Student: Quoc Tuan Vinh, Ngo
ID: 704526

CS-E4820 – Machine Learning: Advanced Probabilistic Methods
Homework Assignment 9

Problem 1. *"Stochastic ELBO gradient for the simple model (1/2)."*

(a) Derive equation 7

From *stochastic_gradient_elbo_search*, we know the following:

$$\log p(\boldsymbol{x}, \boldsymbol{z}, \tau, \theta) = \log p(\tau) + \log p(\theta) + \log p(\boldsymbol{z}|\tau) + \log p(\boldsymbol{x}|\boldsymbol{z}, \theta)$$

We also know the mean-field approximation with the distributions of factors $q(z_n|r_1, r_2), q(\tau), q(\theta)$, and that λ represents all the variational parameters. Therefore, one can easily see that each factor is only conditional on its own parameters, such as $q(\tau|\lambda) = q(\tau|\alpha_\tau, \beta_\tau)$, $\log q(\theta|\lambda) = q(\theta|m_2, \beta_2^{-1})$.

The derivation can be done as follow:

$$\begin{aligned}
&\nabla_\lambda \log q(z|\lambda)\left(\log p(x, z) - \ log q(z|\lambda)\right) \\
&\quad = \nabla_\lambda \log q(\boldsymbol{z}, \tau, \theta|\lambda) \times (\log p(\boldsymbol{x}, \boldsymbol{z}, \tau, \theta) - \log q(\boldsymbol{z}, \tau, \theta|\lambda)) \\
&\quad = \nabla_\lambda \left(\log q(\tau|\lambda) + \log q(\theta|\lambda) + \log \prod_{n=1}^{N} q(z_n|\lambda_n) \right) \\
&\quad \times \Bigg(\log p(\tau) + \log p(\theta) + \log p(\boldsymbol{z}|\tau) + \log p(\boldsymbol{x}|\boldsymbol{z}, \theta) - \log q(\tau|\lambda) \\
&\quad - \log q(\theta|\lambda) - \log \prod_{n=1}^{N} q(z_n|\lambda_n) \Bigg) \\
&\quad = \nabla_\lambda \left(\log q(\tau|\alpha_\tau, \beta_\tau) + \log q(\theta|m_2, \beta_2^{-1}) + \sum_{n=1}^{N} \log(z_n|r_n) \right) \\
&\quad \times \Bigg[\log p(\tau) + \log p(\theta) + \sum_{n=1}^{N} \log\,(z_n|\tau) + \sum_{n=1}^{N} \log p(x_n|z_n, \theta) \\
&\quad - \log q(\tau|\alpha_\tau, \beta_\tau) - \log q(\theta|m_2, \beta_2^{-1}) - \sum_{n=1}^{N} \log(z_n|r_n) \Bigg]
\end{aligned}$$

Which is what we need to prove.

(b) Implementation of function sample_from_q

For *tau_s* and *theta_s*, it's transparent to use built-in random generation for its o wn distribution (normal and beta). For *z_s*, since it is basically a random choice b etween 0 and 1, which resembles Bernoulli distribution. Here, I grant the value t o $z_{n1} = 1$ if the generated value is higher than r_{n1}. z_{n2} is the value other than z_{n1}.

```
def sample_from_q(alpha_tau, beta_tau, r1, r2, m2, beta2):
        tau_s = np.random.beta(alpha_tau,beta_tau)    # EXERCISE
```

```
theta_s = np.random.normal(m2,1/beta2) > r1 # EXERCISE
z1 = np.random.binomial(1,r1,len(r1)) # EXERCISE
z2 = 1 - z1  # EXERCISE

# z_s contains z1 and z2 as its columns
z_s = np.array([z1, z2]).T

return tau_s, theta_s, z_s
```

(c) Implementation of log joint of the approximate distribution

$$q(\tau|\alpha_\tau,\beta_\tau) = \log\Gamma(\alpha_\tau+\beta_\tau) - \log\Gamma(\alpha_\tau) - \log\Gamma(\beta_\tau) + (\alpha_\tau - 1)\log\tau + (\beta_\tau - 1)\log(1-\tau)$$

$$q(\theta|m_2,\beta_2^{-1}) = -\frac{1}{2}\log 2\pi + \frac{1}{2}\log\beta_2 - \frac{1}{2}\beta_2(\theta - m_2)^2$$

$$\log q(z_n|r_{n1},r_{n2}) = \log(z_{n1}\log r_{n1} + z_{n2}\log r_{n2})$$

The implementation is as below:

```
# COMPUTE LOG JOINT OF APPROXIMATION Q:
# log[q(tau)] + log[q(theta)] + log[q(z)]

log_q_tau = gammaln(alpha_tau + beta_tau) - gammaln(alpha_tau) - gammaln(beta_tau
) + (alpha_tau - 1) * np.log(tau_s) + (beta_tau - 1) * np.log(1-tau_s)    # EXERCISE, note:
the gammaln function has been imported from scipy.special

log_q_theta = 0.5 * np.log(beta2) - 0.5 * np.log(2*np.pi) - 0.5 * beta2 * (theta_s - m2) ** 2
# EXERCISE

log_q_z = np.sum(z_s[:,0]*np.log(r1) + z_s[:,1]*np.log(r2)) # EXERCISE

log_joint_q = log_q_tau + log_q_theta + log_q_z
```

Problem 2. *"Stochastic ELBO gradient for the simple model (2/2)."*

(a) Compute partial derivatives required for the stochastic gradient of ELBO

$$f(\alpha_\tau,\beta_\tau,m_2,\beta_2^{-1},\boldsymbol{r}) \triangleq \log q(\tau|\alpha_\tau,\beta_\tau) + \log q(\theta|m_2,\beta_2^{-1}) + \sum_{n=1}^{N}\log(z_n|r_n)$$

(2a.1) Derivation wrt to α_τ:

$$\frac{\delta f}{\delta\alpha_\tau} = \frac{\delta}{\delta\alpha_\tau}\log q(\tau|\alpha_\tau,\beta_\tau)$$
$$= \frac{\delta}{\delta\alpha_\tau}[\log\Gamma(\alpha_\tau+\beta_\tau) - \log\Gamma(\alpha_\tau) - \log\Gamma(\beta_\tau) + (\alpha_\tau - 1)\log\tau + (\beta_\tau - 1)\log(1-\tau)] = \psi(\alpha_\tau+\beta_\tau) - \psi(\alpha_\tau) + \log\tau$$

(2a.2) Derivation wrt to β_τ:

$$\frac{\delta f}{\delta \beta_\tau} = \frac{\delta}{\delta \beta_\tau} \log q(\tau | \alpha_\tau, \beta_\tau)$$

$$= \frac{\delta}{\delta \beta_\tau} [\log \Gamma(\alpha_\tau + \beta_\tau) - \log \Gamma(\alpha_\tau) - \log \Gamma(\beta_\tau) + (\alpha_\tau - 1) \log \tau$$
$$+ (\beta_\tau - 1) \log(1 - \tau)] = \psi(\alpha_\tau + \beta_\tau) - \psi(\beta_\tau) + \log(1 - \tau)$$

(2a.3) Derivation wrt to m_2:

$$\frac{\delta f}{\delta m_2} = \frac{\delta}{\delta m_2} \log q(\theta | m_2, \beta_2^{-1}) = \frac{\delta}{\delta m_2}\left[-\frac{1}{2}\log 2\pi + \frac{1}{2}\log \beta_2 - \frac{1}{2}\beta_2(\theta - m_2)^2\right]$$
$$= \beta_2(\theta - m_2)$$

(2a.4) Derivation wrt to β_2:

$$\frac{\delta f}{\delta \beta_2} = \frac{\delta}{\delta \beta_2} \log q(\theta | m_2, \beta_2^{-1}) = \frac{\delta}{\delta \beta_2}\left[-\frac{1}{2}\log 2\pi + \frac{1}{2}\log \beta_2 - \frac{1}{2}\beta_2(\theta - m_2)^2\right]$$
$$= \frac{1}{2\beta_2} - \frac{1}{2}(\theta - m_2)^2$$

(2a.5) Derivation wrt to r_{n1}:

$$\frac{\delta f}{\delta r_{n1}} = \frac{\delta}{\delta r_{n1}} \sum_{n=1}^{N} \log q(z_n | r_{n1}, r_{n2}) = \frac{\delta}{\delta r_{n1}} \sum_{n=1}^{N} [\log(r_{n1}^{z_{n1}} r_{n2}^{z_{n2}})]$$
$$= \frac{\delta}{\delta r_{n1}} \sum_{n=1}^{N} [z_{n1} \log r_{n1} + z_{n2} \log(1 - r_{n1})] = \frac{z_{n1}}{r_{n1}} - \frac{z_{n2}}{1 - r_{n1}}$$

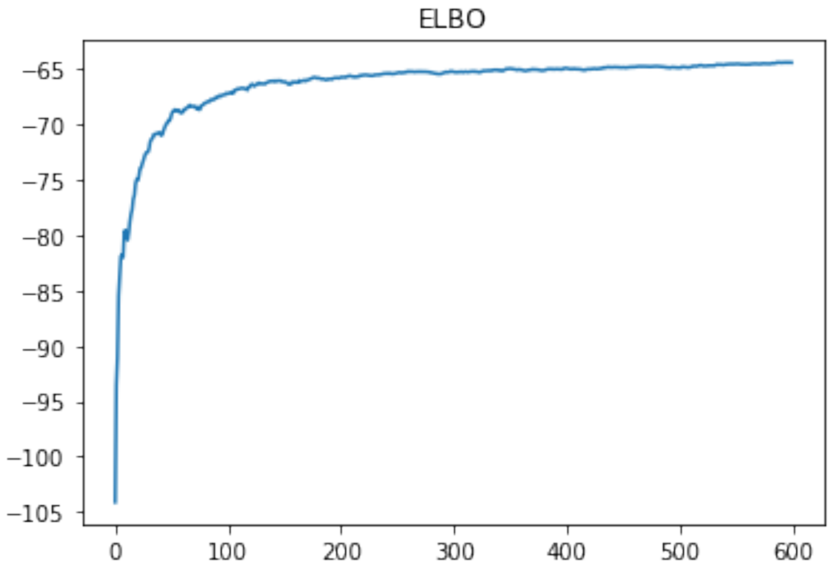
Implementation of the above-derived formulas is as below:

```
# COMPUTE GRADIENT of loq[q(tau,theta,z)] w.r.t. variational parameters

d_alpha_tau = psi(alpha_tau + beta_tau) - psi(alpha_tau) + np.log(tau_s) # EXERCISE, note: the psi function has been imported from scipy.special
d_beta_tau = psi(alpha_tau + beta_tau) - psi(beta_tau) + np.log(1-tau_s) # EXERCISE
d_m2 = beta2 * (theta_s - m2)          # EXERCISE
d_beta2 = 0.5/beta2 -0.5*(theta_s - m2)**2       # EXERCISE
d_rn1 = z_s[:,0]/r1 - z_s[:,1]/(1-r1)       # EXERCISE
```

(b) Run the completed code in ex9_12_template.py

Below is the result of the execution. The ELBO increases significantly from iteration 0th to ~200th, after that it slowly converges to the value of ~ - 65. The final value for theta and tau is `Iteration 599: theta=3.319918, tau=0.262478.` This is unfortunately not too close to the true value (theta = 4, tau = 0.3). Full code that is used to generate this result can see at the end of this question.




```
 Iteration 0: theta=0.977868, tau=0.530960
 Iteration 1: theta=1.233037, tau=0.450139
 Iteration 2: theta=1.321848, tau=0.408247
 Iteration 3: theta=1.603792, tau=0.390808
 Iteration 4: theta=1.702278, tau=0.396963
 Iteration 5: theta=1.836344, tau=0.412946
 Iteration 6: theta=1.822126, tau=0.398032
 Iteration 7: theta=1.760122, tau=0.363667
 Iteration 8: theta=1.910181, tau=0.348455
 Iteration 9: theta=1.918215, tau=0.336495
Iteration 10: theta=1.876110, tau=0.353738
Iteration 11: theta=1.848098, tau=0.347813
Iteration 12: theta=1.899561, tau=0.354232
Iteration 13: theta=1.967408, tau=0.342817
Iteration 14: theta=1.983798, tau=0.337429
Iteration 15: theta=2.035042, tau=0.339331
Iteration 16: theta=2.120342, tau=0.340452
Iteration 17: theta=2.148343, tau=0.343072
Iteration 18: theta=2.191032, tau=0.319568
Iteration 19: theta=2.231852, tau=0.327572
Iteration 20: theta=2.217532, tau=0.318117
Iteration 21: theta=2.262509, tau=0.307194
Iteration 22: theta=2.337510, tau=0.306125
Iteration 23: theta=2.350161, tau=0.296128
Iteration 24: theta=2.370712, tau=0.315448
Iteration 25: theta=2.398326, tau=0.309914
Iteration 26: theta=2.434685, tau=0.304214
Iteration 27: theta=2.461478, tau=0.291401
Iteration 28: theta=2.439413, tau=0.298040
Iteration 29: theta=2.452357, tau=0.299297
Iteration 30: theta=2.498559, tau=0.302159
Iteration 31: theta=2.551941, tau=0.298527
Iteration 32: theta=2.563591, tau=0.298890
Iteration 33: theta=2.573792, tau=0.298788
Iteration 34: theta=2.612104, tau=0.292530
Iteration 35: theta=2.639045, tau=0.288367
Iteration 36: theta=2.655380, tau=0.303048
Iteration 37: theta=2.655907, tau=0.297569
Iteration 38: theta=2.650960, tau=0.303120
Iteration 39: theta=2.647362, tau=0.307589
Iteration 40: theta=2.645685, tau=0.306133
Iteration 41: theta=2.622158, tau=0.312061
Iteration 42: theta=2.643151, tau=0.312517
Iteration 43: theta=2.684511, tau=0.307002
Iteration 44: theta=2.707888, tau=0.303331
Iteration 45: theta=2.736556, tau=0.310763
Iteration 46: theta=2.728877, tau=0.315550
Iteration 47: theta=2.758234, tau=0.304489
Iteration 48: theta=2.773878, tau=0.317574
Iteration 49: theta=2.778805, tau=0.316326
Iteration 50: theta=2.814372, tau=0.306838
```

....

```
Iteration 550: theta=3.334458, tau=0.263525
Iteration 551: theta=3.317234, tau=0.263917
Iteration 552: theta=3.321060, tau=0.262540
Iteration 553: theta=3.332272, tau=0.264686
Iteration 554: theta=3.335685, tau=0.264581
```

```
Iteration 555: theta=3.340009, tau=0.263418
Iteration 556: theta=3.341749, tau=0.265462
Iteration 557: theta=3.349585, tau=0.264473
Iteration 558: theta=3.356117, tau=0.264910
Iteration 559: theta=3.348974, tau=0.261948
Iteration 560: theta=3.338452, tau=0.260439
Iteration 561: theta=3.351105, tau=0.260796
Iteration 562: theta=3.350027, tau=0.261194
Iteration 563: theta=3.357351, tau=0.261794
Iteration 564: theta=3.360327, tau=0.259780
Iteration 565: theta=3.357407, tau=0.258198
Iteration 566: theta=3.351287, tau=0.258894
Iteration 567: theta=3.348427, tau=0.257552
Iteration 568: theta=3.355717, tau=0.258256
Iteration 569: theta=3.350980, tau=0.258932
Iteration 570: theta=3.356107, tau=0.258829
Iteration 571: theta=3.355614, tau=0.257984
Iteration 572: theta=3.362379, tau=0.257004
Iteration 573: theta=3.374386, tau=0.258494
Iteration 574: theta=3.365022, tau=0.259744
Iteration 575: theta=3.365519, tau=0.259608
Iteration 576: theta=3.363830, tau=0.256636
Iteration 577: theta=3.352589, tau=0.257016
Iteration 578: theta=3.347581, tau=0.257965
Iteration 579: theta=3.345725, tau=0.256204
Iteration 580: theta=3.341612, tau=0.257579
Iteration 581: theta=3.343490, tau=0.257844
Iteration 582: theta=3.341807, tau=0.259464
Iteration 583: theta=3.339630, tau=0.260976
Iteration 584: theta=3.330052, tau=0.258542
Iteration 585: theta=3.333116, tau=0.259222
Iteration 586: theta=3.351917, tau=0.259443
Iteration 587: theta=3.355222, tau=0.258929
Iteration 588: theta=3.354607, tau=0.260877
Iteration 589: theta=3.351903, tau=0.260782
Iteration 590: theta=3.345151, tau=0.260528
Iteration 591: theta=3.345620, tau=0.261664
Iteration 592: theta=3.345996, tau=0.262322
Iteration 593: theta=3.334131, tau=0.263130
Iteration 594: theta=3.327413, tau=0.264189
Iteration 595: theta=3.322811, tau=0.264642
Iteration 596: theta=3.328095, tau=0.263513
Iteration 597: theta=3.323350, tau=0.261607
Iteration 598: theta=3.328844, tau=0.263678
Iteration 599: theta=3.319918, tau=0.262478
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import gammaln, psi

np.random.seed(123123120)


def compute_stochastic_elbo_gradient(alpha_tau, beta_tau, r1, r2, m2, beta2, alpha0, beta0, x, n_simulations):
    n_data_items = len(r1)

    # Derivatives w.r.t. alpha_tau, beta_tau, m2, beta2, and all r1 terms.
    elbo_grad_array = np.zeros((n_simulations, 4 + n_data_items))


    for simu_index in range(n_simulations):
        # Estimate the gradient by sampling (i.e. simulating) from the current
        # approximation n_simulations many times, and average in the end.


        # SAMPLE unobservables from the current approximation
        tau_s, theta_s, z_s = sample_from_q(alpha_tau, beta_tau, r1, r2, m2, beta2)

        # COMPUTE MODEL LOG JOINT:
        # log[p(tau)] + log[p(theta)] + log[p(z|tau)] + log[p(x|z,theta)]

        # log[Beta(tau|alpha0,alpha0)]
        log_p_tau = gammaln(2 * alpha0) - 2 * gammaln(alpha0) + (alpha0 - 1) * np.log(tau_s) + (alpha0 - 1) *
np.log(1-tau_s)

        # log[N(theta|0,beta0^(-1))
        log_p_theta = 0.5 * np.log(beta0) - 0.5 * np.log(2*np.pi) - 0.5 * beta0 * theta_s ** 2
```

```
                # log[p(z|tau)]
                N1 = np.sum(z_s[:,0])
                N2 = np.sum(z_s[:,1])
                log_p_z_cond_tau = N1 * np.log(1-tau_s) + N2 * np.log(tau_s)

                # log[p(x|z,theta)]
                N = N1 + N2
                log_p_x_cond_z_theta = -0.5 * N * np.log(2*np.pi) - 0.5 * np.sum(z_s[:,0] * x ** 2) - 0.5 *
np.sum(z_s[:,1] * (x - theta_s) ** 2)

                log_joint_p = log_p_tau + log_p_theta + log_p_z_cond_tau + log_p_x_cond_z_theta


                # COMPUTE LOG JOINT OF APPROXIMATION Q:
                # log[q(tau)] + log[q(theta)] + log[q(z)]

                log_q_tau = gammaln(alpha_tau + beta_tau) - gammaln(alpha_tau) - gammaln(beta_tau) + (alpha_tau -
1) * np.log(tau_s) + (beta_tau - 1) * np.log(1-tau_s)    # EXERCISE, note: the gammaln function has been imported from
scipy.special

                log_q_theta = 0.5 * np.log(beta2) - 0.5 * np.log(2*np.pi) - 0.5 * beta2 * (theta_s - m2) ** 2   #
EXERCISE

                log_q_z = np.sum(z_s[:,0]*np.log(r1) + z_s[:,1]*np.log(r2)) # EXERCISE

                log_joint_q = log_q_tau + log_q_theta + log_q_z


                # COMPUTE GRADIENT of loq[q(tau,theta,z)] w.r.t. variational parameters

                d_alpha_tau = psi(alpha_tau + beta_tau) - psi(alpha_tau) + np.log(tau_s)   # EXERCISE, note: the psi
function has been imported from scipy.special

                d_beta_tau = psi(alpha_tau + beta_tau) - psi(beta_tau) + np.log(1-tau_s)    # EXERCISE

                d_m2 = beta2 * (theta_s - m2)          # EXERCISE

                d_beta2 = 0.5/beta2 -0.5*(theta_s - m2)**2       # EXERCISE

                d_rn1 =  z_s[:,0]/r1 - z_s[:,1]/(1-r1)       # EXERCISE

                # COMBINE EVERYTHING to form the gradient of the ELBO:
                elbo_grad_array[simu_index, :] = np.concatenate([[d_alpha_tau, d_beta_tau, d_m2, d_beta2], d_rn1]) *
(log_joint_p - log_joint_q)


        # AVERAGE over the samples:
        elbo_grad = np.mean(elbo_grad_array, axis=0)

        return elbo_grad


def sample_from_q(alpha_tau, beta_tau, r1, r2, m2, beta2):
        tau_s = np.random.beta(alpha_tau,beta_tau)    # EXERCISE
        theta_s = np.random.normal(m2,1/beta2)  # EXERCISE
        z1 = np.random.binomial(1,r1,len(r1))  > r1 # EXERCISE
        z2 = 1 - z1  # EXERCISE

        # z_s contains z1 and z2 as its columns
        z_s = np.array([z1, z2]).T

        return tau_s, theta_s, z_s


# Compute ELBO for the model described in simple_elbo.pdf
def compute_elbo(alpha_tau, beta_tau, r1, r2, m2, beta2, alpha0, beta0, x):

  # E[log p(tau)]
  term1 = (alpha0 - 1) * (psi(alpha_tau) + psi(beta_tau) - 2 * psi(alpha_tau + beta_tau))

  # E[log p(theta)]
  term2 = -0.5 * beta0 * (beta2**(-1) + m2**2)
```

```
    # E[log p(z|tau)]
    N2 = np.sum(r2); N1 = np.sum(r1); N = N1 + N2
    term3 = N2 * psi(alpha_tau) + N1 * psi(beta_tau) - N * psi(alpha_tau + beta_tau)

    # E[log p(x|z,theta)]
    term4 = -0.5 * np.sum(r1 * x**2) - 0.5 * np.sum(r2 * ((x-m2)**2 + beta2**(-1)))

    # Negative entropy of q(z)
    term5 = np.sum(r1 * np.log(r1)) + np.sum(r2 * np.log(r2))

    # Negative entropy of q(tau)
    term6 = (gammaln(alpha_tau + beta_tau) - gammaln(alpha_tau) - gammaln(beta_tau)
        + (alpha_tau - 1) * psi(alpha_tau) + (beta_tau - 1) * psi(beta_tau)
        - (alpha_tau + beta_tau - 2) * psi(alpha_tau + beta_tau))

    # Negative entropy of q(theta)
    term7 = 0.5 * np.log(beta2)

    elbo = term1 + term2 + term3 + term4 - term5 - term6 - term7

    return elbo

# Simulate data
theta_true = 4
tau_true = 0.3
n_samples = 50
z = (np.random.rand(n_samples) < tau_true)  # True with probability tau_true
x = np.random.randn(n_samples) + z * theta_true


# Parameters of the prior distributions.
alpha0 = 1.5
beta0 = 1

n_iter = 600
# To keep track of the estimates of tau and theta in different iterations:
tau_est = np.zeros(n_iter)
th_est = np.zeros(n_iter)
elbo_array = np.zeros(n_iter)   # To track the elbo


# Some initial values for the variational parameters
alpha_tau = 1
beta_tau = 1
beta_2 = 1
m2 = 1
r1 = np.random.rand(n_samples)   # Responsibilities of the first cluster.
r2 = 1 - r1

for it in range(n_iter):
        step_size = 0.02 / (10+it)**0.5

        # Compute the gradient of the ELBO
        elbo_grad = compute_stochastic_elbo_gradient(alpha_tau, beta_tau, r1, r2, m2, beta_2, alpha0, beta0, x, 200)

        # Update factor q(tau) using stochastic gradient
        alpha_tau = np.max([alpha_tau + step_size * elbo_grad[0], 0.1])
        beta_tau  = np.max([beta_tau  + step_size * elbo_grad[1], 0.1])

        # Update factor q(theta) using stochastic gradient
        m2 = m2 + step_size * elbo_grad[2]
        beta_2 = beta_2 + step_size * elbo_grad[3]

        # Update responsibilites, factor q(z), using closed-form updates
        E_log_tau = psi(alpha_tau) - psi(alpha_tau + beta_tau)
        E_log_tau_c = psi(beta_tau) - psi(alpha_tau + beta_tau)
        E_log_var = (x-m2)**2 + 1/beta_2

        log_rho1 = E_log_tau_c - 0.5 * np.log(2*np.pi) - 0.5 * (x**2)
        log_rho2 = E_log_tau - 0.5 * np.log(2*np.pi) - 0.5 * E_log_var
        max_log_rho = np.maximum(log_rho1, log_rho2)   # Normalize to avoid numerical problems when
exponentiating.
```

```
        rho1 = np.exp(log_rho1 - max_log_rho)
        rho2 = np.exp(log_rho2 - max_log_rho)
        r2 = rho2 / (rho1 + rho2)
        r1 = 1 - r2

        # Keep track of the current estimates
        tau_est[it] = (alpha_tau) / (alpha_tau + beta_tau)
        th_est[it] = m2

        # Compute the ELBO
        elbo_array[it] = compute_elbo(alpha_tau, beta_tau, r1, r2, m2, beta_2, alpha0, beta0, x)

        print("Iteration %i: theta=%f, tau=%f" % (it, m2, tau_est[it]))
        # With large enough n_samples, this should eventually converge
        # to (theta_true, tau_true), at least approximately.


plt.plot(elbo_array)
plt.title('ELBO')
plt.show()
```

Problem 3. *"SVI in Edward"*

(a) See below next page.
(b) Investigate the impact of mini-batch size on convergence speed

We can create a loop to test elapsed time with different mini-batch size. For instance, for 5 different minibatches from 100-500 observations (step size 100), elapsed time does not seem affected very much. It even seems that higher mini-batch size results in quicker convergence.

```
500/500 [100%] ██████████████████████████████ Elapsed: 6s | Loss: 10197
4.953
250/250 [100%] ██████████████████████████████ Elapsed: 4s | Loss: 54505
.922
165/165 [100%] ██████████████████████████████ Elapsed: 3s | Loss: 41153
.016
125/125 [100%] ██████████████████████████████ Elapsed: 2s | Loss: 37041
.762
100/100 [100%] ██████████████████████████████ Elapsed: 4s | Loss: 29400
.410
```

```
#Test batch size
M_list = np.arange(100,500,100)
print(M_list)
for M in M_list:
    n_batch = int(N / M)
    n_epoch = 5

    inference = ed.KLqp({w: qw, b: qb}, data={y: y_ph})
    inference.initialize(n_iter=n_batch * n_epoch, n_samples=5, scale={y: N / M})
    tf.global_variables_initializer().run()

    for _ in range(inference.n_iter):
      X_batch, y_batch = next(data)
      info_dict = inference.update({X: X_batch, y_ph: y_batch})
      inference.print_progress(info_dict)
```

Ex09.Problem3.1.

March 27, 2019

1 Comparison between SVI and LRVI

The code below is gotten from SVI, which is used to reflect the difference between two models. The difference in code is highlighted in Yellow.

1.1 Model

Posit the model as Bayesian linear regression (Murphy, 2012). For a set of N data points $(\mathbf{X}, \mathbf{y}) = \{(\mathbf{x}_n, y_n)\}$, the model posits the following distributions:

$$p(\mathbf{w}) = \text{Normal}(\mathbf{w} \mid \mathbf{0}, \sigma_w^2 \mathbf{I}),$$

$$p(b) = \text{Normal}(b \mid 0, \sigma_b^2),$$

$$p(\mathbf{y} \mid \mathbf{w}, b, \mathbf{X}) = \prod_{n=1}^{N} \text{Normal}(y_n \mid \mathbf{x}_n^\top \mathbf{w} + b, \sigma_y^2).$$

The latent variables are the linear model's weights $\mathbf{w}$ and intercept b, also known as the bias. Assume σ_w^2, σ_b^2 are known prior variances and σ_y^2 is a known likelihood variance. The mean of the likelihood is given by a linear transformation of the inputs $\mathbf{x}_n$.

Let's build the model in Edward, fixing $\sigma_w, \sigma_b, \sigma_y = 1$.

```
In [ ]: X = tf.placeholder(tf.float32, [None, D])
        y_ph = tf.placeholder(tf.float32, [None])

        w = Normal(loc=tf.zeros(D), scale=tf.ones(D))
        b = Normal(loc=tf.zeros(1), scale=tf.ones(1))
        y = Normal(loc=ed.dot(X, w) + b, scale=1.0)
```

>> Comparison between SVI and LRVI:

In general, both models define X first.(which is a place holder for passing the value later). Then it defines w, b, y, which are exactly the same.

The main difference lies at the number of training size: In SVI, number of rows for X and y is not fixed, because we need to enable training with batches of varying size. Therefore, a place holder for y (*y_ph*) is also defined with unfixed size. Meanwhile, the number of training size is fixed in LRVI, so the number of rows is pre-defined as N, and there is no need to create a place holder for y.

1.2 Inference

```
In [ ]: qw = Normal(loc=tf.get_variable("qw/loc", [D]),
                    scale=tf.nn.softplus(tf.get_variable("qw/scale", [D])))
        qb = Normal(loc=tf.get_variable("qb/loc", [1]),
                    scale=tf.nn.softplus(tf.get_variable("qb/scale", [1])))
```

```
In [ ]: n_batch = int(N / M)
        n_epoch = 5

        inference = ed.KLqp({w: qw, b: qb}, data={y: y_ph})
        inference.initialize(n_iter=n_batch * n_epoch, n_samples=5, scale={y: N / M})
        tf.global_variables_initializer().run()

        for _ in range(inference.n_iter):
          X_batch, y_batch = next(data)
          info_dict = inference.update({X: X_batch, y_ph: y_batch})
          inference.print_progress(info_dict)
```

>> Comparison between SVI and LRVI:

Firstly, both models need to define variational model to be a fully factorized normal across the weights with exact similar codes.

In the second code block:

- In LRVI: the KL_qp implementation is simple with just two lines: Define KLqp and then run. There is no need to define training set size as it is fixed to N.
- In SVI:

The number of batch is defined according to batch size and size of total training set (N/M); and the number of epoch is also defined. One can see that M is much smaller than N. Therefore, it is more complicated in the sense that after feeding the number of batches to the placeholder, we need to initialize the infernece first. This involves the scaling of y by N/M to get the correct expectation (Statistically, this avoids inference being dominated by the prior).

Then, we need to construct the loop in order to execute the update of batch training data for X_batch and y_batch.

Since we run SVI by batch, the reported Loss as we run corresponds to the computed objective given the current batch and not the total data set.