Problem 1

Similar to the proof in the LDA paper, we derive the ELBO for smoothed LDA first, then show the update formula for λ, γ, ϕ

The ELBO for smoothed LDA is shown below, notice the first 5 terms are the same as LDA paper, and the last two are from the smoothed LDA assumption: $\beta \sim Dirichlet(\eta)$

$$\begin{aligned} L(\lambda, \gamma, \phi; \alpha, \eta) &= \log p(w|\alpha, \eta) - KL(q(\beta, \theta, z|\lambda, \gamma, \phi)||p(\beta, \theta, z|w, \alpha, \eta)) \\ &= E_q \log p(\theta|\alpha) + E_q \log p(z|\theta) + E_q \log p(w|z, \beta) - E_q \log q(\theta) - E_q \log q(z) \\ &+ E_q \log p(\beta|\eta) - E_q \log q(\beta) \end{aligned} \tag{1}$$

As shown in LDA paper appendix A.1(, and shown in class), Dirichlet distribution belongs to exponential family with natural parameter $\alpha - 1$ and sufifcient statistic $\log x$

$$\begin{aligned} f(x|\alpha) &= \frac{1}{B(\alpha)} \prod x_i^{\alpha_i - 1} \\ &= \exp\{\sum_i (\alpha_i - 1) \log x_i + \log \Gamma(\sum_i \alpha_i) - \sum_i \log \Gamma(\alpha_i)\} \end{aligned} \tag{2}$$

, and we have the below formula for exponential family.

$$\frac{d}{d\eta(\alpha)} A(\alpha) = E_{p(x)} T(x) \tag{3}$$

Thus, we show the explicit form for the sixth term

$$\begin{aligned} E_q \log p(\beta|\eta) &= E_q \log \prod_k \frac{\Gamma(\sum_i \eta_i)}{\prod_i \Gamma(\eta_i)} \prod_i \beta_{k,i}^{\eta_i - 1} \\ &= K \log \Gamma(\sum_i \eta_i) - K \sum_i \log \Gamma(\eta_i) + \sum_k E_q[\sum_i (\eta_i - 1) \log \beta_{k,i}] \\ &= K \log \Gamma(\sum_i \eta_i) - K \sum_i \log \Gamma(\eta_i) + \sum_k \sum_i (\eta_i - 1) E_q[\log \beta_{k,i}] \\ &= K \log \Gamma(\sum_i \eta_i) - K \sum_i \log \Gamma(\eta_i) + \sum_k \sum_i (\eta_i - 1) \frac{d}{d\lambda_{k,i}} (\log \Gamma(\lambda_{k,i}) - \log \Gamma(\sum_j \lambda_{k,j})) \\ &= K \log \Gamma(\sum_i \eta_i) - K \sum_i \log \Gamma(\eta_i) + \sum_k \sum_i (\eta_i - 1)(\Psi(\lambda_{k,i}) - \Psi(\sum_j \lambda_{k,j})) \end{aligned} \tag{4}$$

Similarly, we have

$$\begin{aligned} E_q \log p(\theta|\alpha) &= \log \Gamma(\sum_i \alpha_i) - \sum_i \log \Gamma(\alpha_i) + \sum_i (\alpha_i - 1)(\Psi(\gamma_i) - \Psi(\sum_j \gamma_j)) \\ E_q \log p(z|\theta) &= \sum_d \sum_n \sum_i \phi_{d,n,i} (\Psi(\gamma_{d,i}) - \Psi(\sum_j \gamma_{d,j})) \\ E_q \log p(w|z, \beta) &= \sum_d \sum_n \sum_i \sum_j \phi_{d,n,i} w_{d,n,j} (\Psi(\lambda_{k,j}) - \Psi(\sum_k \lambda_{i,k})) \\ E_q \log q(\theta) &= \sum_d (\log \Gamma(\sum_j \gamma_{d,j}) - \sum_i \log \Gamma(\gamma_{d,i}) + \sum_i (\gamma_{d,i} - 1)(\Psi(\gamma_{d,i}) - \Psi(\sum_j \gamma_{d,j}))) \\ E_q \log q(z) &= \sum_d \sum_n \sum_i \phi_{d,n,i} \log \phi_{d,n,i} \\ E_q \log q(\beta) &= \sum_k (\log \Gamma(\sum_i \lambda_{k,i}) - \sum_i \log \Gamma(\lambda_{k,i}) + \sum_i (\lambda_{k,i} - 1)(\Psi(\lambda_{k,i}) - \Psi(\sum_j \lambda_{k,j}))) \end{aligned} \tag{5}$$

Problem 1 continued on next page...

(1)
We take the relevant terms w.r.t. $\phi_{d,n,i}$ with a Lagrange multiplier $\lambda(\sum_i \phi_{d,n,i} - 1)$, since $\sum_i \phi_{d,n,i} = 1$

$$\begin{aligned} L = & \phi_{d,n,i}(\Psi(\gamma_{d,i}) - \Psi(\sum_j \gamma_{d,j})) \\ & + \phi_{d,n,i} \sum_j w_{d,n,j}(\Psi(\lambda_{k,j}) - \Psi(\sum_k \lambda_{i,k})) \\ & - \phi_{d,n,i} \log \phi_{d,n,i} \\ & + \lambda(\sum_i \phi_{d,n,i} - 1) \end{aligned} \tag{6}$$

and set it to zero, we have

$$\phi_{d,n,i} \propto \exp(\Psi(\gamma_{d,i}) - \Psi(\sum_j \gamma_{d,j}) + \sum_j w_{d,n,j}(\Psi(\lambda_{k,j}) - \Psi(\sum_k \lambda_{i,k}))), \tag{7}$$

the $\Psi(\sum_j \gamma_{d,j})$ term could be removed, since it is constant for fixed d, thus does not matter after normalizing ϕ. Similarly, we have

$$\begin{aligned} \lambda_i &= \eta + \sum_d \sum_n \phi_{d,n,i} w_{d,n} \\ \gamma_d &= \alpha + \sum_n \phi_{d,n,i} \end{aligned} \tag{8}$$

(2)
Already shown in Equation(1, 4, 5)
(3), (4)
The vocabulary size is 100.
We show below the negative ELBO as a function of epoch and batch size

- Batched LDA's performance is greatly influenced by initialization, while full-batched LDA is not
- full batched LDA outperforms batched version consistently, almost regradless of initialization

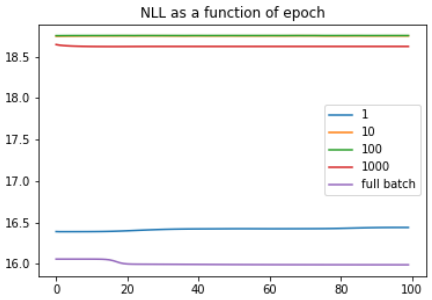

Figure 1: results of ELBO

Problem 2

(1)
The ELBO for logistic regression is

$$
\begin{aligned}
L(\mu,\sigma^2) &= E_q \log p(x,\beta) - \log q(\beta|\mu,\sigma^2) \\
\nabla_{\mu,\sigma^2} L &= \nabla_{\mu,\sigma^2} \int q(\beta,|\mu,\sigma^2)(\log p(x,\beta) - \log q(\beta|\mu,\sigma^2))d\beta \\
&= \int q(\beta|\mu,\sigma^2)\nabla_{\mu,\sigma^2} \log q(\beta,|\mu,\sigma^2)(\log p(x,\beta) - \log q(\beta|\mu,\sigma^2)) \\
&\quad - q(\beta|\mu,\sigma^2)\nabla_{\mu,\sigma^2} \log q(\beta,|\mu,\sigma^2)d\beta \\
&= E_q \nabla_{\mu,\sigma^2} \log q(\beta|\mu,\sigma^2)(\log p(x,\beta) - \log q(\beta|\mu,\sigma^2) - 1) \\
&= E_q \nabla_{\mu,\sigma^2} \log q(\beta|\mu,\sigma^2)(\log p(x,\beta) - \log q(\beta|\mu,\sigma^2)) \\
\log p(x,\beta) &= \sum_i y_i \log \sigma(\beta^T x_i) + (1-y_i)\log(1-\sigma(\beta^T x_i)) + \log N(\beta|0,1) \\
\log q(\beta|\mu,\sigma^2) &= \log N(\beta|\mu,\sigma^2)
\end{aligned} \tag{9}
$$

Now we only need to solve for $\nabla_{\mu,\sigma^2} \log q(\beta|\mu,\sigma^2)$.

$$
\begin{aligned}
\log q(\beta|\mu,\sigma^2) &= \log N(\beta|\mu,\sigma^2) \\
&= -\frac{D\log\sigma^2}{2} - \frac{\|\beta-\mu\|_2}{2\sigma^2} \\
\nabla_{\mu_i} \log q(\beta|\mu,\sigma^2) &= \frac{\beta_i - \mu_i}{\sigma^2} \\
\nabla_{\sigma^2} \log q(\beta|\mu,\sigma^2) &= -\frac{D}{2\sigma^2} + \frac{\|\beta-\mu\|_2}{2(\sigma^2)^2}
\end{aligned} \tag{10}
$$

By substituting the above equation into (9), we derive the score function estimator for the gradient of ELBO w.r.t. μ and σ.

(2)

We use $\nabla_{\mu,\sigma^2} \log q(\beta,|\mu,\sigma^2)$ to control variation, which is also adopted in BBVI paper.

As for implementation details, I used Adam for optimizing BBVI(vanilla bbvi), BBVI with control variates(bbvi cv) and BBVI with reparameterization trick(bbvi rt). The results in ELBO(log -ELBO) are shown below.

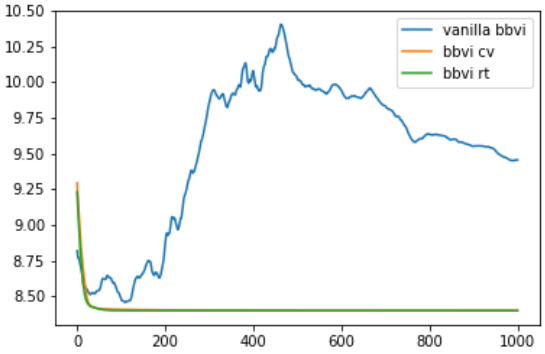

Figure 2: results of various BBVI

(3)

We have

$$
\begin{aligned}
\nabla_{\mu,\sigma^2} L &= \nabla_{\mu,\sigma^2} E_q \log p(x,\beta) - \log q_{\mu,\sigma^2}(\beta) \\
&= E_{q(\epsilon)} \nabla_{\mu,\sigma^2} \log p(x, g_{\mu,\sigma^2}(\epsilon)) - \log q_{\mu,\sigma^2}(g_{\mu,\sigma^2}(\epsilon))
\end{aligned} \tag{11}
$$

For logistic regression, we have (note that σ is short for $\sigma((\mu+\sigma\epsilon)^T x_i)$ except in $\mu+\sigma\epsilon$)

$$
\begin{aligned}
\log p(x, g_{\mu,\sigma^2}(\epsilon)) &= \sum_i y_i \log\sigma + (1-y_i)\log(1-\sigma) + \log N(\mu+\sigma\epsilon|0,1) \\
\nabla_\mu \log p(x, g_{\mu,\sigma^2}(\epsilon)) &= \sum_i y_i \frac{\sigma(1-\sigma)}{\sigma} x_i + (1-y_i)\frac{-\sigma(1-\sigma)}{1-\sigma} x_i - (\mu+\sigma\epsilon) \\
&= \sum_i y_i(1-\sigma)x_i + (y_i-1)\sigma x_i - (\mu+\sigma\epsilon) \\
\nabla_{\sigma^2} \log p(x, g_{\mu,\sigma^2}(\epsilon)) &= \{\sum_i y_i \frac{\sigma(1-\sigma)}{\sigma}\epsilon x_i + (1-y_i)\frac{-\sigma(1-\sigma)}{1-\sigma}\epsilon x_i - (\mu+\sigma\epsilon)\epsilon\}\frac{d\sigma}{d(\sigma^2)} \\
&= (\sum_i y_i(1-\sigma)x_i + (y_i-1)\sigma x_i - (\mu+\sigma\epsilon))\frac{\epsilon}{2\sqrt{\sigma^2}} \\
\log q_{\mu,\sigma^2}(g_{\mu,\sigma^2}(\epsilon)) &= \log N(\mu+\sigma\epsilon|\mu,\sigma^2) \\
&= -\frac{D}{2}\log\sigma^2 + C \\
\nabla_{\sigma^2} \log q_{\mu,\sigma^2}(g_{\mu,\sigma^2}(\epsilon)) &= -\frac{D}{2\sigma^2}
\end{aligned}
\tag{12}
$$

The results of BBVI with reparameterization trick is shown in **(2)**. Here we show the performance in minibatch senario.

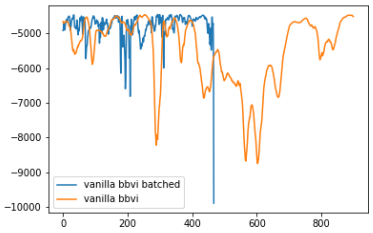

Figure 3: results of vanilla BBVI

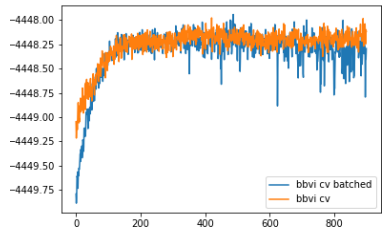

Figure 4: results of BBVI cv

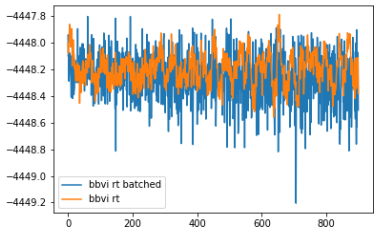

Figure 5: results of BBVI rt

The settings are: batch size being 1000, sample result every 10 iteration *i.e.* one epoch. Notice that vanilla BBVI diverged after 500 epoch even in such a large batch size. For BBVI with control variates and reparameterization trick, we only show their ELBO after 100 epoch to show their performance after convergence.

(4)

We show the result w.r.t. $Var(\mu)$ and $Var(\sigma)$ below. Notice that in my implementation, we estimate $\nabla_{\log\sigma^2}$ instead. As batch size increases, BBVI+CV/RT reduces the variances more and the variance becomes more stable.

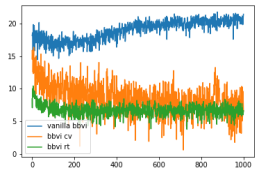

Figure 6: results of μ, 4 samples

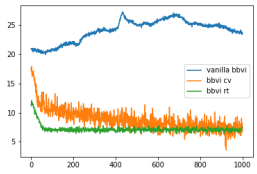

Figure 7: results of μ, 32 samples

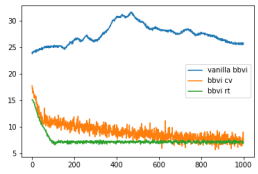

Figure 8: results of μ, 64 samples

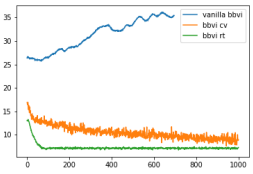

Figure 9: results of μ, 128 samples

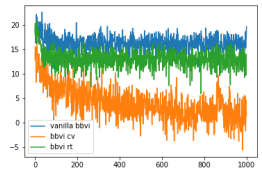

Figure 10: results of σ, 4 samples

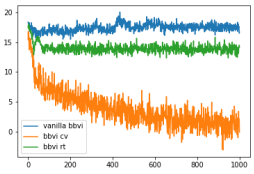

Figure 11: results of σ, 32 samples

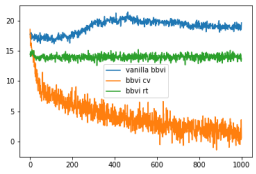

Figure 12: results of σ, 64 samples

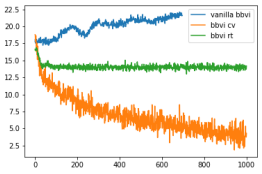

Figure 13: results of σ, 128 samples

(6)

NOTE: no better results made!

- replace σ with σ_1, σ_2 theortically should perform no worse than a single σ, but in practice not(ELBO drops to -1460)
- replace Gaussian prior by t-distribution or Laplace distribution does not work neither(ELBO drops to -1480)

```
import numpy as np
from scipy.special import digamma as dga
from scipy.special import gamma as ga
from scipy.special import loggamma as lga
```

```
eps=1e-10
def log(x):
    return np.log(x + eps)

def digamma(x):
    return dga(x + eps)

def loggamma(x):
    return lga(x + eps)
```

```
def init(data):
    vocab = np.array([i for i in range(100)])

    num_doc = data.shape[0]
    num_vocab = vocab.shape[0]
    len_doc = data.shape[1]
    num_topic = 10

    w = np.zeros([num_doc, len_doc, num_vocab])
    for d in range(num_doc):
        for n in range(len_doc):
            w[d, n, data[d, n]] = 1

    alpha = np.ones(shape=num_topic)
    eta = np.ones(shape=num_vocab)

    phi = np.random.rand(num_doc, len_doc, num_topic)
    for d in range(num_doc):
        for n in range(len_doc):
            phi[d, n] /= np.sum(phi[d, n])

    gam = np.random.rand(num_doc, num_topic)
    gam /= np.sum(gam, axis=1)[:, np.newaxis]

    lam = np.random.rand(num_topic, num_vocab)
    lam /= np.sum(lam, axis=1)[:, np.newaxis]
    return lam, gam, phi, w, num_doc, num_topic, num_vocab, len_doc, alpha, eta
```

```
def one_step(lam, gam, phi, w, num_doc, num_topic, num_vocab, len_doc, alpha, eta):
    #print(num_doc, num_topic, num_vocab)
    for k in range(num_topic):
```

```
        lam[k] = eta
        for d in range(num_doc):
            for n in range(len_doc):
                lam[k] += phi[d, n, k] * w[d, n]
    #lam /= np.sum(lam, axis=1)[:, np.newaxis]

    gam = alpha + np.sum(phi, axis=1)
    #gam /= np.sum(gam, axis=1)[:, np.newaxis]

    def get_single_doc(lam, gam, phi, w, d):
        for n in range(len_doc):
            #phi[d, n, :] = np.exp(digamma(gam[d, :]) + digamma(lam[:, data[d, n]]) -
digamma(np.sum(lam, axis=1)))
            for k in range(num_topic):
                phi[d, n, k] = np.exp(digamma(lam[k, data[d, n]]) -
digamma(np.sum(lam[k])) + digamma(gam[d, k]) - digamma(np.sum(gam[d])))
            phi[d, n, :] /= np.sum(phi[d, n, :])
        return phi[d], d

    with concurrent.futures.ThreadPoolExecutor(max_workers=8) as executor:
        future_list = [executor.submit(get_single_doc, lam, gam, phi, w, d) for d in
range(num_doc)]
        for future in concurrent.futures.as_completed(future_list):
            phi_d, d = future.result()
            phi[d] = phi_d

    return lam, gam, phi, w, num_doc, num_topic, num_vocab, len_doc, alpha, eta
```

```
import concurrent.futures

def get_res1(lam, gam, phi, w):
    res_1 = 0.0
    res_1 += num_topic * loggamma(np.sum(eta))
    res_1 -= num_topic * np.sum(loggamma(eta))
    '''
    for k in range(num_topic):
        for i in range(num_vocab):
            res_1 += (eta[i] - 1) * (digamma(lam[k, i]) - digamma(np.sum(lam[k])))
    '''
    return res_1


def get_res2(lam, gam, phi, w):
    res_2 = 0.0
    for n in range(len_doc):
        for k in range(num_topic):
            res_2 += phi[:, n, k] * (digamma(gam[:, k]) - digamma(np.sum(gam, axis=1)))
    #res_2 -= digamma(np.sum(gam, axis=1))
    res_2 = np.sum(res_2)
    return res_2


def get_res3(lam, gam, phi, w):
```

```
    res_3 = 0.0
    res_3 += loggamma(np.sum(alpha))
    res_3 -= np.sum(loggamma(alpha))
    '''
    for k in range(num_topic):
        res_3 += (alpha[k] - 1) * (digamma(gam[:, k] - digamma(np.sum(gam[:, k]))))
    '''
    res_3 = np.sum(res_3)
    return res_3

def get_res4(lam, gam, phi, w):
    res_4 = 0.0
    def get_res4_single_loc(lam, gam, phi, w, n):
        res_loc = 0.0
        for k in range(num_topic):
            sum_lam_k = np.sum(lam[k])
            for i in range(num_vocab):
                res_loc += phi[:, n, k] * w[:, n, i] * (digamma(lam[k, i]) -
digamma(sum_lam_k))
        res_loc = np.sum(res_loc)
        return res_loc

    with concurrent.futures.ThreadPoolExecutor(max_workers=32) as executor:
        future_list = [executor.submit(get_res4_single_loc, lam, gam, phi, w, n) for n
in range(len_doc)]
        for future in concurrent.futures.as_completed(future_list):
            res_4 += future.result()
    return res_4

def get_res5(lam, gam, phi, w):
    res_5 = 0.0
    for k in range(num_topic):
        res_5 += loggamma(np.sum(lam[k])) - np.sum(loggamma(lam[k]))
    for k in range(num_topic):
        sum_lam_k = np.sum(lam[k])
        #'''
        res_5 += np.sum((lam[k] - 1) * (digamma(lam[k]) - digamma(sum_lam_k)))
        '''
        for i in range(num_vocab):
            res_5 += (lam[k, i] - 1) * (digamma(lam[k, i]) - digamma(sum_lam_k))
        '''
    return -res_5

def get_res6(lam, gam, phi, w):
    res_6 = 0.0
    res_6 += np.sum(phi * log(phi))
    return -res_6

def get_res7(lam, gam, phi, w):
    res_7 = 0.0
    res_7 += loggamma(np.sum(gam, axis=1)) - np.sum(loggamma(gam), axis=1)
    #print(res_7)
    res_7 = np.sum(res_7)
```

```
    for d in range(num_doc):
        res_7 += np.sum((gam[d] - 1) * (digamma(gam[d]) - digamma(np.sum(gam[d]))))
    return -res_7

def elbo(lam, gam, phi, w):
    res = 0.0
    func_list = [get_res1, get_res2, get_res3, get_res4, get_res5, get_res6, get_res7]
    with concurrent.futures.ThreadPoolExecutor(max_workers=8) as executor:
        future_list = [executor.submit(func, lam, gam, phi, w) for func in func_list]
        for future in concurrent.futures.as_completed(future_list):
            res += future.result()

    return res
```

```
elbo_list = []
if True:
    data = np.load("mcs_hw4_p1_lda.npy")
    lam, gam, phi, w, num_doc, num_topic, num_vocab, len_doc, alpha, eta = init(data)
    for i in range(100):
        lam, gam, phi, w, num_doc, num_topic, num_vocab, len_doc, alpha, eta =
one_step(lam, gam, phi, w, num_doc, num_topic, num_vocab, len_doc, alpha, eta)
        #print("iteration " + str(i) + " done")
        elbo_per_point = elbo(lam, gam, phi, w)
        elbo_list.append(elbo_per_point)
        print(elbo_per_point)
```

```
import numpy as np
```

```
data = np.load("mcs_hw2_p3_data.npy")
```

```
x = data[:, :2]
y = data[:, 2]
```

```
import scipy.stats

def get_gradient_mu(beta, mu, sigma2):
    return (beta - mu) / sigma2

def get_gradient_logsigma2(beta, mu, sigma2):
    norm = np.linalg.norm(beta - mu)
    return (- 1 / sigma2 + norm * norm / (2 * sigma2 * sigma2)) * sigma2

def get_gradient_mu_t(beta, mu, sigma2, v):
    res = -(v + 2) / 2
    norm = np.linalg.norm(beta - mu)
    res *= 1 / (1 + norm * norm / (v * sigma2))
    res *= 2 * (mu - beta) / (v * sigma2)
    return res

def get_gradient_logsigma2_t(beta, mu, sigm2, v):
    res_1 = - 1 / sigma2
    norm = np.linalg.norm(beta - mu)
    res_2 = (v + 2) / 2
    res_2 *= 1 / (1 + norm * norm / (v * sigma2))
    res_2 *= norm * norm / v
    res_2 *= -1 / (sigma2 * sigma2)
    return res_1 - res_2

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def get_log_p(x, y, beta):
    res = 0.0
    res += np.sum(y * np.log(sigmoid(np.dot(x, beta))) + (1.0 - y) * np.log(1 -
sigmoid(np.dot(x, beta))))
    res += np.sum(scipy.stats.norm.logpdf(beta, np.zeros(2), np.ones(2)))
    return res

def get_log_q(mu, sigma2, beta):
    res = np.sum(scipy.stats.norm.logpdf(beta, mu, np.sqrt(sigma2)))
    return res
```

```
import concurrent.futures

def elbo(x, y, mu, sigma2):
    res = 0.0
    sample_size = 1024
    sample_beta = np.random.normal(mu, np.sqrt(sigma2), size=[sample_size,
mu.shape[0]])
    with concurrent.futures.ThreadPoolExecutor(max_workers=32) as executor:
        future_list = [executor.submit(get_log_p, x, y, beta) for beta in sample_beta]
        for future in concurrent.futures.as_completed(future_list):
            res += future.result()

    with concurrent.futures.ThreadPoolExecutor(max_workers=32) as executor:
        future_list = [executor.submit(get_log_q, mu, sigma2, beta) for beta in
sample_beta]
        for future in concurrent.futures.as_completed(future_list):
            res += future.result()

    return res / sample_size
```

```
def bbvi(x, y, mu, sigma2, lr, n_iter, m, v):
    sample_size = 4
    sample_beta = np.random.normal(mu, np.sqrt(sigma2), size=[sample_size,
mu.shape[0]])
    # update mu
    loss_mu = np.zeros(shape=[sample_size, mu.shape[0]])
    loss_logsigma2 = np.zeros(shape=[sample_size, sigma2.shape[0]])
    for i in range(sample_size):
        loss_mu[i] = get_gradient_mu(sample_beta[i], mu, sigma2)
        loss_logsigma2[i] = get_gradient_logsigma2(sample_beta[i], mu, sigma2)
        log_p = get_log_p(x, y, sample_beta[i])
        log_q = get_log_q(mu, sigma2, sample_beta[i])
        loss_mu[i] *= (log_p - log_q)
        loss_logsigma2[i] *= (log_p - log_q)
    update_mu = np.mean(loss_mu, axis=0)
    var_loss_mu = np.var(loss_mu, axis=0)
    update_logsigma2 = np.mean(loss_logsigma2, axis=0)
    var_loss_logsigma2 = np.var(loss_logsigma2, axis=0)

    grad = np.concatenate([update_mu, update_logsigma2])

    m = 0.9 * m + 0.1 * grad
    v = 0.999 * v + 0.001 * np.power(grad, 2)

    m_hat = m / (1 - np.power(0.9, n_iter))
    v_hat = v / (1 - np.power(0.999, n_iter))

    update = m_hat / (np.sqrt(v_hat) + 1e-10)

    mu += lr * update[:2]
    sigma2 = np.exp(np.log(sigma2) + lr * update[2])
    return mu, sigma2, m, v, var_loss_mu, var_loss_logsigma2
```

```
def train_bbvi(x, y, n_iter):
    mu_list = []
    sigma2_list = []
    var_list = []
    mu = np.random.normal(size=2)
    sigma2 = np.power(np.random.normal(size=1), 2)
    lr = 0.1
    m = np.zeros(shape=3)
    v = np.zeros(shape=3)
    for i in range(n_iter):
        mu, sigma2, m, v, var_mu, var_sigma = bbvi(x, y, mu, sigma2, lr, i + 1, m, v)
        mu_list.append(mu.copy())
        sigma2_list.append(sigma2.copy())
        var_list.append([var_mu.copy(), var_sigma.copy()])
    return mu_list, sigma2_list, var_list
```

```
def bbvi_cv(x, y, mu, sigma2, lr, n_iter, m, v):
    sample_size = 4
    sample_beta = np.random.normal(mu, np.sqrt(sigma2), size=[sample_size,
mu.shape[0]])
    # update mu
    loss_mu = np.zeros(shape=[sample_size, mu.shape[0]])
    loss_logsigma2 = np.zeros(shape=[sample_size, sigma2.shape[0]])
    cv_mu = np.zeros(shape=[sample_size, mu.shape[0]])
    cv_sigma2 = np.zeros(shape=[sample_size, sigma2.shape[0]])
    for i in range(sample_size):
        loss_mu[i] = cv_mu[i] = get_gradient_mu(sample_beta[i], mu, sigma2)
        loss_logsigma2[i] = cv_sigma2[i] = get_gradient_logsigma2(sample_beta[i], mu,
sigma2)
        log_p = get_log_p(x, y, sample_beta[i])
        log_q = get_log_q(mu, sigma2, sample_beta[i])
        loss_mu[i] *= (log_p - log_q)
        loss_logsigma2[i] *= (log_p - log_q)

    cov_mu0 = np.cov(np.stack((cv_mu.T[0], loss_mu.T[0]), axis=0))
    a_mu0 = cov_mu0[0][1] / cov_mu0[0][0]
    cov_mu1 = np.cov(np.stack((cv_mu.T[1], loss_mu.T[1]), axis=0))
    a_mu1 = cov_mu1[0][1] / cov_mu1[0][0]
    cov_logsigma2 = np.cov(np.stack((cv_sigma2.T[0], loss_logsigma2.T[0]), axis=0))
    a_logsigma2 = cov_logsigma2[0][1] / cov_logsigma2[0][0]

    update_mu = np.mean(loss_mu, axis=0)
    update_logsigma2 = np.mean(loss_logsigma2, axis=0)
    update_h_mu = np.mean(cv_mu, axis=0) * [a_mu0, a_mu1]
    update_h_logsigma2 = np.mean(cv_sigma2, axis=0) * a_logsigma2

    var_mu = np.var(loss_mu - cv_mu * [a_mu0, a_mu1], axis=0)
    var_sigma = np.var(loss_logsigma2 - cv_sigma2 * a_logsigma2, axis=0)

    grad = np.concatenate([update_mu - update_h_mu, update_logsigma2 -
update_h_logsigma2])
```

```
    m = 0.9 * m + 0.1 * grad
    v = 0.999 * v + 0.001 * np.power(grad, 2)

    m_hat = m / (1 - np.power(0.9, n_iter))
    v_hat = v / (1 - np.power(0.999, n_iter))

    update = m_hat / (np.sqrt(v_hat) + 1e-10)

    mu += lr * update[:2]
    sigma2 = np.exp(np.log(sigma2) + lr * update[2])
    #print(mu, sigma2)
    return mu, sigma2, m, v, var_mu, var_sigma
```

```
def train_bbvi_cv(x, y, n_iter):
    mu_list = []
    sigma2_list = []
    var_list = []
    mu = np.random.normal(size=2)
    sigma2 = np.power(np.random.normal(size=1), 2)
    lr = 0.1
    m = np.zeros(shape=3)
    v = np.zeros(shape=3)
    for i in range(n_iter):
        mu, sigma2, m, v, var_mu, var_sigma = bbvi_cv(x, y, mu, sigma2, lr, i + 1, m,
v)
        mu_list.append(mu.copy())
        sigma2_list.append(sigma2.copy())
        var_list.append([var_mu.copy(), var_sigma.copy()])
    return mu_list, sigma2_list, var_list
```

```
def get_gradient_mu_rt(x, y, mu, sigma2, eps):
    beta = mu + eps * np.sqrt(sigma2)
    data_part = (y * (1 - sigmoid(np.dot(x, beta))))[:, None] * x
    data_part += ((y - 1) * sigmoid(np.dot(x, beta)))[:, None] * x
    data_part = np.sum(data_part, axis=0)
    return data_part - beta
```

```
def get_gradient_logsigma2_rt(x, y, mu, sigma2, eps):
    res = 0.0
    beta = mu + eps * np.sqrt(sigma2)
    data_part = (y * (1 - sigmoid(np.dot(x, beta))))[:, None] * x
    data_part += ((y - 1) * sigmoid(np.dot(x, beta)))[:, None] * x
    data_part = np.sum(data_part, axis=0)
    res += (data_part - beta) * eps / (2 * np.sqrt(sigma2))
    res = np.sum(res)
    res += 1 / sigma2
    return res
```

```
def bbvi_rt(x, y, mu, sigma2, lr, n_iter, m, v):
    sample_size = 4
    sample_eps = np.random.normal(size=[sample_size, mu.shape[0]])
```

```
    # update mu
    loss_mu = np.zeros(shape=[sample_size, mu.shape[0]])
    loss_logsigma2 = np.zeros(shape=[sample_size, sigma2.shape[0]])
    for i in range(sample_size):
        loss_mu[i] = get_gradient_mu_rt(x, y, mu, sigma2, sample_eps[i])
        loss_logsigma2[i] = get_gradient_logsigma2_rt(x, y, mu, sigma2, sample_eps[i])
    update_mu = np.mean(loss_mu, axis=0)
    update_logsigma2 = np.mean(loss_logsigma2, axis=0)

    var_loss_mu = np.var(loss_mu, axis=0)
    var_loss_logsigma2 = np.var(loss_logsigma2, axis=0)

    grad = np.concatenate([update_mu, update_logsigma2])
    m = 0.9 * m + 0.1 * grad
    v = 0.999 * v + 0.001 * np.power(grad, 2)

    m_hat = m / (1 - np.power(0.9, n_iter))
    v_hat = v / (1 - np.power(0.999, n_iter))

    update = m_hat / (np.sqrt(v_hat) + 1e-10)

    mu += lr * update[:2]
    sigma2 = np.exp(np.log(sigma2) + lr * update[2])
    #print(mu, sigma2)
    return mu, sigma2, m, v, var_loss_mu, var_loss_logsigma2
```

```
def train_bbvi_rt(x, y, n_iter):
    mu_list = []
    sigma2_list = []
    var_list = []
    mu = np.random.normal(size=2)
    sigma2 = np.power(np.random.normal(size=1), 2)
    lr = 0.1
    m = np.zeros(shape=3)
    v = np.zeros(shape=3)
    for i in range(n_iter):
        mu, sigma2, m, v, var_mu, var_sigma = bbvi_rt(x, y, mu, sigma2, lr, i + 1, m,
v)
        mu_list.append(mu.copy())
        sigma2_list.append(sigma2.copy())
        var_list.append([var_mu.copy(), var_sigma.copy()])
    return mu_list, sigma2_list, var_list
```

```
def get_batch(X, Y, batch_size):
    n = X.shape[0]
    start_ele = np.random.randint(0, n)
    if start_ele + batch_size >= n:
        X_batch = np.concatenate((X[start_ele: ], X[ :start_ele + batch_size - n]))
        Y_batch = np.concatenate((Y[start_ele: ], Y[ :start_ele + batch_size - n]))
    else:
        X_batch = X[start_ele: (start_ele + batch_size)]
        Y_batch = Y[start_ele: (start_ele + batch_size)]
    return X_batch, Y_batch
```

```
def train_bbvi_batch(x, y, n_iter, batch_size):
    mu_list = []
    sigma2_list = []
    var_list = []
    mu = np.random.normal(size=2)
    sigma2 = np.power(np.random.normal(size=1), 2)
    lr = 0.1
    m = np.zeros(shape=3)
    v = np.zeros(shape=3)
    for i in range(n_iter):
        x_b, y_b = get_batch(x, y, batch_size)
        mu, sigma2, m, v = bbvi(x, y, mu, sigma2, lr, i + 1, m, v)
        mu_list.append(mu.copy())
        sigma2_list.append(sigma2.copy())
    return mu_list, sigma2_list
```

```
def train_bbvi_rt_batch(x, y, n_iter, batch_size):
    mu_list = []
    sigma2_list = []
    var_list = []
    mu = np.random.normal(size=2)
    sigma2 = np.power(np.random.normal(size=1), 2)
    lr = 0.1
    m = np.zeros(shape=3)
    v = np.zeros(shape=3)
    for i in range(n_iter):
        x_b, y_b = get_batch(x, y, batch_size)
        mu, sigma2, m, v, var_mu, var_sigma = bbvi_rt(x, y, mu, sigma2, lr, i + 1, m,
v)
        mu_list.append(mu.copy())
        sigma2_list.append(sigma2.copy())
        var_list.append([var_mu.copy(), var_sigma.copy()])
    return mu_list, sigma2_list, var_list
```

```
def train_bbvi_cv_batch(x, y, n_iter, batch_size):
    mu_list = []
    sigma2_list = []
    mu = np.random.normal(size=2)
    sigma2 = np.power(np.random.normal(size=1), 2)
    lr = 0.1
```

```
    m = np.zeros(shape=3)
    v = np.zeros(shape=3)
    for i in range(n_iter):
        x_b, y_b = get_batch(x, y, batch_size)
        mu, sigma2, m, v = bbvi_cv(x, y, mu, sigma2, lr, i + 1, m, v)
        mu_list.append(mu.copy())
        sigma2_list.append(sigma2.copy())
    return mu_list, sigma2_list
```

```
mu_bbvi, sigma2_bbvi, var_bbvi = train_bbvi(x, y, 1000)
```

```
mu_bbvi_cv, sigma2_bbvi_cv, var_bbvi_cv = train_bbvi_cv(x, y, 1000)
```

```
mu_bbvi_rt, sigma2_bbvi_rt, var_bbvi_rt = train_bbvi_rt(x, y, 1000)
```