Stein Variational Gradient Descent and projected Stein Variational Gradient Descent

Changhao Ge, Zheming Cao

January 3, 2021

Contents

Chapter 1

Overview

1.1 Introduction

This report is based on two papers of Bayesian inference. The first one is **Stein Variational Gradient Descent: A General Purpose Bayesian Inference Algorithm**, and the second one is **Projected Stein Variational Gradient Descent**.

Stein Variational Gradient Descent(SVGD) is a general purpose variational inference algorithm that forms a natural counterpart of gradient descent for opptimization. The method iteratively transports a set of particles to match the target distribution, by applying a form of functional gradient descent that minimizes the KL divergence. Comparing with other algorithms such as EM algorithm, SVGD shows better performance at ugly initial values.

However, SVGD meets the curse of dimensionality in high dimensions. Projected Stein variational gradient descent(pSVGD) overcomes this challenge by exploiting the fundamental property of intrinsic low dimensionality of the data informed subspace stemming from ill-posedness of such problems.

We briefly introduced the knowledge and theory needed for these two methods, finished the algorithms in these two articles with R, and improved the computational performance with Rcpp. Besides, we implemented practical experiment with multivariate normal and Gaussian mixture models, and compared them with EM algorithm. The result shows SVGD and pSVGD algorithms have better performance. Finally, we made an R package and uploaded it on Github.

In the last chapter, we introduced some applications of SVGD and pSVGD.

1.2 Contribution

Changhao Ge finished Algorithm 1 and 2 and their practical experiments, rewrote Algorithm 3 by Rcpp and compiled this report.

Zheming Cao finished Algorithm 3 and its practical experiment, wrote the R package, and little part of the report.

Chapter 2

Theories and Algorithms

2.1 Background Knowledge

Preliminaries

Let $x \in \mathbb{R}^d$ is a random parameter of dimension d with a continuous prior distribution $p_0 : \mathbb{R}^d \to \mathbb{R}$.

Let $y = \{y_i\}_{i=1}^s$ denote the i.i.d observation data.

Let $f(x) := \prod_{i=1}^s p(y_i|x)$ denote a continuous likelyhood of y at given x(up to a irrelavant constant).

Then the posterior density of parameter x is given by Bayes' rule:

$$p(x) = \frac{1}{Z} f(x) p_0(x) \tag{2.1}$$

where Z is the normalization constant:

$$Z = \int_{\mathbb{R}^d} f(x) p_0(x) dx \tag{2.2}$$

Stein's Identity

Let $\mathcal{A}_p \phi(x) = \phi(x) \nabla_x \log p(x)^\top + \nabla_x \phi(x)$,then we have

$$\mathbf{E}_{x\sim p}[\mathcal{A}_p \phi(x)] = 0 \tag{2.3}$$

when ϕ is good enough(we call in Stein class), and

$$\mathbf{E}_{x\sim q}[\mathcal{A}_p \phi(x)] \neq 0 \tag{2.4}$$

for general ϕ when $p \neq q$. So this could determine a metric between two distributions p and q.

KL divergence

Similar to the metric above, the KL divergence:

$$\mathbf{D}_{KL}(q|p) := \mathbf{E}_{x\sim q}[\log(q/p)] \tag{2.5}$$

can also define a metric between two distributions p an q. Noticeably, the metric is not symmetric w.r.t p and q.

Functional Gradient

For any functional $F[\boldsymbol{f}]$ of $\boldsymbol{f} \in \mathcal{H}^d$, its (functional) gradient $\nabla_{\boldsymbol{f}} F[\boldsymbol{f}]$is a function in $\mathcal{H}^d$ such that $F[\boldsymbol{f} + \epsilon \boldsymbol{g}(x)] = F[\boldsymbol{f}] + \epsilon \langle \nabla_{\boldsymbol{f}} F[\boldsymbol{f}], \boldsymbol{g} \rangle_{\mathcal{H}^d} + O(\epsilon^2)$ for any $\boldsymbol{g}$ in $\mathcal{H}^d$ and $\epsilon \in \mathbb{R}$.

2.2 SVGD

The key point of Stein Variational Gradient Descent(SVGD) is to estimate posterior distribution p by optimal $q* \in \mathcal{Q}$, such that

$$q* = \arg\min_{q\in\mathcal{Q}} \mathbf{D}_{KL}(q|p) \tag{2.6}$$

where $\mathcal{Q}$ is a family of distributions having good properties.

We take $\mathcal{Q}$ to be the set of distributions of form $z = \mathbf{T}(x)$ where $\mathbf{T} : X \to X$ is a smooth one-to-one transfor, and x has density $q_0(x)$. By the chain rule, the density of z is

$$q_{[\mathbf{T}]}(z) = q(\mathbf{T}^{-1}(z)) \cdot |\det(\nabla_z \mathbf{T}^{-1}(z))|$$

where $\mathbf{T}^{-1}$ denotes the inverse map of $\mathbf{T}$ and $\nabla_z \mathbf{T}^{-1}$ the Jacobian matrix of $\mathbf{T}^{-1}$.

In order to implement gradient descent, we consider a small vibration of the identity map: $\mathbf{T}(x) = x + \epsilon\phi(x)$, where $\phi(x)$ is a smooth function characterizing the disturbance direction and the scalar ϵ represents the magnitude. In this work we take $|\epsilon|$ so small that the Jacobian of $\mathbf{T}$ is full rank to guarantee an one-to-one map by the inverse function theorem.

The following result draws a connection between Stein operator and the derivative of KL divergence w.r.t the disturbance magnitude ϵ.

Theorem 1. *Let $\mathbf{T}(x) = x + \epsilon\phi(x) and q_{[\mathbf{T}]}(z)$ the density of $z = \mathbf{T}(x)$ when $x \sim q(x)$, we have*

$$\nabla_\epsilon D_{KL}(q_{[\mathbf{T}]}(z)\|p)|_{\epsilon=0} = -\mathbf{E}_{x\sim q}[trace(\mathcal{A}_p\phi(x))] \tag{2.7}$$

where $\mathcal{A}_p\phi(x) = \nabla_x \log p(x)\phi(x)^\top + \nabla_x\phi(x)$ is the stein operator.

Define RBF kernel as below:

$$k(x, y) = \exp(-\frac{1}{h}\|x - y\|_2^2) \tag{2.8}$$

where h is chosen by a criterion. Note that RBF kernel is in Stein's class defined in the former section.

Define KSD as below:

$$\mathbf{S}(q, p) = \max_{\phi\in\mathcal{H}^d}\{[\mathbf{E}_{x\sim q}(trace(\mathcal{A}_p\phi(x)))]^2, \quad s.t \quad \|\phi\|_{\mathcal{H}^d} \le 1\} \tag{2.9}$$

It's shown that the optimal solution to 2.9 is $\phi(x) = \phi^*_{q,p}(x)/\|\phi^*_{q,p}(x)\|_{\mathcal{H}^d}$,where

$$\phi^*_{q,p}(\cdot) = \mathbf{E}_{x\sim q}[\mathcal{A}_p k(s, \cdot)], \quad \text{for which we have} \quad \mathbf{S}(q, p) = \|\phi^*_{q,p}\|^2_{\mathcal{H}^d} \tag{2.10}$$

Lemma 1. *Assume the conditions in Theorem 1. Consider all the perturbation directions ϕ in the ball $\mathcal{B} = \{\phi \in \mathcal{H}^d : \|\phi\|^2_{\mathcal{H}^d} \le \mathbf{S}(q, p)\}$ of vector-valued RKHS $\mathcal{H}^d$, the direction of steepest descent that maximizes the negative gradient in 2.12 is the $\phi^*_{q,p}$ in 2.10, i.e.,*

$$\phi^*_{q,p}(\cdot) = \mathbf{E}_{x\sim q}[k(x, \cdot)\nabla_x \log p(x) + \nabla_x k(x, \cdot)] \tag{2.11}$$

for which the negative gradient in 2.12 equals KSD, that is, $\nabla_\epsilon KL(q[\mathbf{T}]\|p)|_{\epsilon=0} = -\mathbf{S}(q, p)$

Reinterpreting 2.7 as a functional gradient in RKHS, we have the following theoreom:

Theorem 2. *let $\boldsymbol{T}(x) = x + \boldsymbol{f}(x)$, where $\boldsymbol{f} \in \mathcal{H}^d$, and $q_{[\boldsymbol{T}]}$ the density of $z = \boldsymbol{T}(x)$ when $x \sim q$,*

$$\nabla_{\boldsymbol{f}} \mathbf{D}_{KL}(q_{[\boldsymbol{T}]} \| p)|_{\boldsymbol{f}=0} = -\phi^*_{q,p}(x) \tag{2.12}$$

whose squared RKHS norm is $\|\phi^*_{q,p}\|^2_{\mathcal{H}^d} = \mathbf{S}(q,p)$ given by 2.9.

Then we can make Bayesian Inference via this gradient descent algorithm. Note that $\hat{\phi}^*(x)$ is the approximation of function $\phi^*_{q,p}(x)$ for the later one is hard to compute with an integral.

Algorithm 1 Stein Variational Gradient Descent

```
Input: A target distribution with density function $p(x)$ and a set of initial particles $\{x_i^0\}_n^{i=1}$.
Output: A set of particles $\{x_i\}_{i=1}^n$ that approximates the target distribution.
1: for iteration $\ell$ do
       $x_i^{\ell+1} \leftarrow x_i^\ell + \epsilon_\ell \hat{\phi}^*(x_i^\ell)$  where  $\hat{\phi}^*(x) = \frac{1}{n}\sum_{j=1}^n [k(x_j^\ell, x)\nabla_{x_j^\ell} \log p(x_j^\ell) + \nabla_{x_j^\ell} k(x_j^\ell, x)]$   (2.13)
   where $\epsilon_\ell$ is the step size at the $\ell$-th iteration.
```

2.3 pSVGD

Because of the existence of kernel $k(x,y)$ which faces the curse of dimensionality, pSVGD tackles the same question with SVGD, but with a procedure to reduce the dimensionality of parameters.

Define $H \in \mathbb{R}^{d\times d}$, which is called the gradient information matrix representing the average outer product of the gradient of the log-likelihood w.r.t the posterior, as below:

$$H = \int_{\mathbb{R}^d} (\nabla_x \log f(x))(\nabla_x \log f(x))^\top p(x)dx \tag{2.14}$$

By $(\lambda_i, \psi_i)_{i=1}^r$ we denote the r eigen pairs of (H, Γ) having largest eigenvalues, with Γ representing the covariance of the parameter x w.r.t its prior.

$$H\psi_i = \lambda_i \Gamma \psi_i \tag{2.15}$$

Then define a projector (also can be seen as a project matrix) of rank r, $P_r : \mathbb{R}^d \to \mathbb{R}^d$ as

$$P_r x ;= \sum_{i=1}^r \psi_i \psi_i^\top x = \Psi_r w, \quad \forall x \in \mathbb{R}^d \tag{2.16}$$

where $\Psi_r := (\psi_1, \psi_2, \ldots, \psi_r) \in \mathbb{R}^{d\times r}$ represents the projection matrix w.r.t w and $w := (w_1, w_2, \ldots, w_r)^\top \in \mathbb{R}^r$ is a coefficient vector with element $w_i := \psi_i^\top x$ for $i = 1, 2, \ldots, r$.

Given w, we want to find a function $g : \mathbb{R}^d \to \mathbb{R}$, which receive projected vector and is a good approximation of likelihood function $f(x)$,i.e. $g(P_r x) \approx f(x)$. Suppose we have found the optimal g, and define the projected density $p_r : \mathbb{R}^d \to \mathbb{R}$:

$$p_r(x) := \frac{1}{Z_r} g(P_r x) p_0(x) \tag{2.17}$$

where $Z_r := \mathbf{E}_{x\sim p_0}[g(P_r x)]$. It's proven that an optimal $g = g^*$ exists such that

$$\mathbf{D}_{KL}(p|p_r^*) \leq \mathbf{D}_{KL}(p|p_r) \tag{2.18}$$

when p_r^* is defined as in 2.17 with g^*.

Moreover, under some mild conditions one can show that

$$\mathbf{D}_{KL}(p|p_r^*) \leq \frac{\gamma}{2}\sum_{i=r+1}^{d}\lambda_i \tag{2.19}$$

for a constanc γ independent of r.

Now the fact is that the optimal profile function g^* is the marginal likelihood, i.e.

$$g^*(P_r x) = \int_{X_\perp} f(P_r x + \xi)p_0^\perp(\xi|P_r x)d\xi \tag{2.20}$$

where $X_\perp$ is the complement space of $X_r := span(\psi_1, \psi_2, \ldots, \psi_r)$ and

$$p_0^\perp(\xi|P_r x) = p_0(P_r x + \xi)/p_0^r(P_r x) \quad \text{with} \quad p_0^r(P_r x) = \int_{X_\perp} p_0(P_r x + \xi)d\xi \tag{2.21}$$

Then the prior distribution can be rewritten as

$$p_0(x) = p_0^r(x^r)p_0^\perp(x^\perp|x^r) \tag{2.22}$$

and we define

$$\pi_0(w) = p_0^r(\Psi_r w) \qquad \pi(w) = \frac{1}{Z_w}g^*(\Psi_r w)\pi_0(w) \tag{2.23}$$

where Z_w is a normalization constant.

Then the projected distribution in 2.17 can be expressed as

$$p_r(x) = \pi(w)p_0^\perp(x^\top|\Psi_r w) \tag{2.24}$$

Then we only need to optimize $\pi(w)$ and $p_0^\perp(x^\top|\Psi_r w)$ respectively. In fact, $\pi(w)$ can be processed by SVGD method, so we only need to think about the later one. Moreover, we have the properties below:

$$\nabla_w \log \pi(w) = \Psi_r^\top\left(\frac{\nabla_x g(P_r x)}{g(P_r x)} + \frac{\nabla_x p_0^r(P_r x)}{p_0^r(P_r x)}\right) \tag{2.25}$$

which links w in pSVGD and x in SVGD.

Now we deal with some difficulties in this algorithm.

g^* in 2.20 involve high-dimensional integrals. One possible solution is to approximate it by

$$g^*(P_r x_n^l) \approx f(P_r x_n^l + x_n^\perp)$$

where $x_n^\perp = x_n^0 - P_r x_n^0$ is seen as a sample from distribution $p_0^\perp(x^\perp|P_r x)$.

The second problem is the calculation of H in 2.15. We also approximate it by

$$\hat{H} := \frac{1}{M}\sum_{m=1}^{M}(\nabla_x \log f(x_m))(\nabla_x \log f(x_m))^\top \tag{2.26}$$

Algorithm 2 projected Stein Variational Gradient Descent

```
Input: samples $\{x_n^0\}_{n=1}^N\}$, basis $\Psi_r$, maximum iteration $L_{\max}$,tolerance $w_{tol}$.
Output: posterior samples $\{x_n^*\}_{n=1}^N$.
  Set $\ell = 0$, project $w_n^0 = \Psi_r^\top x_n^0, x_n^\perp = x_n^0 - \Psi_r w_n^0$.
  repeat
      Compute gradients $\nabla_{w_n^\ell} \log \pi(w_n^\ell)$ by 2.25 for $n = 1, \cdots, N$.
      Compute the kernel values $k^r(w_n^\ell, w_m^\ell)$ and their gradients $\nabla_{w_n^\ell} k^r(w_n^\ell, w_m^\ell)$ for $n = 1, \cdots, N, m = 1, \cdots, N$.
      Update samples $w_m^{\ell+1}$ from $w_m^\ell$ by Algorithm 1, i.e.
          $$\{w_m^{\ell+1}\}_{n=1}^N = SVGD(\{w_m^\ell\}_{n=1}^N) \quad (2.27)$$
      Set $\ell \leftarrow \ell + 1$
  until $\ell \geq L_{\max}$ or mean$(\|w_m^\ell - w_m^{\ell+1}\|_2) \leq w_{tol}$.
  Reconstruct samples $x_n^* = \Psi_r w_n^\ell + x_n^\perp$.
```

Algorithm 3 Adaptive projected Stein Variational Gradient Descent

```
Input: samples $\{x_n^0\}_{n=1}^N, L_{\max}^x, L_{\max}^w, x_{tol}, w_{tol}$.
Output: posterior samples $\{x_n^*\}_{n=1}^N$.
  Set $\ell_x = 0$
  repeat
      Compute $\nabla_x \log f(x_n^{\ell_x})$ in 2.26 for $n = 1, \cdots, N$.
      Solve 2.15 with $H$ approximated as in 2.26, to get bases $\Psi_r^{\ell_x}$.
      Apply the pSVGD Algorithm 2, i.e.
          $$\{x_n^*\}_{n=1}^N = \text{pSVGD}(\{x_n^{\ell_x}\}_{n=1}^N, \Psi_r^{\ell_x}, L_{\max}^w, w_{tol}). \quad (2.28)$$
      Set $\ell_x \leftarrow \ell_x + 1$ and $x_n^{\ell_x} = x_n^*, n = 1, \cdots, N$.
  until $\ell_x \geq L_{\max}^x$ or mean$(\|x_m^{\ell_x} - x_m^{\ell_x - 1}\|_X) \leq x_{tol}$.
```

Chapter 3

Our Code

In this chapter, we only listed the main code for three algorithms given above, including code of SVGD, pSVGD and apSVGD. The most part concent of code can be seen at https://github.com/kylekaston/Kyle.git in the form of R package.

3.1 SVGD

```
svgd=function(y,x0,lnprob,n_iter=1000,stepsize=1e-3,bandwidth=-1,alpha=0.9,dtol=1e-3){
  theta=x0
  thetaold=2*x0
  fudge_factor=1e-6
  historical_grad=0
  for(iter in 1:n_iter){
    if(iter%%100==0){
      d=diter(theta-thetaold)
      if(d<dtol) break
      thetaold=theta
    }
    lnpgrad=lnprob(y,theta)
    A=kernel(theta,h=-1)
    Kxy=A[[1]];dxkxy=A[[2]]
    grad_theta=(Kxy%*%lnpgrad+dxkxy)/nrow(x0)
    if(iter==0) historical_grad=historical_grad+grad_theta^2
    else historical_grad=alpha*historical_grad+(1-alpha)*(grad_theta^2)
    adj_grad=grad_theta/(fudge_factor+historical_grad^0.5)
    theta=theta+stepsize*adj_grad
  }
  return(theta)
}
```

3.2 pSVGD

```
psvgd=function(y,x0,phir,lnprob,n_iter=1000,stepsize=1e-3,alpha=0.9,wtol=1e-3){
  w0=x0%*%phir
  x_vert=x0-w0%*%t(phir)
  dln_pi=function(y,w) lnprob(y,w%*%t(phir)+x_vert)%*%phir
  wp=svgd(y,w0,dln_pi,n_iter=n_iter,stepsize=stepsize,bandwidth=bandwidtn,alpha=alpha,dtol=wtol)
  x=wp%*%t(phir)+x_vert
  return(x)
}
```

3.3 apSVGD

```
adpt_psvgd <- function(y,x0,f,lxmax,lwmax,r,xtol=1e-3,wtol=1e-3){
  lx = 0
  x = x0
  nrx = nrow(x)
  ncx = ncol(x)
  xold = 2*x
  dist = 1
  hesse = diag(x=0,ncol = nrx,nrow = nrx)
  while(lx<lxmax){
    d=diter(xold-x)
    if(lx%%10==0){
      if(d<xtol) break
    }
    xold = x
    hs=f(y,x)
    hesse=t(hs)%*%hs
    hesse = hesse/ncx
    eigen <- eigen(hesse)
    phi = eigen$vector[,1:r]
    x <- psvgd(y=y,x0=x,phir=phi,lnprob=f,n_iter=lwmax,wtol=wtol)
    lx <- lx+1
  }
  return(x)
}
```

3.4 Update by Rcpp

In consideration of the poor computational performance of apSVGD with R, we rewrite the codes of iteration and eigenpair solver by C++. As we can see in the next chapter, the final result turns out to be much better when using Rcpp. Note that due to the limited time, we didn't complete our R package with Rcpp. Instead, we merely test the time efficiency with Rcpp code.

```r
require('Rcpp')
require('inline')
require('RcppEigen')
eig<-'
using namespace Eigen;
const Map<MatrixXd> A(as<Map<MatrixXd>>(As));
SelfAdjointEigenSolver<MatrixXd>es(A);
if(es.info()!=Success) stop("Problem with Matrix");
return List::create(Named("values")=es.eigenvalues(),Named("vectors")=es.eigenvectors());
'
eigEx<-cxxfunction(signature(As="mat"),body=eig,plugin = "RcppEigen")

src='
Rcpp::NumericMatrix xx= Rcpp::clone<Rcpp::NumericMatrix>(x);
Rcpp::NumericMatrix xxold=Rcpp::clone<Rcpp::NumericMatrix>(xold);
Rcpp::Function e(eg);
Rcpp::Function di(diter);
int lxma =Rcpp::as<int>(lxmax);
int lxmax = lxma;
double xt= Rcpp::as<double>(xtol);
double dp;
int lx=0;
for(lx=1;lx<lxmax;lx++){
xxold=xx;
xx=e(A,y,xx,r);

}
return xx;'

eg=function(A,y,x,r){
  hs=f(A,y,x)
  hesse=t(hs)%*%hs
  hesse = hesse/nrow(x)
  eigen = eigEx(hesse)
  phi = eigen$vectors[,(ncol(hesse)-r+1):ncol(hesse)]
  x <- psvgd(y=y,x0=x,phir=phi,lnprob=f,n_iter=1000,wtol=1e-3)
  return(x)
}

apSVGD_cpp=cxxfunction(signature(y="ANY",A="ANY",x="ANY",xold='ANY',xtol='ANY',
                        diter="function", eg="function",lxmax="ANY",r="ANY"),src,
                        plugin='Rcpp')
```

3.5 An example using package: multivariate Normal Model

```
n=100      # 100 dimension normal distributio
theta=matrix(0,10,n) n
for(i in 1:n) theta[,i]=rnorm(10,2*i-1)
mu=matrix(1:n,1,n)  # the sample has been generated

library(psvgd)  #our package with svgd related functions
p=adpt_psvgd(y=mu,x0=theta,f=dlnprob,lxmax=6000,lwmax=1000,r=10)
#then p is the sample (from theta)already been regenerated by psvgd
q=svgd(mu,theta,dlnprob,100000)
# and q is sample regenerated by svgd(lower case is because the function--
#-named in package is lower case for writing consistence)
```

lxmax=6000, lwmax=1000 enables the result as generated ones in the situation of 100 dimensions. And all functions are compiled as the most effient form, with result in the 5.3 part by changing inputs: lxmax, lwmax; using two algorithms as a comparison in time(by compare p and q, along with the CPU time), and itieration needed to reach the convergence.

Chapter 4

Experiments

In this chapter, we implemented numerical experiments with Gaussian Mixture Model and Multivariate Normal Model, and compared our methods with EM algorithm. It turns out that EM algorithm faces the challenge of sensitive dependence on initial conditions in Gaussian Mixture Model. In addition, we compared the time efficiency of SVGD with pSVGD in two ways: in the same iterations and in the same tolerance.

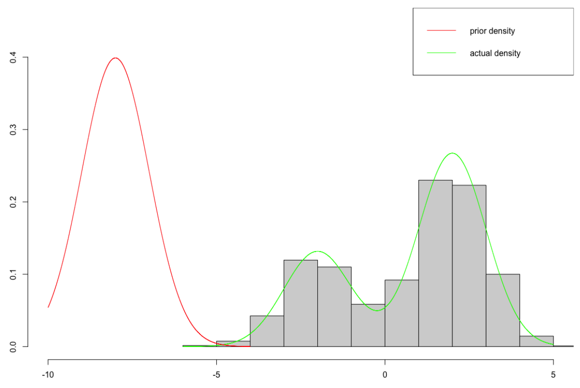
4.1 Gaussian Mixture Model

Let $\{y_i\}$ is a set of i.i.d samples generated by a Gaussian mixture model

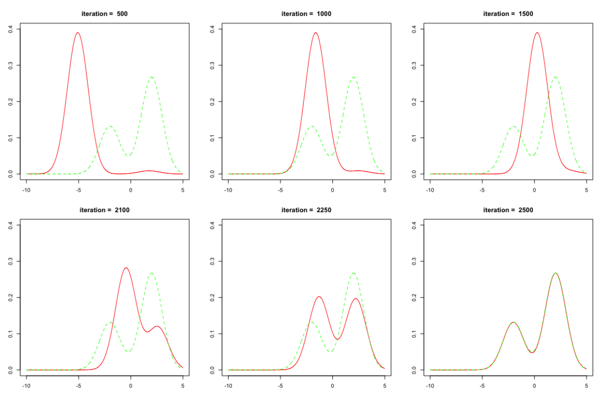
$$f(y_i) = \alpha\mathcal{N}(y_i; \mu_1, 1) + (1-\alpha)\mathcal{N}(y_i; \mu_2, 1), \quad i = 1, \cdots, 2000$$

where $\mathcal{N}$ represents the density of a normal distribution. Set $\mu_1 = -2, \mu_2 = 2, \alpha = 0.33$.

Set prior density of $\mu_1 \sim N(-8, 1), \mu_2 \sim N(4, 1)$. The number of parameter particles is 10.



(a)



(b)

Figure 4.1: (a): initial conditions of prior density (b): posterior density during iteration

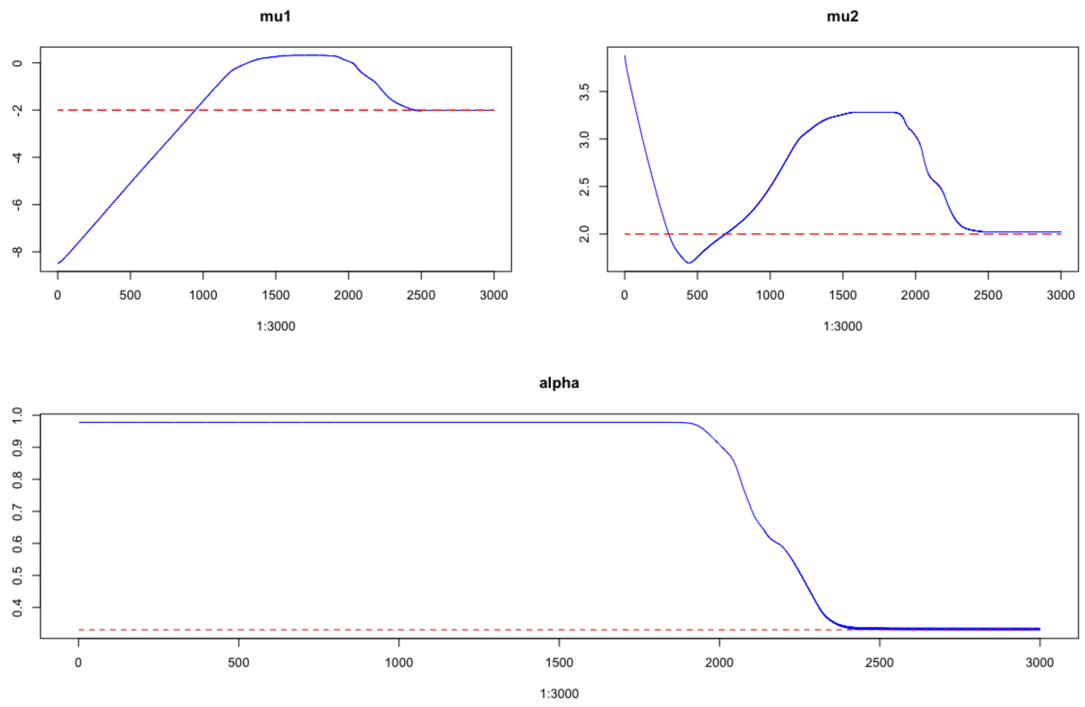


Figure 4.2: parameter value during iteration

From figure 4.1(a) we can tell that the choose of initial condition is extraordinarily ill since the probability mass of $p(x)$ and $q_0(x)$ are far away each other (with almost zero overlap). Figure 4.1(b) showshow the distribution of the particles change as the iteration goes. We see that despite the small overlap between $q_0(x)$ and $p(x)$, SVGD can push the particle towards the target distribution, and the final result meets our expectation.

Figure 4.2 shows how the parameter changes step by step - first μ_1 and μ_2 and then α. After about 2500 iterations the parameter converges to real value.

4.2 Comparison with EM

We set the same distribution of $\{y_i\}_{n=1}^{2000}$ as in former section, and did the same numerical experiment using EM algorithm.

From figure 4.3 we can tell that when the initial condition is good enough, the speed of EM algorithm in GMM model is really fast - after about thirty iterations, μ_1 and μ_2 converge to the real value. However, EM algorithm faces the challenge of sensitive dependence on initial condition. EM algorithm converges to a false value when we change the initial value of α from 0.9 to 0.91. This is because EM algorithm only maximizes the target function $Q(\theta|\theta_n) = \mathbf{E}[\log f(X|\theta)|Y = y, \theta_n]$ at each step. This procedure finds the optimal θ informing the maximum likelihood estimation in the sense of expectation, or in other words, mean value, rather than the distribution itself. So when the question is not convex, the result may be totally different with true value. However, SVGD measures two distributions with KL divergence, and the metric vanishes iff the two distributions are equal. This property guarantees a better performance of SVGD method.

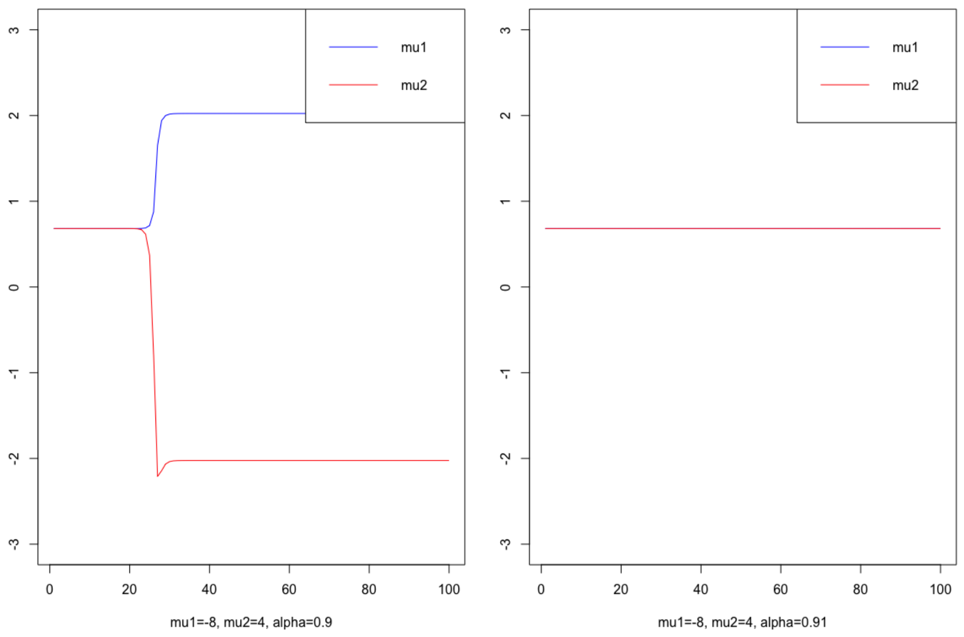


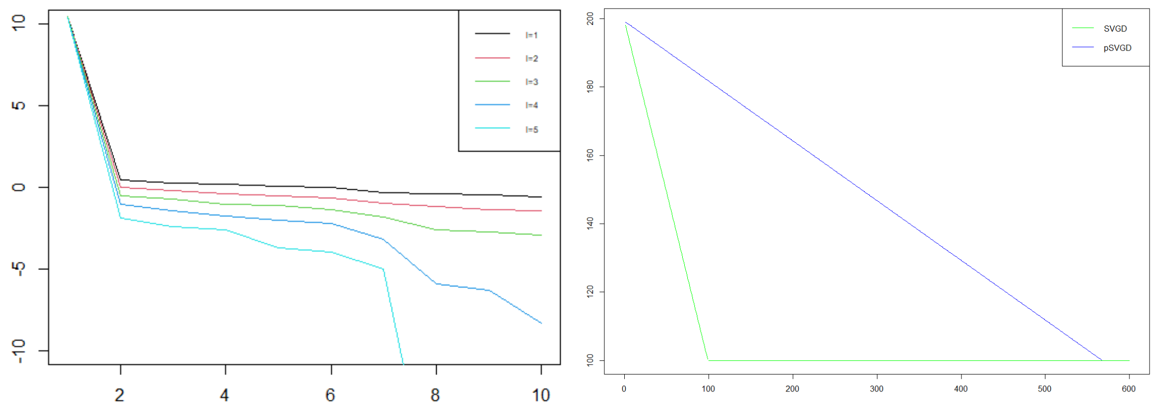
Figure 4.3: the performance of EM algorithm in different initial conditions

4.3 Multivariate Normal Model

We use both SVGD and pSVGD in Multivariate Normal Model, and compares their behavior. Set Set $y \sim N(\mu, I_{100}), i = 1, 2, \cdots, n$, where $\mu = (1, 2, \cdots, 100)$.

Figure 4.4 (a) shows the biggest log eigenvalues in first 5 iterations. It's shown that the first eigenvalue remains relatively still, while smaller ones decrease fast, even in first several iterations. This gives us the information that it's feasible to project 100-dimensional parameter space to a 10-dimensional subspace. Figure 4.4(b) is a comparison of iteration speed of SVGD and pSVGD. SVGD is much quicker, for pSVGD sacrifices the number of iterations for the efficiency in per steps.

Note that Table 4.1 and 4.2 are based on **Rcpp** codes. Table 4.1 shows the time spent in the same number of iterations. The dimension of parameter is fixed, that is, 100. One can say in most cases pSVGD is better than SVGD, even in low-dimensional cases. This isn't surprising since pSVGD reduced the dimension of kernel from 100×100 to 10×10. Table 4.2 shows the time spent in the same tolerance 10^{-3}, where the dimension of parameters differs. When the dimension is relatively low - lower than 100, SVGD is better than pSVGD since it needs fewer iterations. However, as dimension increases, pSVGD performs better, due to the lower time spent in calculating the kernel matrix. When dealing with high-dimensional data, such as more than 1000 dimensions, pSVGD is extraordinarily faster than SVGD.



(a) (b)

Figure 4.4: (a): top 10 log eigenvalues in first 5 iterations (b): iteration speed of SVGD and pSVGD

Table 4.1: CPU time of SVGD and pSVGD in same iterations

	time(s) \ l_w / l_x	1	10	100	1000
SVGD	100	0.08	0.79	7.59	75.22
pSVGD		0.20	0.61	4.15	37.02
SVGD	500	0.37	3.81	37.63	376.55
pSVGD		0.96	2.32	18.26	179.5
SVGD	1000	0.79	7.59	75.22	752.11
pSVGD		2.04	5.45	39.08	329.14

Table 4.2: CPU time of SVGD and pSVGD in same tolerance

d / time(s)	10	50	100	200	400
SVGD	4.43	42.39	124.7	424.31	1587.92
pSVGD	8.27	70.61	152.6	362.37	834.98