{b}') \quad (15)$$

where $\mathbf{f}'_\theta, \mathbf{W}'$ and $\mathbf{b}'$ for the decoder may differ in general from the corresponding $\mathbf{f}_\phi, \mathbf{W}$ and $\mathbf{b}$ for the encoder. θ is the parameter we want to estimate depending on W' and b'.

These two stages in autoencoding are traditionally trained to minimise reconstruction errors:

$$\mathbf{L}(\mathbf{x},\mathbf{x'}) = ||\mathbf{x}-\mathbf{x'}||^2 = ||\mathbf{x} - f'_\theta(\mathbf{W'}(f_\phi(\mathbf{Wx}+\mathbf{b}))+\mathbf{b'})||^2 \tag{16}$$
$$W, b, W', b' = argmin \quad L \tag{17}$$

Now we change traditional auto-encoding by our VB approach and SGVB estimator to generate AEVB. In AEVB, we do not use the traditional reconstruction loss function (Equation 16). We see encoder (Equation (14)) as variational approximation of z on x, and see decoder (Equation (15)) as conditional likelihood of x on z. Instead of minimizing $\mathbf{L}(\mathbf{x},\mathbf{x'})$ in Equation(16), we use SGVB estimator $\widetilde{L}^B$ to maximize the variational lower bound. Equation (16) and (17) thus becomes:

$$\widetilde{L}^B(\theta,\phi;x^{(i)}) = -D_{KL}(f_\phi(z|x^{(i)}))||p_\theta(z|x^{(i)})) + \frac{1}{L}\sum_{l=1}^{L}\log f'_\theta(x^{(i)}|z^{(i,l)}) \tag{18}$$
$$W, b, W', b' = argmaxL \quad updated-by-SGD \tag{19}$$

2.3 AEVB (Auto-Encoding Variational Bayes) Algorithm

Specifically in our AEVB, we use a neural network for encoding and decoding in the above section. Let the prior over the latent variables be the centered isotropic multivariate Gaussian $p_\theta(\mathbf{z}) = \mathcal{N}(\mathbf{z};\mathbf{0},\mathbf{I})$. We assume the true (but intractable) posterior takes on a approximate Gaussian form with an approximately diagonal covariance. In this case, we can let the variational approximate posterior be a multivariate Gaussian with a diagonal covariance structure:

$$\log q_\phi(\mathbf{z}|\mathbf{x}^{(i)}) = \log\mathcal{N}(\mathbf{z};\mu^{(i)},\sigma^{2(i)}\mathbf{I}) \tag{20}$$

where the mean and s.d. of the approximate posterior, $\mu^{(i)}$ and $\sigma^{2(i)}$, are outputs of the encoding MLP (a fully-connected neural network with a single hidden layer), i.e. nonlinear functions of datapoint $x^{(i)}$ and the variational parameters ϕ:

$$\log q_\phi = \log\mathcal{N}(\mathbf{z};\mu,\sigma^2\mathbf{I}) \tag{21}$$
$$\mu = \mathbf{W_1h}+\mathbf{b_1} \tag{22}$$
$$\log\sigma^2 = \mathbf{W_2h}+\mathbf{b_2} \tag{23}$$
$$\mathbf{h} = \tanh(\mathbf{W_3z}+\mathbf{b_3}) \tag{24}$$

Then, we sample from the posterior $\mathbf{z}^{(i,l)} \sim q_\phi(\mathbf{z}|\mathbf{x}^{(i)})$ using $\mathbf{z}^{(i,l)} = g_\phi(\mathbf{x}^{(i)}, \epsilon^{(l)}) = \mu^{(i)} + \sigma^{(i)} \bigodot \epsilon^{(l)}$ where $\epsilon^{(l)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. With $\bigodot$ we signify an element-wise product. Since both the prior $p_\theta(\mathbf{z})$ and $q_\phi(\mathbf{z}|\mathbf{x})$ are Gaussian, KL divergence are analytically solved and we use $\widetilde{L}^B$ with a solved KL divergence:

$$L(\theta, \phi; \mathbf{x}^{(i)}) \simeq \frac{1}{2} \sum_{j=1}^{J} (1 + \log((\sigma_j^{(i)})^2) - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2) + \frac{1}{L} \sum_{l=1}^{L} \log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)}) \tag{25}$$

Case 1: Binary Dataset (Binary Decoder)
For binary dataset such as MNIST with N datapoints, we use a Bernoulli decoder for $\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)})$:

$$\log p_\theta(x|z) = \sum_{i=1}^{N} x_i \log y_i + (1 - x_i) \cdot \log(1 - y_i) \tag{26}$$

$$y = f_{sigmoid}(\mathbf{W_5} tanh(\mathbf{W_4} z + \mathbf{b_4}) + \mathbf{b_5}) \tag{27}$$

where $f_{sigmoid}$ is the sigmoid activation function.

Case 2: Continuous Dataset (Gaussian Decoder)
For datasets with continuous variables such as FreyFaces, we use the Gaussian decoder for $\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)})$:

$$\log p_\theta(x|z) = \log \mathcal{N}(\mathbf{x}; \mu', \sigma'^2 \mathbf{I}) \tag{28}$$

$$\mu' = \mathbf{W_4} \mathbf{h}' + \mathbf{b_4} \tag{29}$$

$$\log \sigma'^2 = \mathbf{W_5} \mathbf{h}' + \mathbf{b_5} \tag{30}$$

$$\mathbf{h}' = \tanh(\mathbf{W_6} \mathbf{x} + \mathbf{b_6}) \tag{31}$$

$\{\mathbf{W}_i, \mathbf{b}_i\}$ are the weights and biases of the MLPs for encoder and decoder and the parameters that are actually updated and optimized in coding process. $\phi = \{\mu, \sigma\}$ and $\theta = y$ (Bernoulli) or $\theta = \{\mu', \sigma'\}$ (Gaussian) are parameters we want to estimate as a result.

3 Implementation

Considering the size of the datasets, we run the model on each minibatch of size 100 and run 10000 epochs on Bernoulli case (MNIST) and 100000 epochs on Gaussian case (FreyFaces) with first 1000 burn-in.

Initializing Specific settings for the MNIST and Frey Face datasets:

- The size of minibatch $M = 100$ and the size of samples per datapoint $L = 1$.
- The initialized values of parameters ϕ_0 and θ_0 are randomly sampled from $\mathcal{N}(0, 0.01)$.
- Stepsizes used for updating parameters within the algorithm are adapted by Adagrad with parameters chosen from $\{0.01, 0.02, 0.1\}$.
- When obtaining the likelihood lower bound, we trained the encoder and decoder with 100 hidden units for both MNIST, and Frey Faces dataset.

The steps of implementation can be summarized as:

```
- Initialize parameters $\theta_{\mathbf{0}}, \phi_{\mathbf{0}}$
- repeat:
    1. Draw a random minibatch of M datapoints from datasets of size N, noted as $X^M$
    2. - Encoding: $X \to Z; q_\phi(z|x)$
       - sample z from $q_\phi(z|x)$ with noise $\epsilon \sim \mathcal{N}(0, \mathcal{I})$
       - Decoding: $Z \to X; p_\theta(x|z)$
       - Note: $\theta$ and $\phi$ are jointly updated with objective function $L(\theta, \phi; \mathbf{x}^{(i)})$ in Equation(25) with Gradient Descent.
- until: convergence of parameters $(\theta, \phi)$ or max-epochs is reached.
- return: $\theta, \phi$
```

We run the process above to get the estimation for θ, ϕ from the training set, and we test the estimations on the test set and reconstruct the images from both MNIST and FreyFaces.

The complete code is done in python and in package :
https://github.com/ql75/AEVB-Lin-Xu-2018.

4 Result

4.1 Convergence Analysis

We operated the model on binary dataset(MNIST, dimension of $70000 * 28 * 28$)and continuous dataset (FreyFaces, $1965 * 28 * 20$). For MNIST, we used the first 60000 data points as training set, last 10000 as test set; and ran the Berboulli case AEVB with 10^5 epochs. For FreyFaces, we used first 1500 data points as training, last 465 as testing; and ran the Gaussian case AEVB with 10^6 epochs with first 1000 burn-in. We repeated the model with latent variable z of dimension $N_z = 2, 3, 5, 10$ and plot the variational lower bound as below:

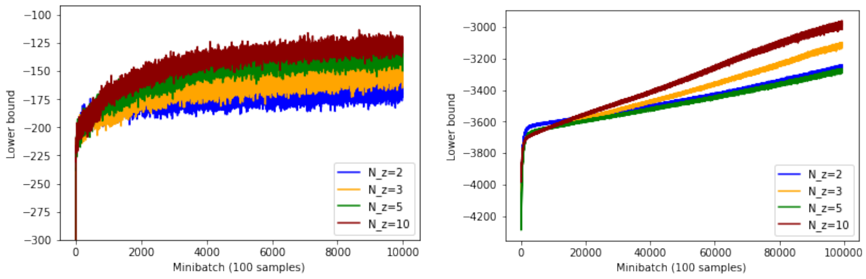


Figure 1: Variational Lower Bound implementing AEVB for Bernoulli case MNIST (left) and Gaussian case FreyFaces (right), with latent variable z of dimension 2, 3, 5 and 10.

Both models converge after sufficient epochs. The numerical result of lower bound is very similar to the paper we are replicating. Bernoulli case lower bound converges to around -150 and Gaussian case converges to around -3000.

Regarding the dimension of latent variable z, we find that higher dimension of latent variable results in a higher lower bound, and thus a closer approximation to true posteriors in both cases. Kingma and Welling (2013) pointed out in the original paper that superfluous latent variable did not result in overfitting because of the regularization nature of lower bound.

The marginal likelihoods are shown below, with similar shape and convergence to lower bounds and a higher accuracy with respect to higher dimension of z.

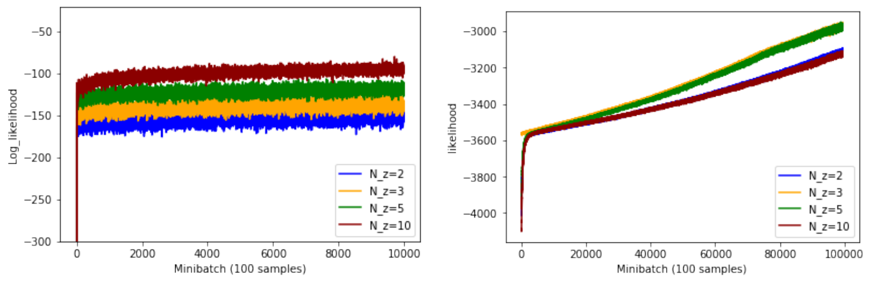


Figure 2: Marginal loglikelihood of original data x implementing AEVB for Bernoulli case MNIST (left) and Gaussian case FreyFaces (right), with latent variable z of dimension 2, 3, 5 and 10.

4.2 Reconstruction and Visualization

We use our trained model of learned parameter *theta* and ϕ on our test set to reconstruct the test set data after compressing into a lower dimension space. We show an example of visualizing the test set reconstruction of MNIST and FreyFaces in 2D latent variable case.

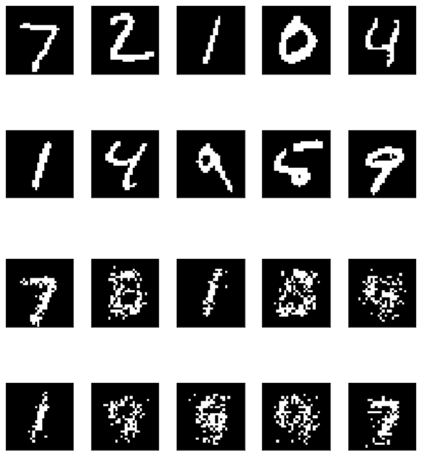


Figure 3: MNIST using Bernoulli decoder AEVB with 2D latent variable z, first two lines are original test set example and the last two lines are the reconstruction result

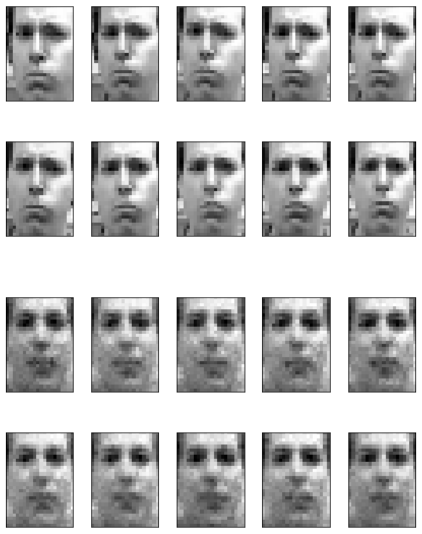


Figure 4: FreyFaces using Gaussian decoder AEVB with 2D latent variable z, first two lines are original test set example and the last two lines are the reconstruction result

5 Comparative Analysis

5.1 Comparison among different methods of gradient descent

We incorporate five different methods of gradient descent for updating the estimates $\hat{\phi}$ and $\hat{\theta}$ in the AEVB algorithm. They are SGD, Momentum SGD, AdaGrad, Adam and RMSprop. As follows, we show the comparison among them in terms of the value of lower bound with $N_z = 2$, and experiments are on the MNIST dataset.

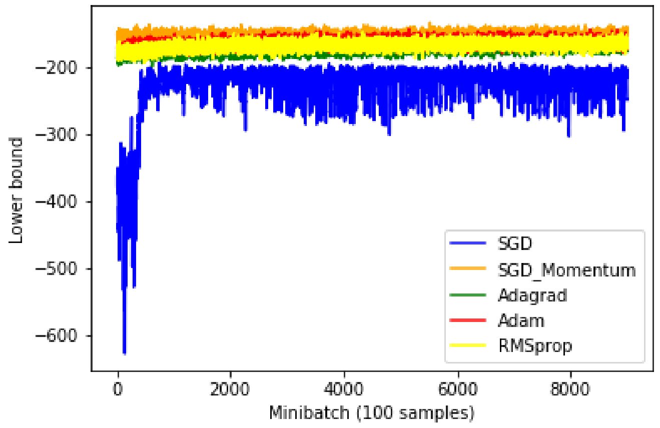


Figure 5: The lower bound of MNIST dataset using Bernoulli decoder with 2D latent variables z, in terms of different gradient descent approahces.

From this figure, we find that the Momentum SGD performs the best with the largest lower bound and quick convergence. By contrast, the SGD has the smallest value of lower bound and the rate of convergence. Therefore, in terms of the performance on optimization and convergence, the rank of superiority among these five methods is: Momentum SGD, Adam, RMSprop, Adagrad, SGD. Furthermore, we compare the operation time among them and achieve the following table:

Methods	SGD	Momentum SGD	AdaGrad	Adam	RMSprop
Time	5'29"	5'56"	6'42"	6'21"	6'50"

From the result above, we can find that the Momentum SGD also has an advantage over other approaches excpet SGD, in terms of the operation speed.

5.2 Comparison between AEVB and Wake-Sleep algoritm

The previous work shown by Kingma & Welling indicates that the AEVB algorithm can converge considerably faster and reach a better solution in all experiments of optimizing the lower bound, compared to the wake-sleep algorithm. In terms of the estimated marginal likelihood, the AEVB can also dominate other approaches such as the wake-sleep algorithm and Monte

Carlo EM. Especially, the Monte Carlo EM algorithm cannot work on the online dataset.

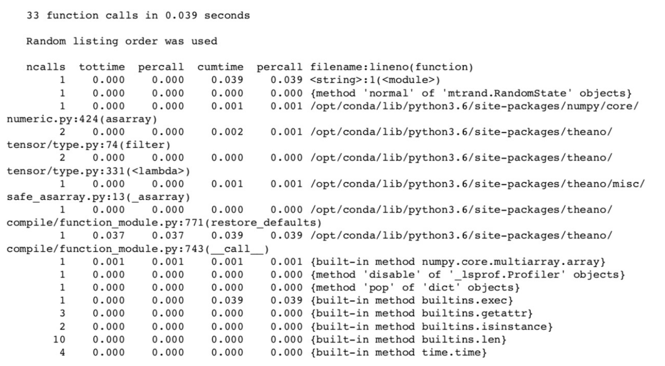
6 Testing with Simulated Dataset

To test if our codes can be applied to any dataset with correctly, we work on two random samples including binary and continuous data. The main task is to check if the values of lower bound and loglikelihood are negative, and if they converge as iteration increases. The random test set for Bernoulli decoder is randomly sampled from 0,1 with the size [1000, 50], and the set for Gaussian decoder is from the standard normal distribution with the same size. We applied all methods of gradient descent in the test. The results have confirmed that all values of lower bound and loglikelihood are negative, and our algorithm can lead to a quick convergence with 10000 iterations.

7 Profiling and Optimization

As we work on a machine learning case with high-dimensional image datasets, the original code before optimization is compiled through Theano, a python library, optimize and evaluate mathematical expressions, especially efficient for high-dimensional matrices. The fact is that using the plain Python with numpy or scipy is not feasible for neural networks training in our case, which needs us to mannually solve complex computations and updates of function gradients. Therefore, the use of Theano facilities and accelerates our work on "Big Data" via GPU, given its symbolic expressions and their advantages on calculating gradients, compared to the plain Python. Considering that we also define different methods of gradient descent as functions, we apply the JIT from numba to optimize them. However, we may figure out little improvement from this perspective. This result can be explained by our use of theano within the definition of those functions, where all variables incorporated in calculations have been symbolized and particular computations are wrapped to operate in the last training model (also in a form of theano function).

The good performance of theano codes can also be reflected by the profiling report, as follows:

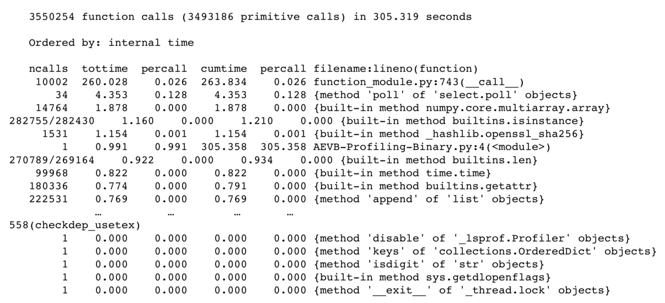


```
   33 function calls in 0.039 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.039    0.039 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'normal' of 'mtrand.RandomState' objects}
        1    0.000    0.000    0.001    0.001 /opt/conda/lib/python3.6/site-packages/numpy/core/
numeric.py:424(asarray)
        2    0.000    0.000    0.002    0.001 /opt/conda/lib/python3.6/site-packages/theano/
tensor/type.py:74(filter)
        2    0.000    0.000    0.000    0.000 /opt/conda/lib/python3.6/site-packages/theano/
tensor/type.py:331(<lambda>)
        1    0.000    0.000    0.001    0.001 /opt/conda/lib/python3.6/site-packages/theano/misc/
safe_asarray.py:13(_asarray)
        1    0.000    0.000    0.000    0.000 /opt/conda/lib/python3.6/site-packages/theano/
compile/function_module.py:771(restore_defaults)
        1    0.037    0.037    0.039    0.039 /opt/conda/lib/python3.6/site-packages/theano/
compile/function_module.py:743(__call__)
        1    0.001    0.001    0.001    0.001 {built-in method numpy.core.multiarray.array}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {method 'pop' of 'dict' objects}
        1    0.000    0.000    0.039    0.039 {built-in method builtins.exec}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.getattr}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
       10    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        4    0.000    0.000    0.000    0.000 {built-in method time.time}
```

Figure 6: Profiling Report of adagrad Nz=2 MNIST Dataset with only one minibatch, sorted by operation time and listed the most and least time-consuming functions



```
   3550254 function calls (3493186 primitive calls) in 305.319 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    10002  260.028    0.026  263.834    0.026 function_module.py:743(__call__)
       34    4.353    0.128    4.353    0.128 {method 'poll' of 'select.poll' objects}
    14764    1.878    0.000    1.878    0.000 {built-in method numpy.core.multiarray.array}
282755/282430    1.160    0.000    1.210    0.000 {built-in method builtins.isinstance}
     1531    1.154    0.001    1.154    0.001 {built-in method _hashlib.openssl_sha256}
        1    0.991    0.991  305.358  305.358 AEVB-Profiling-Binary.py:4(<module>)
270789/269164    0.922    0.000    0.934    0.000 {built-in method builtins.len}
    99968    0.822    0.000    0.822    0.000 {built-in method time.time}
   180336    0.774    0.000    0.791    0.000 {built-in method builtins.getattr}
   222531    0.769    0.000    0.769    0.000 {method 'append' of 'list' objects}
           …         …          …         …
558(checkdep_usetex)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {method 'keys' of 'collections.OrderedDict' objects}
        1    0.000    0.000    0.000    0.000 {method 'isdigit' of 'str' objects}
        1    0.000    0.000    0.000    0.000 {built-in method sys.getdlopenflags}
        1    0.000    0.000    0.000    0.000 {method '__exit__' of '_thread.lock' objects}
```

Figure 7: Profiling Report of adagrad Nz=2 MNIST Dataset with full dataset with all 10000 epochs, sorted by operation time and listed the most and least time-consuming functions

8 Discussion

We introduced the Stochastic Gradient VB estimator of the variational lower bound, with continuous latent variables which also have a lower dimension. We also implemented the Anto-Encoding VB algorithm to achieve parameters of recognition and generative models to optimize the lower bound, with the MLP encoder and decoder. This newly proposed algorithm has a better performance on convergence rate and the final solutions, considering the value of lower bound and marginal likelihood in all experiments, compared to wake-sleep algorithm and Monte Carlo EM. Additionally, we provided several different and popular methods of gradient descent in the package. To show all application results, we worked on the MNIST and FreyFace datesets, where the former one is binary data and the later one is continuous.

References

[1] DiederikP.Kingma and Max Welling (2014): Auto-Encoding Variational Bayes [J]. *arXiv*: 1312.6114v10 [stat.ML].

[2] John Duchi, Elad Hazan, and Yoram Singer (2010): Adaptive subgradient methods for online learning and stochastic optimization [J]. *Journal of Machine Learning Research*, 12: 2121-2159.

[3] Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth (1987): Hybrid monte carlo [J]. *Physics Letters B*, 195(2): 216-222.

[4] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal (1995): The "wake-sleep" algorithm for unsupervised neural networks[J].*Science*, pp 1158-1158.

[5] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra (2014): Stochastic backpropagation and variational inference in deep latent gaussian models [J]. *arXiv preprint arXiv*: 1401.4082.

[6] Alec Radford (2015): Theano Tutorials. http://github.com/Newmu/Theano-Tutorials.