

Assignment 1AD, 2025

DD2434 Machine Learning, Advanced Course

Bruno Carchia
carchia@kth.se

December 10, 2025

D Section

D.1

Question 1.1.1

Starting from the definition of the KL divergence $\text{KL}(q(Z)||p(Z|X))$, we want to rewrite it and identify the quantity referred as the Evidence Lower Bound (ELBO).

$$\begin{aligned}\text{KL}(q(z)||p(z|x)) &= E_q \left[\log \frac{q(z)}{p(z|x)} \right] \\ &= \int q(z) \log \frac{q(z)}{p(z|x)} dz \\ &= \int q(z) \log \frac{q(z)}{\frac{p(z,x)}{p(x)}} dz \\ &= \int q(z) \log \frac{q(z)}{p(z,x)} dz + \int q(z) \log p(x) dz \\ &= E_q \left[\log \frac{q(z)}{p(z,x)} \right] + \log p(x)\end{aligned}$$

Knowing that the ELBO is defined as $\mathcal{L} = \text{KL}(p(x, z)|q(z))$

$$\text{KL}(q(z)||p(z|x)) = -\mathcal{L} + \log p(x)$$

$$\log p(x) = \mathcal{L} + \text{KL}(q(z)||p(z|x))$$

The ELBO is a lower bound because the KL divergence is always non-negative. Maximizing the ELBO effectively maximizes the lower bound on $\log p(x)$ and minimizes the KL divergence.

Question 1.1.2

- A more expressive variational family can better approximate the true posterior, reducing $\text{KL}(q(z)||p(z|x))$ and increasing the ELBO (a fully factorized mean field distribution produces a looser ELBO because it ignores dependencies between latent variables)
- A more expressive variational family can capture closely dependencies between latent variables, yielding $q(z)$ closer to the true posterior $p(z|x)$ (a fully factorized mean field distribution produces a less accurate posterior approximation)

D.2

Question 1.1.3

Considering a mean field assumption on our variation distribution $q(Z_1, Z_2, Z_3) = q(Z_1)q(Z_2)q(Z_3)$, we want to prove that $\log q_1^*(Z_1) = E_{-Z_1} [\log p(X, Z)]$

$$\begin{aligned}\mathcal{L} &= \int \prod_i^3 q(z_i) \log \left(\frac{p(x, z)}{\prod_l^3 q(z_l)} \right) dz \\ &= \int \prod_i^3 q(z_i) \left[\log(p(x, z)) - \sum_l^3 \log q(z_l) \right] dz \\ &= \int \prod_i^3 q(z_i) \log p(x, z) dz - \int \prod_i^3 q(z_i) \sum_l^3 \log q(z_l) dz\end{aligned}$$

We work individually on these two terms:

- First term can be decomposed in an outer component that depends on z_j and an inner component that depends on all the other variables z_{-j}

$$\int \prod_i^3 q(z_i) \log p(x, z) dz = \int_{z_j} q(z_j) \left[\int_{z_{-j}} \prod_{i \neq j}^3 q(z_i) \log p(x, z) dz_{-j} \right] dz_j$$

We define $\log \tilde{p}(x, z_j) = E_{\prod_{i \neq j} q(z_i)} [\log p(x, z)]$, the first term becomes

$$\int_{z_j} q(z_j) \log \tilde{p}(x, z_j) dz_j$$

- Second term can be rewritten in the following way

$$- \int \prod_i^3 q(z_i) \sum_l^3 \log q(z_l) dz = - \int_z \sum_l^3 \log q(z_l) \prod_i^3 q(z_i) dz$$

After switching the sum and the integral, we can decompose the term in an outer (depends on all the variables excepting for z_l) and inner component (depends just on z_l)

$$\begin{aligned} &= - \sum_l^3 \int_z \log q(z_l) \prod_i^3 q(z_i) dz \\ &= - \sum_l^3 \int_{z_{-l}} \prod_{i \neq l} q(z_i) \left[\int_{z_l} q(z_l) \log q(z_l) dz_l \right] dz_{-l} \end{aligned}$$

The inner component does not depend on terms related to z_{-l} , the external integral gives 1 and we got as final result

$$= - \sum_l^3 \int_{z_l} q(z_l) \log q(z_l) dz_l$$

The ELBO is

$$\mathcal{L} = \int_{z_j} q(z_j) \log \tilde{p}(x, z_j) dz_j - \sum_l^3 \int_{z_l} q(z_l) \log q(z_l) dz_l$$

But our objective is to maximize the ELBO by iteratively optimizing a single variational factor $q(z_j)$ while holding all other factors $q(z_{-j})$ constant. In other words we treat as constant all the terms that do not depend on z_j

$$\begin{aligned} \mathcal{L} &= \int_{z_j} q(z_j) \log \tilde{p}(x, z_j) dz_j - \int_{z_j} q(z_j) \log q(z_j) dz_j + \text{const} \\ &= E_{q(z_j)} \left[\frac{\log \tilde{p}(x, z_j)}{q(z_j)} \right] \\ &= -\text{KL}(\tilde{p}(x, z_j) || q(z_j)) \end{aligned}$$

Maximizimng the ELBO wrt a single variational factor $q(z_j)$ is equivalent to minimizing $\text{KL}(\tilde{p}(x, z_j) || q(z_j))$: the minimum occurs when $q(z_j)^* = \tilde{p}(x, z_j)$

$$\log \tilde{p}(x, z_j) = \log q(z_j)^* = E_{\prod_{l \neq j} q(z_l)} [\log p(x, z)]$$

In this case $j = 1$

D.3

We will analyze the model with Normal-likelihood and NormalGamma prior of 1E.3. instead of using the Coordinate Ascent Variational Inference (CAVI) algorithm, we will employ Black-Box Variational Inference (BBVI). This BBVI will utilize the RE-INFORCE gradient estimator (in its basic, high-variance form) to infer the variational distributions $q(\mu)$ and $q(\tau)$ under the mean-field assumption $q(\mu, \tau) = q(\mu)q(\tau)$

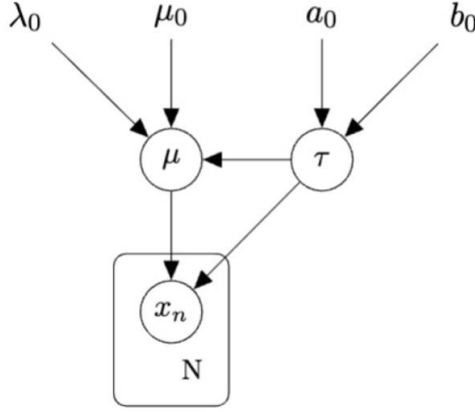


Figure 1: Bayesian network of the Normal-NormalGamma model

Question 1.2.4

We provide the final expressions for

- Log Likelihood $\log P(D|\mu, \tau)$

$$\begin{aligned}
 \log P(D|\mu, \tau) &= \log \left(\prod_{n=1}^N f_{\tau, \mu}(x_n) \right) \\
 &= \sum_{n=1}^N \log \left(\sqrt{\frac{\tau}{2\pi}} e^{-\frac{\tau}{2}(x_n - \mu)^2} \right) \\
 &= \sum_{n=1}^N [0.5 \log \tau - 0.5 \log 2\pi - 0.5\tau(x_n - \mu)^2]
 \end{aligned}$$

- Log Prior $\log P(\tau, \mu)$

$$\begin{aligned}
 \log P(\tau, \mu) &= \log P(\tau)P(\mu|\tau) \\
 &= \log \left[\frac{\beta_0^{\alpha_0}}{\Gamma(\alpha_0)} \sqrt{\frac{\lambda_0 \tau}{2\pi}} \tau^{\alpha_0-1} e^{-\beta_0 \tau} e^{-\frac{\lambda_0 \tau}{2}(\mu - \mu_0)^2} \right] \\
 &= \alpha_0 \log \beta_0 - \log \Gamma(\alpha_0) + 0.5 \log \lambda_0 \tau - 0.5 \log 2\pi + (\alpha_0 - 1) \log \tau - \beta_0 \tau - \frac{\lambda_0 \tau}{2}(\mu - \mu_0)^2
 \end{aligned}$$

- Log Variational Distribution $q(z[s]|\lambda)$

$$\log q(\mu, \tau|\mu_N, \lambda_N, \alpha_N, \beta_N) = \log q(\mu|\mu_N, \lambda_N) + \log q(\tau|\alpha_N, \beta_N)$$

Knowing that $q(\tau) \sim \text{Gamma}(\alpha_N, \beta_N)$ and $q(\mu) \sim \text{Normal}(\mu_N, \lambda_N^{-1})$

$$\left[0.5 \log \frac{\lambda_N}{2\pi} - \frac{\lambda_N(\mu - \mu_N)^2}{2} \right] + [\alpha_N \log \beta_N + (\alpha_N - 1) \log \tau - \log \Gamma(\alpha_N) - \beta_N \tau]$$

- Score function $\nabla_{\lambda} \log q(z[s]|\lambda)$

$$\nabla_{\mu_N, \lambda_N, \alpha_N, \beta_N} \log q(\mu, \tau | \mu_N, \lambda_N, \alpha_N, \beta_N) = \nabla_{\mu_N, \lambda_N} \log q(\mu | \mu_N, \lambda_N) + \nabla_{\alpha_N, \beta_N} \log q(\tau | \alpha_N, \beta_N)$$

We compute each term individually

$$\begin{aligned} & - \nabla_{\mu_N, \lambda_N} \log q(\mu | \mu_N, \lambda_N) \\ & \quad * \nabla_{\mu_N} \log q(\mu | \mu_N, \lambda_N) = \lambda_N (\mu - \mu_N) \\ & \quad * \nabla_{\lambda_N} \log q(\mu | \mu_N, \lambda_N) = +0.5 \frac{\frac{1}{2\pi}}{\lambda_N} - 0.5 (\mu - \mu_N)^2 = \frac{1}{2\lambda_N} - \frac{(\mu - \mu_N)^2}{2} \\ & - \nabla_{\alpha_N, \beta_N} \log q(\tau | \alpha_N, \beta_N) \\ & \quad * \nabla_{\alpha_N} \log q(\tau | \alpha_N, \beta_N) = \log \beta_N + \log \tau - \psi(\lambda_N), \text{ where } \psi(x) = \frac{d \ln \Gamma(x)}{dx} \\ & \quad * \nabla_{\beta_N} \log q(\tau | \alpha_N, \beta_N) = \frac{\alpha_N}{\beta_N} - \tau \end{aligned}$$

- $\log P(D, z[s]) = \log P(D, \mu_s, \tau_s) = \log P(D | \mu_s, \tau_s) + \log P(\mu_s, \tau_s)$

Question 1.2.5

We implemented algorithm 1 of the BBVI paper using Pytorch
Generation of dataset

Algorithm 1 Black Box Variational Inference

Input: data x , joint distribution p , mean field variational family q .
Initialize λ randomly, $t = 1$.
repeat
 // Draw S samples from q
 for $s = 1$ to S **do**
 $z[s] \sim q$
 end for
 $\rho = t$ th value of a Robbins Monro sequence
 $\lambda = \lambda + \rho \frac{1}{S} \sum_{s=1}^S \nabla_{\lambda} \log q(z[s] | \lambda) (\log p(x, z[s]) - \log q(z[s] | \lambda))$
 $t = t + 1$
until change of λ is less than 0.01.

```

1 def generate_data(mu, tau, N):
2     x = torch.linspace(-10, 10, N)
3     # Insert your code here
4     sigma = 1 / torch.sqrt(torch.tensor(tau))    # precision  $\tau = 1/\sigma^2$ 
5     torch.manual_seed(10)
6
7     D = torch.normal(mu, sigma, size=(N,))
8
9     return D
10
11 mu = 1
12 tau = 0.5
13
14 dataset = generate_data(mu, tau, 100)

```

Black Box Algorithm 1 implementation

```
1 class BlackBoxVI:
2     """Black Box Variational Inference implementation."""
3
4     def __init__(self, D, log_joint_distribution,
5         variational_family_q, S=10, learning_rate=1e-3):
6         """
7         Black Box Variational Inference implementation without any
8         .
9         Args:
10             D: dataset
11             log_joint_distribution: function that computes the
12                 log joint distribution
13             variational_family_q: variational family q with
14                 parameters to optimize
15             S: number of samples for Monte Carlo estimation
16             learning_rate: learning rate for the optimizer
17         """
18         self.D = D
19         self.log_joint_distribution = log_joint_distribution
20         self.variational_family_q = variational_family_q
21         self.S = S
22         self.learning_rate = learning_rate
23
24     def fit(self, max_iterations = 1000):
25         """Fit the variational parameters using BBVI algorithm.
26         SGD optimization
27         Args:
28             threshold: convergence threshold
29             max_iterations: maximum number of iterations
30         """
31         history = {
32             "elbo" : [],
33             "final_params" : None,
34             'mu_expected': [],
35             'tau_expected': [],
36         }
37         lr_sgd = self.learning_rate # A small learning rate
38         optimizer_sgd = optim.SGD([self.variational_family_q.
39             parameters], lr=lr_sgd)
40
41         for t in range(1, max_iterations+1):
42             elbo=0
43             loss=0
44             optimizer_sgd.zero_grad()
45             for s in range(self.S):
46                 z_s = self.variational_family_q.sample()
```

```

44         #compute log(q(z[s]))
45         log_q_z_s = self.variational_family_q.log_prob(z_s
46             )
47
48         # Compute log p(x, z[s])
49         log_p_z_s_D = self.log_joint_distribution(self.D,
50             z_s)
51
52         # Compute the score function  $\nabla_{\lambda} \log q(z[s]; \lambda)$ 
53         learning_signal = (log_p_z_s_D - log_q_z_s).detach
54             ()
55
56         #loss
57         elbo += learning_signal
58         loss += (- log_q_z_s * learning_signal)
59
60         elbo /= self.S
61         loss /= self.S
62
63         loss.backward()
64         optimizer_sgd.step()
65
66         #we compute the elb and we check the params every 10
67         iterations and at the first iteration
68         if t % 10 == 0 or t == 1:
69             mu_N, lambda_N, alpha_N, beta_N = self.
70                 variational_family_q.get_actual_parameters()
71             history["elbo"].append((t, elbo))
72             history['mu_expected'].append((t, mu_N.item()))
73             history['tau_expected'].append((t, (alpha_N /
74                 beta_N).item()))
75
76         history['final_params'] = self.variational_family_q.
77             get_parameters()
78
79         return history

```

The optimization loop performs these steps at each iteration:

- Sample latent variables \rightarrow Draw S samples $z[s]$ from the current variational distribution $q(z; \lambda)$
- Evaluate probabilities \rightarrow For each sample, we compute $\log q(z[s]; \lambda)$ and $\log p(D, z[s])$
- Compute learning signal \rightarrow Calculate $f[s] = \log p(D, z[s]) - \log q(z[s])$, which represents how much better sample $z[s]$ explains the data compared to the variational approximation

- Calculate gradient → Use the score function estimator
- Update parameters → Use SGD to update λ based on the averaged gradient

```

1
2 class NormalGammaVariationalFamily():
3
4     """Variational family for Normal-NormalGamma conjugate model.
5
6     Variational distribution:  $q(\mu, \tau | \lambda) = q(\mu | \tau) q(\tau)$ 
7     where:
8         -  $\mu | \tau \sim \text{Normal}(\mu_N, (\lambda_N * \tau)^{-1})$ 
9         -  $\tau \sim \text{Gamma}(\alpha_N, \beta_N)$ 
10
11     Parameters:  $\lambda = [\mu_N, \lambda_N, \alpha_N, \beta_N]$ 
12     """
13     def __init__(self):
14         """Initialize with dimension of latent variable."""
15         mu_N = torch.randn(1).item()
16         lambda_N = torch.rand(1).item() * 2 + 0.5
17         alpha_N = torch.rand(1).item() * 3 + 1.0
18         beta_N = torch.rand(1).item() * 3 + 0.5
19
20         self.parameters = torch.tensor([
21             mu_N, torch.log(torch.tensor(lambda_N)), torch.log(
22                 torch.tensor(alpha_N)), torch.log(torch.tensor(
23                     beta_N))
24         ], requires_grad=True)
25
26     def get_actual_parameters(self):
27         mu_N = self.parameters[0]
28
29         lambda_N = torch.exp(self.parameters[1])
30         alpha_N = torch.exp(self.parameters[2])
31         beta_N = torch.exp(self.parameters[3])
32
33         return mu_N, lambda_N, alpha_N, beta_N
34
35     def get_parameters(self):
36         return self.parameters.clone()
37
38     def set_parameters(self, new_params):
39         self.parameters = new_params.clone()
40
41     def sample(self):
42         """Sample from the variational distribution.
43          $z[s] \sim q(\mu, \tau | \lambda)$ 
44         """

```



```

45     mu_N, lambda_N, alpha_N, beta_N = self.
46         get_actual_parameters()
47
48     tau = torch.distributions.Gamma(alpha_N, beta_N).sample()
49
50     precision = lambda_N * tau
51
52     sigma_mu = 1.0 / torch.sqrt(precision)
53
54     mu = torch.distributions.Normal(mu_N, sigma_mu).sample()
55
56     return (mu, tau)
57
58 def log_prob(self, z):
59     """Compute log probability of z under the variational
60         distribution.
61         Compute log q( $\mu$ ,  $\tau$  |  $\lambda$ ).
62     """
63     mu, tau = z
64     mu_N, lambda_N, alpha_N, beta_N = self.
65         get_actual_parameters()
66
67     log_q_tau = torch.distributions.Gamma(alpha_N, beta_N).
68         log_prob(tau)
69
70     precision = lambda_N * tau
71
72     sigma_mu = 1.0 / torch.sqrt(precision)
73
74     log_q_mu_given_tau = torch.distributions.Normal(mu_N,
75         sigma_mu).log_prob(mu)
76
77     return log_q_tau + log_q_mu_given_tau

```

We store the logarithms of parameters to enforce the necessary constraint that these parameters must remain strictly positive during optimization, thereby allowing the optimization algorithm to operate effectively over the entire unconstrained range of real numbers.

```

1 def log_joint_distribution(D, z):
2     """Compute the log joint distribution log p(D, Z).
3     Args:
4         D: dataset
5         Z: latent variables
6     Returns:
7         log p(D, Z)
8     """

```

```

9      # log p(D, Z) = log p(D|Z) + log p(Z)
10     # Z = (mu, tau)
11
12     mu, tau = z
13     sigma = 1 / torch.sqrt(tau)
14
15     mu_0 = 1.0
16     lambda_0 = 0.1
17     a_0 = 1.0
18     b_0 = 2.0
19
20     #log P(D|Z)
21     log_likelihood = torch.distributions.Normal(mu, sigma).
        log_prob(D).sum()
22
23     #log P(mu , tau) = log P(mu | tau) + log P(tau)
24
25     # Log prior p( $\mu$  |  $\tau$ )
26     precision_mu = lambda_0 * tau
27     sigma_mu = 1.0 / torch.sqrt(precision_mu)
28     log_prior_mu = torch.distributions.Normal(mu_0, sigma_mu).
        log_prob(mu)
29
30     # Log prior p( $\tau$ )
31     log_prior_tau = torch.distributions.Gamma(a_0, b_0).
        log_prob(tau)
32
33
34     return log_likelihood + log_prior_mu + log_prior_tau

```

The initial parameters have been taken from the last assignments ($\mu_0, \lambda_0, \alpha_0, \beta_0$)
Application of the model

```

1  q = NormalGammaVariationalFamily()
2
3  bbvi = BlackBoxVI(dataset, log_joint_distribution=
    log_joint_distribution, S = 30, variational_family_q=q,
    learning_rate=1e-5)
4  results = bbvi.fit(max_iterations=10**4)
5
6  print(results['mu_expected'][-1])
7  print(f"\n{'='*60}")
8  print(f"Final Results:")
9  print(f"E_q[ $\mu$ ] = {results['mu_expected'][-1][1]:.4f} (true: {mu})"
    )
10 print(f"E_q[ $\tau$ ] = {results['tau_expected'][-1][1]:.4f} (true: {tau
    })")
11 print(f>Data mean: {dataset.mean():.4f}")
12 print(f>Data precision: {1.0/dataset.var():.4f}")
13 print(f"{'='*60}")

```

```

14
15 ### Plot of the ELBO
16 plt.figure(figsize=(14, 6))
17 iterations, elbos = zip(*results["elbo"])
18 plt.plot(iterations, elbos, 'b-')
19 plt.xlabel("Iteration")
20 plt.ylabel("Elbo")
21 plt.title('ELBO over Iterations', fontsize=14)
22 plt.grid(alpha=0.3)
23
24
25 ### Plot for m
26 plt.figure(figsize=(14, 6))
27 iterations, mu_expected = zip(*results["mu_expected"])
28 plt.plot(iterations, mu_expected, 'b-')
29 plt.axhline(mu, color='r', linestyle='--', label="real value")
30 plt.xlabel("Iteration")
31 plt.ylabel("mu")
32 plt.title('SGD: Convergence of Mean (m)', fontsize=14)
33 plt.grid(alpha=0.3)
34
35
36
37
38 ### Plot for tau
39 plt.figure(figsize=(14, 6))
40 iterations, tau_expected = zip(*results["tau_expected"])
41 plt.plot(iterations, tau_expected, 'b-')
42 plt.axhline(tau, color='r', linestyle='--', label="real value")
43 plt.xlabel("Iteration")
44 plt.ylabel("tau")
45 plt.title('SGD: Convergence of Tau', fontsize=14)
46 plt.grid(alpha=0.3)

```

Conclusions

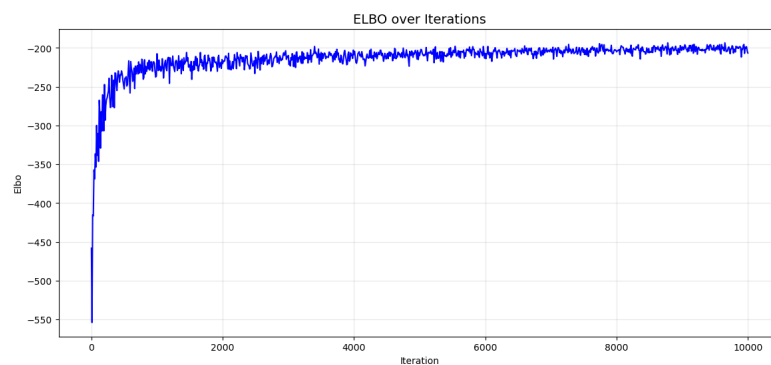


Figure 2: Evidence Lower Bound (ELBO) over training iterations.

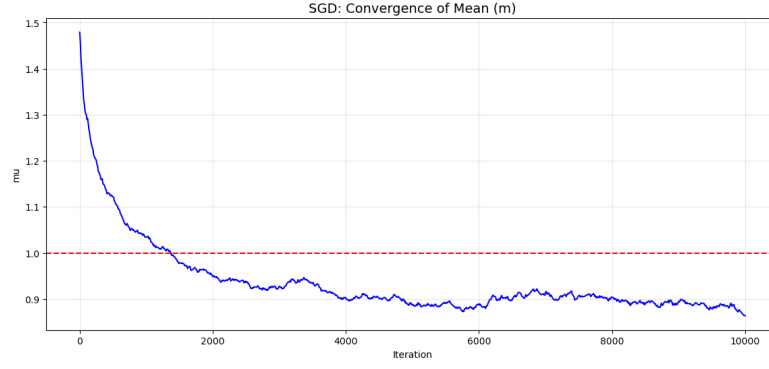


Figure 3: Convergence results for Tau parameter.

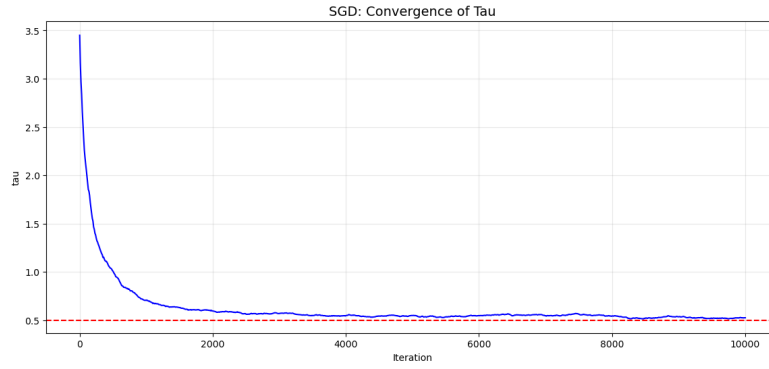


Figure 4: Convergence results for the Tau parameter.

Parameter	Variational Estimate ($E_q[\cdot]$)	True/Data Value
Mean (μ)	0.8634	1.0000
Precision (τ)	0.5255	0.5000
Dataset Mean	N/A	0.8901
Dataset Precision	N/A	0.4580

Table 1: Data Statistics

C Section

C.1

Question 2.1.10

We want to show the the IWELBO is a valid lower bound on the log-marginal likelihood.

$$\mathcal{L}_K = \mathbb{E}_{Z_1, \dots, Z_K} \left[\log \left(\frac{1}{K} \sum_{k=1}^K \frac{p(X, Z_k)}{q(Z_k|X)} \right) \right]$$

$$\mathcal{L}_K \leq \log p(X)$$

We start the proof by applying the Jensen's equality

$$\mathbb{E}_Z [\log f(Z)] \leq \log \mathbb{E}_Z [f(Z)]$$

The IWELBO becomes:

$$\begin{aligned} \mathbb{E}_{Z_1, \dots, Z_K} \left[\log \left(\frac{1}{K} \sum_{k=1}^K \frac{p(X, Z_k)}{q(Z_k|X)} \right) \right] &\leq \log \left(\frac{1}{K} \sum_{k=1}^K \mathbb{E}_{Z_K} \left[\frac{p(X, Z_k)}{q(Z_k|X)} \right] \right) \\ &\leq \log \left(\frac{1}{K} \sum_{k=1}^K \int \frac{P(Z_k, X)}{q(Z_k|X)} q(Z_k|X) dZ_k \right) \\ &\leq \log \frac{1}{K} \sum_{k=1}^K p(X) \\ &\leq \log P(X) \end{aligned}$$

This is the expected result

$$\mathcal{L}_K \leq \log P(X)$$

Question 2.1.11

We want to prove that the IWELBO tightens the variational bound for K samples

$$\mathcal{L}_K \geq \mathcal{L}_1 \quad K > 1$$

We know that

$$\mathcal{L}_K = \mathbb{E}_{Z_1, \dots, Z_K} \left[\log \left(\frac{1}{K} \sum_{k=1}^K \frac{p(X, Z_k)}{q(Z_k|X)} \right) \right] \quad \mathcal{L}_1 = \mathbb{E}_Z \left[\log \frac{P(X, Z)}{q(Z)} \right]$$

We start by observing

$$\mathbb{E}_{Z_1, \dots, Z_K} \left[\log \left(\frac{1}{K} \sum_{k=1}^K \frac{p(X, Z_k)}{q(Z_k|X)} \right) \right] = \mathbb{E}_{Z_1, \dots, Z_K} \left[\log \mathbb{E}_{I=\{k_1\}} \left[\frac{p(X, Z_{k_1})}{q(Z_{k_1}|X)} \right] \right]$$

I is a set which has size equal to 1 (it contains only one index k_1 which is randomly chosen from $1, 2, \dots, K$ with equal probability $\frac{1}{K}$)

$$\mathbb{E}_{I=\{k_1\}} \left[\frac{a_{k_1}}{1} \right] = \frac{a_1 + \dots + a_K}{K}$$

In other words, it is another way of expressing the sum.

Then, we apply the Jensen's inequality

$$\mathcal{L}_K \geq \mathbb{E}_{Z_1, \dots, Z_K} \left[\mathbb{E}_{I=\{k_1\}} \left[\log \frac{p(X, Z_{k_1})}{q(Z_{k_1}|X)} \right] \right]$$

We invert the two expectations

$$\mathcal{L}_K \geq \mathbb{E}_{I=\{k_1\}} \left[\mathbb{E}_{Z_{k_1}} \left[\log \frac{p(X, Z_{k_1})}{q(Z_{k_1}|X)} \right] \right]$$

Since I uniformly select one index from $1, \dots, K$ and each z_k is i.i.d from $q(z|x)$ and the inner expectation gives the same value for any choice of k_1

$$\mathbb{E}_{I=\{k_1\}} \left[\mathbb{E}_{Z_{k_1}} \left[\log \frac{p(X, Z_{k_1})}{q(Z_{k_1}|X)} \right] \right] = \mathbb{E}_z \left[\log \frac{p(X, Z)}{q(Z|X)} \right] = \mathcal{L}_1$$

We have proved that

$$\mathcal{L}_K \geq \mathcal{L}_1$$

C.2

Question 2.1.12

Considering the model described below and the mean-field approximation, we derive the Rao-Blackwellized partial gradient of the ELBO w.r.t λ_3

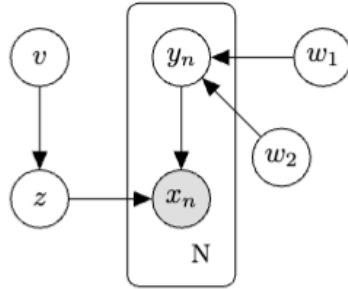


Figure 5: PGM of some generic model for Rao-Blackwellization

$$q(w_1, w_2, z, v, y) = q_{\lambda_1}(w_1)q_{\lambda_2}(w_2)q_{\lambda_3}(z)q_{\lambda_4}(v) \prod_n q_{\lambda_{5,n}}(y_n).$$

$$\nabla_{\lambda_3} \mathcal{L} = E_{q_{(3)}} \left[\nabla_{\lambda_3} \log q(z_3|\lambda_3) (\log p_3(x, z_{(3)}) - \log q(z_3|\lambda_3)) \right]$$

Where:

- $z_3 = z$
- $z_{(3)} = z_3 \cup \text{Markov Blanket of } z_3 = \{z, v, x_n, y_n\} \forall n$
 - Markov Blanket of a variable z_3 is all the parents, co-parents and children of $z_3 \rightarrow \{v, x_n, y_n\} \forall n$
- $\log p_3(x, z)$ is the the portion of the joint probability that depends on $z_3 = z$

$$\log p_3(x, z) = \log p(z|v) + \sum_{n=1}^D \log p(x_n|z, y_n)$$

- $q_{(3)}$ is the distribution of variables in the model that depend on the third variable z (the Markov Blanket of z and z)

$$q_{(3)}(z_{(3)}) = q_{(3)}(z) = \prod_{z_j \in z_{(3)}} q(z_j|\lambda_j) = q(z|\lambda_3) \cdot q(v|\lambda_4) \cdot \prod_n q(y_n|\lambda_{5,n})$$

C.3

Question 2.2.13

We extend the implementation of problem 1.D.3 with the Control Variate used in the BBVI paper (without Rao-blackwellization). We provide also the same plots seen for 1.D.3. Preliminary informations:

- D is the number of variational parameters $\lambda = \{\mu_N, \lambda_N, \alpha_N, \beta_N\}$
- Z is $\{\mu, \tau\}$
- S is the number of Monte Carlo samples

Algorithm 2 Black Box Variational Inference (II)

Input: data x , joint distribution p , mean field variational family q .
Initialize $\lambda_{1:n}$ randomly, $t = 1$.
repeat
 // Draw S samples from the variational approximation
 for $s = 1$ **to** S **do**
 $z[s] \sim q$
 end for
 for $d = 1$ **to** D **do**
 for $s = 1$ **to** S **do**
 $f_d[s] = \nabla_{\lambda_d} \log q_t(z[s] | \lambda_t) (\log p_t(x, z[s]) - \log q_t(z[s] | \lambda_t))$
 $h_d[s] = \nabla_{\lambda_d} \log q_t(z[s] | \lambda_t)$
 end for
 $\hat{a}_d^* = \frac{\text{Cov}(f_d, h_d)}{\text{Var}(h_d)}$, Estimate from a few samples
 $\hat{V}_{\lambda_d} \mathcal{L} \triangleq \frac{1}{S} \sum_{s=1}^S f_d[s] - \hat{a}_d^* h_d[s]$
 end for
 $\rho = t$ th value of a Robbins Monro sequence
 $\lambda = \lambda + \rho \hat{V}_{\lambda} \mathcal{L}$
 $t = t + 1$
until change of λ is less than 0.01.

Figure 6: BBVI II

We just ignore the pedix of p and q in the algorithm. The generation code is the same of the 1.D.3

```

1 class BlackBoxVI:
2     """Black Box Variational Inference implementation."""
3
4     def __init__(self, D, log_joint_distribution,
5         variational_family_q, S=10, learning_rate=1e-3):
6         """
7         Black Box Variational Inference implementation without
8         any .
9         Args:
10             D: dataset
11             log_joint_distribution: function that computes
12             the log joint distribution
13             variational_family_q: variational family q
14             with parameters to optimize
15             S: number of samples for Monte Carlo
16             estimation
17             learning_rate: learning rate for the optimizer
18         """
19         self.D = D
20         self.log_joint_distribution = log_joint_distribution
21         self.variational_family_q = variational_family_q
22         self.S = S
23         self.learning_rate = learning_rate
24
25     def fit(self, max_iterations = 1000):
26         """Fit the variational parameters using BBVI algorithm
27         .
28         SGD optimization
29         Args:
30             threshold: convergence threshold
31             max_iterations: maximum number of iterations
32         """
33         history = {
34             "elbo" : [],
35             "final_params" : None,
36             'mu_expected': [],
37             'tau_expected': [],
38         }
39         lr_sgd = self.learning_rate # A small learning
40             rate
41         optimizer_sgd = optim.SGD([self.variational_family_q.
42             parameters], lr=lr_sgd)
43
44         for t in range(1, max_iterations+1):
45             elbo=0
46             loss=0
47             optimizer_sgd.zero_grad()

```



```

42         number_of_variational_parameters = self.
           variational_family_q.get_number_of_parameters
           ()
43
44         f = torch.zeros((number_of_variational_parameters,
45                           self.S))
46         h = torch.zeros((number_of_variational_parameters,
47                           self.S))
48
49         for s in range(self.S):
50             z_s = self.variational_family_q.sample()
51
52             #compute log(q(z[s]))
53             log_q_z_s = self.variational_family_q.log_prob
54                 (z_s)
55
56             # Compute log p(x, z[s])
57             log_p_z_s_D = self.log_joint_distribution(self
58                 .D, z_s)
59
60             # Compute the score function  $\nabla_{\lambda} \log q(z[s]; \lambda)$ 
61             learning_signal = (log_p_z_s_D - log_q_z_s).
62                 detach()
63
64             #loss
65             elbo += learning_signal
66
67             # We need to compute  $\nabla_{\lambda} \log q(z[s]; \lambda)$  for
68             # each parameter
69             grad_log_q_z_s = torch.autograd.grad(
70                 log_q_z_s,
71                 self.variational_family_q.parameters,
72                 retain_graph=True,
73                 create_graph=False)[0]
74
75             # For each variational parameter d
76             for d in range(
77                 number_of_variational_parameters):
78                 # Control variate computation
79                 grad_d = grad_log_q_z_s[d]
80
81                 # Compute f_t and h_t for this sample
82                 f[d, s] += learning_signal * grad_d
83                 h[d, s] += grad_d
84
85         elbo /= self.S

```

```

81         # Compute gradient with control variates for each
           parameter d
82         final_gradient = torch.zeros(
           number_of_variational_parameters)
83
84         for d in range(number_of_variational_parameters):
85             # a_d* = Cov(f_d, h_d) / Var(h_d)
86
87             f_d_mean = f[d].mean()
88             h_d_mean = h[d].mean()
89
90             # Covariance
91             cov_f_h = ((f[d] - f_d_mean) * (h[d] -
           h_d_mean)).mean()
92
93             # Variance of h_d
94             var_h = ((h[d] - h_d_mean) ** 2).mean()
95
96             # We add small epsilon to avoid division by
           zero
97             if var_h > 1e-8:
98                 a_d_star = cov_f_h / var_h
99             else:
100                 a_d_star = 0.0
101
102             #  $\nabla_{\lambda} L \approx (1/S) \sum [f_i[s] - a_d^* h_i[s]]$ 
103             final_gradient[d] = (f[d] - a_d_star * h[d]).
           mean()
104
105         with torch.no_grad():
106             self.variational_family_q.parameters += self.
           learning_rate * final_gradient
107
108         #we compute the elb and we check the params every
           10 iterations and at the first iteration
109         if t % 10 == 0 or t == 1:
110             mu_N, lambda_N, alpha_N, beta_N = self.
           variational_family_q.get_actual_parameters
           ()
111             history["elbo"].append((t, elbo))
112             history['mu_expected'].append((t, mu_N.item()))
113             history['tau_expected'].append((t, (alpha_N /
           beta_N).item()))
114
115         history['final_params'] = self.variational_family_q.
           get_parameters()
116
117         return history

```

The rest of the code follows the same structure as before and is omitted for brevity (the entire code can be found in the appendix)

Brief explanation of the changes

For each Monte Carlo sample s , the algorithm computes two critical matrices:

$f[d, s]$ represents the “learning signal times gradient” for parameter d at sample s . Specifically:

- First, compute the learning signal: $\log p(D, z[s]) - \log q(z[s]; \lambda)$, which measures how well sample $z[s]$ explains the data
- Then compute $\nabla_{\lambda} \log q(z[s]; \lambda)$ using PyTorch’s autograd, giving gradients for all parameters
- For each parameter d : $f[d, s] = \text{learning_signal} \times \text{grad}_d$

$h[d, s]$ stores just the score function gradient for parameter d at sample s :

$$h[d, s] = \nabla_{\lambda_d} \log q(z[s]; \lambda) \quad (1)$$

These matrices have dimensions $[\text{number_of_parameters} \times S]$, storing all gradient information across samples.

The control variates method exploits a key mathematical property: $\mathbb{E}[h[d, s]] = 0$ under the variational distribution q . This means h can serve as a **zero-mean baseline** that, when properly scaled, reduces variance without introducing bias.

Conclusions

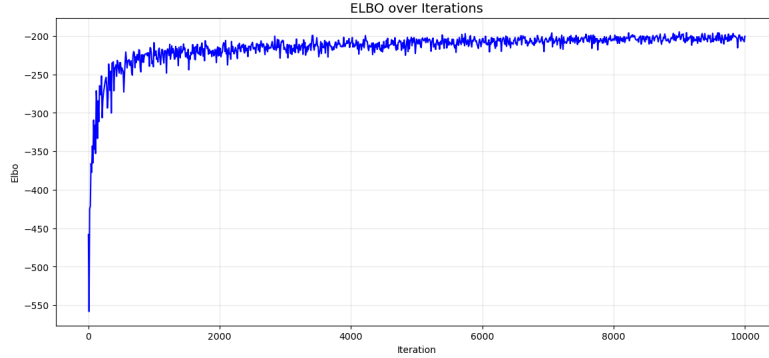


Figure 7: Evidence Lower Bound (ELBO) over training iterations.

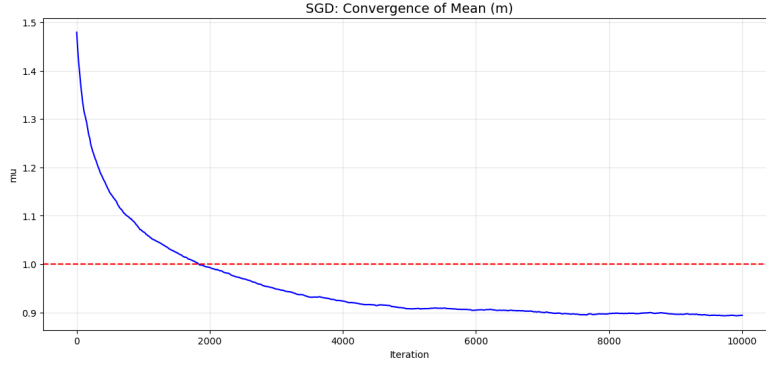


Figure 8: Convergence results for Tau parameter.

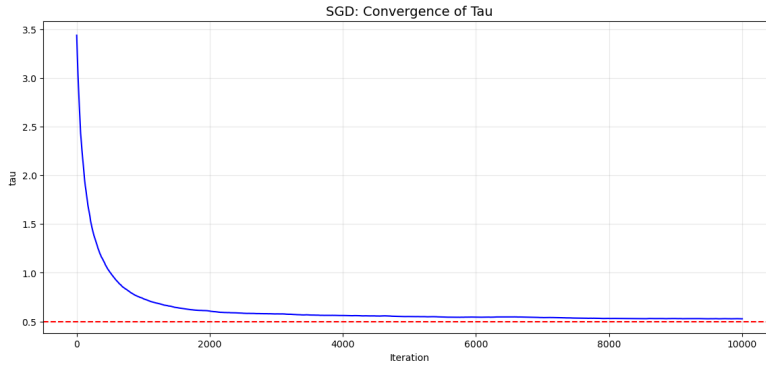


Figure 9: Convergence results for the Tau parameter.

Parameter	Variational Estimate ($E_q[\cdot]$)	True/Data Value
Mean (μ)	0.8939	1.0000
Precision (τ)	0.5269	0.5000
Dataset Mean	N/A	0.8901
Dataset Precision	N/A	0.4580

Table 2: Data Statistics

The variance reduction results in more stable parameter updates, allowing the algorithm to take more confident steps toward the optimum. This is particularly evident in the steeper initial descent and the reduced noise in later iterations, confirming that control variates substantially improve the efficiency and reliability of the BBVI optimization process.

C.4

We study the Gamma distribution and use its exponential-family structure to derive the Fish Information Matrix (FIM) and implement Natural Gradient Descent (NGD).

We consider the Gamma distribution parameterized by shape $\alpha > 0$ and $\beta > 0$

$$p(x | \alpha, \beta) = \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1} \exp\left(-\frac{x}{\beta}\right), \quad x > 0.$$

Question 2.2.14

We rewrite the Gamma Distribution in canonical exponential-family form identifying the natural parameters η , sufficient statistics $t(x)$ and log-normalizer $A(\eta)$.

We also derive gradient $\nabla_\eta A(\eta)$

$$p(x|\eta) = h(x)e^{\eta^T t(x) - A(\eta)}$$

We start by considering the definition of Gamma Distribution pdf

$$\begin{aligned} p(x|\alpha, \beta) &= \frac{1}{\Gamma(\alpha)} x^{\alpha-1} e^{\frac{-x}{\beta} - \alpha \log \beta} \\ &= x^{\alpha-1} e^{\frac{-x}{\beta} - \alpha \log \beta - \log \Gamma(\alpha)} \\ &= e^{-\frac{x}{\beta} + (\alpha-1) \log x - (\alpha \log \beta + \log \Gamma(\alpha))} \end{aligned}$$

Where:

- $t(x) = \begin{pmatrix} x \\ \log x \end{pmatrix}$
- $\eta = \begin{pmatrix} -\frac{1}{\beta} \\ \alpha-1 \end{pmatrix} = \begin{pmatrix} \eta_1 \\ \eta_2 \end{pmatrix}$
- $\beta = -\frac{1}{\eta_1}$
- $\alpha = \eta_2 + 1$
- $A(\eta)$

$$\begin{aligned} A(\eta) &= \alpha \log \beta + \log \Gamma(\alpha) \\ &= -(\eta_2 + 1) \log(-\eta_1) + \log \Gamma(\eta_2 + 1) \end{aligned}$$

- $\nabla_\eta A(\eta)$

$$\nabla_\eta A(\eta) = \begin{bmatrix} \nabla_{\eta_1} A(\eta) \\ \nabla_{\eta_2} A(\eta) \end{bmatrix} = \begin{bmatrix} -\frac{(\eta_2+1)}{\eta_1} \\ -\log(-\eta_1) + \frac{\Gamma'(\eta_2+1)}{\Gamma(\eta_2+1)} \end{bmatrix} = \begin{bmatrix} +\beta\lambda \\ +\log(\beta) + \Psi(a) \end{bmatrix} = \begin{bmatrix} E[x] \\ E[\log x] \end{bmatrix}$$

Note: Digamma Function

$$\Psi(z) = \frac{d\Gamma(z)}{\Gamma(z)}$$

Question 2.2.15

We are considering MSE as the differentiable loss function $L(\alpha, \beta)$

$$L(\alpha, \beta) = \frac{1}{N} \sum_{i=1}^N (x_i - \tilde{x}_i(\alpha, \beta))^2$$

We use the properties of exponential-family distributions and the Fisher Information $G(\alpha, \beta)$ for expressing the natural gradient $\tilde{\nabla}_{(\alpha, \beta)} \mathcal{L}$ as a function of the standard gradient $\nabla_{(\alpha, \beta)} \mathcal{L}$ and the FIM

The general formula is given from the Stochastic Variational Inference paper of Hoffman, Blei, Wang and Paisley:

$$\tilde{\nabla}_{(\alpha, \beta)} \mathcal{L} = G(\alpha, \beta)^{-1} \nabla_{(\alpha, \beta)} \mathcal{L}$$

In particular, for exponential-family distribution, we can express the relation between the two types of gradients as:

$$\tilde{\nabla}_{(\alpha, \beta)} \mathcal{L} = \left[\nabla_{(\alpha, \beta)}^2 A(\alpha, \beta) \right]^{-1} \nabla_{(\alpha, \beta)} \mathcal{L}$$

Question 2.2.16

We compute the inverse Fisher Information Matrix $F(\alpha, \beta)^{-1}$ explicitly. We verify your expression numerically.

$$\begin{aligned} G(\alpha, \beta) &= \nabla_{\alpha, \beta}^2 A(\alpha, \beta) \\ G(\eta) &= \nabla_{\eta}^2 A(\eta) \\ &= \begin{pmatrix} \nabla_{\eta_1} \nabla_{\eta_1} A(\eta) & \nabla_{\eta_1} \nabla_{\eta_2} A(\eta) \\ \nabla_{\eta_1} \nabla_{\eta_2} A(\eta) & \nabla_{\eta_2} \nabla_{\eta_2} A(\eta) \end{pmatrix} \\ &= \begin{pmatrix} \frac{(\eta_2+1)}{\eta_1^2} & -\frac{1}{\eta_1} \\ -\frac{1}{(-\eta_1)} & \psi'(\eta_2+1) \end{pmatrix} \\ &= \begin{pmatrix} \frac{\eta_2+1}{\eta_1^2} & -\frac{1}{\eta_1} \\ -\frac{1}{\eta_1} & \psi'(\eta_2+1) \end{pmatrix} \end{aligned}$$

Knowing that $\eta_1 = -\frac{1}{\beta}$ and $\eta_2 = \alpha - 1$, we can express $G(\eta)$ as $G(\alpha, \beta)$:

$$G(\alpha, \beta) = \begin{bmatrix} \beta^2 \alpha & \beta \\ \beta & \psi'(\alpha) \end{bmatrix}$$

Then, we use the inverse formula for 2x2 matrices

$$E = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{bmatrix} \beta^2 \alpha & \beta \\ \beta & \psi'(\alpha) \end{bmatrix}$$

$$\begin{aligned}
E^{-1} &= \frac{1}{ad-bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \\
&= \frac{1}{\beta^2(\alpha\psi'(\alpha) - 1)} \begin{bmatrix} \psi'(\alpha) & -\beta \\ -\beta & \beta^2\alpha \end{bmatrix} \\
&= \frac{1}{\alpha\psi'(\alpha) - 1} \begin{bmatrix} \frac{\psi'(\alpha)}{\beta^2} & -\frac{1}{\beta} \\ -\frac{1}{\beta} & \alpha \end{bmatrix}
\end{aligned}$$

We choose $\alpha = 2$ and $\beta = 3$
 $\psi'(\alpha) = 0.6449$

$$G(2, 3) = \begin{pmatrix} 18 & 3 \\ 3 & 0.64473 \end{pmatrix}$$

$$G^{-1}(2, 3) = \frac{1}{0.2878} \begin{pmatrix} 0.071653 & -\frac{1}{3} \\ -\frac{1}{3} & 2 \end{pmatrix} = \begin{pmatrix} 0.24922 & -1.15022 \\ -1.15022 & 6.90734 \end{pmatrix}$$

We verify numerically that : $GG^{-1} = I$

$$\begin{pmatrix} 18 & 3 \\ 3 & 0.64473 \end{pmatrix} \begin{pmatrix} 0.24922 & -1.15022 \\ -1.15022 & 6.90734 \end{pmatrix} = \begin{pmatrix} 1 & \approx 0 \\ \approx 0 & 1 \end{pmatrix}$$

The Fisher information for the Gamma distribution can take slightly different forms depending on whether the parameter β , which could be interpreted as

- a scale parameter (as in the assignment PDF) $\text{Gamma}(\alpha, \beta_{\text{scale}})$
- as a rate parameter (as used in the instructor's code) $\text{Gamma}(\alpha, \lambda_{\text{rate}})$ where $\lambda_{\text{rate}} = \frac{1}{\beta_{\text{scale}}}$

Since the exponential-family formulation depends on this choice, the corresponding Fisher matrices appear different at first glance.

In this report, I followed the definition of β provided in the PDF, where β represents the scale. When converting to the parameterization used in the instructor's implementation (where β is treated as a rate-like quantity) the expressions align perfectly after applying the appropriate change of variables. Thus, the discrepancy is purely due to the difference in parameter meaning (scale vs. rate), and not to an error in the derivation.

Question 2.2.17

Assuming $x_{1:N}$ as i.i.d. samples from $\text{Gamma}(\alpha^*, \beta^*)$, we want to write down the gradient of the following average negative log-likelihood wrt (α, β)

$$\begin{aligned}\mathcal{L}(\alpha, \beta) &= -\frac{1}{N} \sum_{n=1}^N \log p(x_n | \alpha, \beta) \\ &= -\frac{1}{N} \sum_{n=1}^N \log \left(\frac{1}{\Gamma(\alpha) \beta^\alpha} x_n^{\alpha-1} e^{-\frac{x_n}{\beta}} \right) \\ &= -\frac{1}{N} \sum_{n=1}^N \left[-\alpha \log \beta - \log \Gamma(\alpha) + (\alpha - 1) \log x_n - \frac{x_n}{\beta} \right] \\ &= +\alpha \log \beta + \log \Gamma(\alpha) - \frac{1}{N} \sum_{n=1}^N \left[+(\alpha - 1) \log x_n - \frac{x_n}{\beta} \right] \\ &= +\alpha \log \beta + \log \Gamma(\alpha) + (\alpha - 1) \overline{\log x} + \frac{\bar{x}}{\beta}\end{aligned}$$

Where:

- $\overline{\log x}$ is the empirical mean wrt to $\log x \rightarrow \frac{1}{N} \sum_{n=1}^N \log x_n$
- \bar{x} is the empirical mean wrt to $x \rightarrow \frac{1}{N} \sum_{n=1}^N x_n$

Now, we compute its gradient

$$\begin{aligned}\nabla_{(\alpha, \beta)} \mathcal{L}(\alpha, \beta) &= \begin{pmatrix} \nabla_\alpha \mathcal{L}(\alpha, \beta) \\ \nabla_\beta \mathcal{L}(\alpha, \beta) \end{pmatrix} \\ &= \begin{pmatrix} \log \beta + \psi(\alpha) - \overline{\log x} \\ \frac{\alpha}{\beta} - \frac{\bar{x}}{\beta^2} \end{pmatrix}\end{aligned}$$

$$\psi = (\log \Gamma(\alpha))' = \frac{d\Gamma(\alpha)}{\Gamma(\alpha)}$$

Question 2.2.18

We implement a python notebook for estimating (α, β) for a $\text{Gamma}(\alpha^* = 3.0, \beta^* = 2.0)$ with a dataset of 1000 points and an initial poor guess. We use both standard gradient descent (GD) and natural gradient descent (NGD) ensuring $\alpha > 0$ and $\beta > 0$ during optimization

Definition of hyperparameters and some model settings

```
1 import torch
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from torch.distributions import Gamma
5 # -----
```



```

6
7 # True parameters of the Gamma distribution we want to discover
8 ALPHA_TRUE = 3.0    # shape
9 BETA_TRUE  = 2.0    # scale
10
11 # Number of observed data points
12 N_DATA = 1000
13
14 # Optimization parameters
15 LEARNING_RATE = 0.1
16 EPOCHS = 150
17
18 # Initial "wrong" guess for our parameters (still positive)
19 ALPHA_INIT = 0.5
20 BETA_INIT  = 8.0
21
22 # Fix random seed for reproducibility (optional)
23 torch.manual_seed(0)

```

```

1 # Our parameterisation is shape-scale, but PyTorch's Gamma uses
  shape-rate.
2 rate_true = 1.0 / BETA_TRUE
3 dist_true = Gamma(concentration=torch.tensor(ALPHA_TRUE),
4                   rate=torch.tensor(rate_true))
5
6 data = dist_true.sample((N_DATA,))
7
8 print(f"Generated {N_DATA} data points from Gamma(alpha={
9       ALPHA_TRUE}, beta={BETA_TRUE})")
10 print(f"Sample mean:      {data.mean().item():.4f}")
11 print(f"Sample variance: {data.var().item():.4f}\n")

```

Implementation of the negative log-likelihood lossfunction for the Gamma distribution.

```

1 # ToDo: Loss function
2 def gamma_nll(alpha, beta, data_points):
3     """
4     ToDo:
5         Implement the average negative log-likelihood for Gamma
6         distribution with shape=alpha and scale=beta.
7
8     Hints:
9         - Enforce positivity using clamp (e.g. min=1e-4).
10        - PyTorch's Gamma takes (concentration=alpha, rate=1/beta)
11
12        - Return the *mean* negative log-likelihood.
13
14    """
15     alpha = torch.clamp(alpha, min=1e-4)
16     beta = torch.clamp(beta, min=1e-4)

```

```

15     # Convert data_points to tensor if not already
16     data_tensor = torch.tensor(data_points) if not torch.is_tensor
17         (data_points) else data_points
18
19     empirical_mean_x = data_tensor.mean()
20     empirical_mean_log_x = torch.log(data_tensor).mean()
21     nll = +alpha*torch.log(beta) + torch.lgamma(alpha) - (alpha-1)
22         *empirical_mean_log_x + (1/beta)*empirical_mean_x
23     return nll

```

```

1  # Parameters for Standard Gradient Descent (GD)
2  alpha_gd = torch.tensor(ALPHA_INIT, requires_grad=True)
3  beta_gd  = torch.tensor(BETA_INIT,  requires_grad=True)
4
5  # Parameters for Natural Gradient Descent (NGD)
6  alpha_ngd = torch.tensor(ALPHA_INIT, requires_grad=True)
7  beta_ngd  = torch.tensor(BETA_INIT,  requires_grad=True)
8
9  # History trackers
10 history_gd  = []
11 history_ngd = []
12 history_loss_gd = []
13 history_loss_ngd = []

```

We detach because the Fisher Information matrix (and its inverse) is a curvature estimate used to precondition the gradient updates. It should not be part of the computational graph for gradient computation.

```

1  def fisher_inverse(alpha, beta):
2      """
3      TODO:
4      Implement the inverse Fisher Information matrix  $F^{-1}(\alpha, \beta)$ 
5      for the Gamma(shape= $\alpha$ , scale= $\beta$ ) distribution.
6
7      Theory:
8       $F(\alpha, \beta) =$ 
9      
$$\begin{bmatrix} \psi_1(\alpha) & 1/\beta \\ 1/\beta & \alpha/\beta^2 \end{bmatrix}$$

10
11       $F^{-1}(\alpha, \beta) =$ 
12      
$$\frac{1}{(\alpha \psi_1(\alpha) - 1)} \begin{bmatrix} \alpha & -\beta \\ -\beta & \beta^2 \psi_1(\alpha) \end{bmatrix}$$

13
14      Hints:
15      - Use torch.polygamma(1, alpha) for  $\psi_1(\alpha)$  (trigamma).
16      - Make sure to detach alpha, beta so  $F^{-1}$  is not part of
17        the graph.
18      """
19
20

```

```

21     alpha_detached = alpha.detach()
22     beta_detached = beta.detach()
23
24     external_factor = 1/(alpha_detached* torch.polygamma(1,
25         alpha_detached)-1)
26     inv11 = alpha_detached * external_factor
27     inv12 = -beta_detached * external_factor
28     inv22 = beta_detached**2 * torch.polygamma(1, alpha_detached)
29         * external_factor
30     return inv11, inv12, inv22

```

We run the optimization loop for a specified number of epochs. In each epoch, we perform both standard gradient descent and natural gradient descent updates. We compute the gradients, build the Fisher Information Matrix for NGD, and update the parameters accordingly. We also log the parameter values and losses at regular intervals.

```

1     print(f"Optimizing with LR={LEARNING_RATE} for {EPOCHS} epochs
2         ...")
3
4     for epoch in range(EPOCHS):
5
6         # ===== A. Standard Gradient Descent (GD) =====
7
8         if alpha_gd.grad is not None:
9             alpha_gd.grad.zero_()
10        if beta_gd.grad is not None:
11            beta_gd.grad.zero_()
12
13        loss_gd = gamma_nll(alpha_gd, beta_gd, data)
14
15        loss_gd.backward()
16
17        with torch.no_grad():
18            alpha_gd -= LEARNING_RATE * alpha_gd.grad
19            beta_gd -= LEARNING_RATE * beta_gd.grad
20
21            alpha_gd.clamp_(min=1e-4)
22            beta_gd.clamp_(min=1e-4)
23
24        history_loss_gd.append(loss_gd.item())
25        history_gd.append((alpha_gd.item(), beta_gd.item()))
26
27        # ===== B. Natural Gradient Descent (NGD) =====
28
29        if alpha_ngd.grad is not None:
30            alpha_ngd.grad.zero_()
31        if beta_ngd.grad is not None:
32            beta_ngd.grad.zero_()
33
34        loss_ngd = gamma_nll(alpha_ngd, beta_ngd, data)

```

```

34     loss_ngd.backward()
35
36     g_alpha = alpha_ngd.grad
37     g_beta  = beta_ngd.grad
38
39     # ToDo : compute natural gradient using  $F^{-1}(\alpha, \beta)$ 
40     # 1) Get  $F^{-1}$  entries using fisher_inverse(...)
41     # 2) Compute:
42     #     - ng_alpha
43     #     - ng_beta
44
45     inv11, inv12, inv22 = fisher_inverse(alpha_ngd, beta_ngd)
46
47     ng_alpha = inv11*g_alpha + inv12*g_beta
48     ng_beta  = inv12*g_alpha + inv22*g_beta
49
50     with torch.no_grad():
51         alpha_ngd -= LEARNING_RATE * ng_alpha
52         beta_ngd  -= LEARNING_RATE * ng_beta
53
54         alpha_ngd.clamp_(min=1e-4)
55         beta_ngd.clamp_(min=1e-4)
56
57     history_loss_ngd.append(loss_ngd.item())
58     history_ngd.append((alpha_ngd.item(), beta_ngd.item()))
59
60     if (epoch + 1) % 15 == 0 or epoch == 0:
61         print(f"\n--- Epoch {epoch + 1} ---")
62         print(f"    GD:  alpha={alpha_gd.item():.4f}, beta={beta_gd.
63             item():.4f}, "
64             f"Loss={loss_gd.item():.4f}")
65         print(f"    NGD: alpha={alpha_ngd.item():.4f}, beta={
66             beta_ngd.item():.4f}, "
67             f"Loss={loss_ngd.item():.4f}")
68     print("\nOptimization finished.")

```

To illustrate the difference between standard gradients and natural gradients, we print out the gradients computed in the first epoch for both methods.

```

1 hist_gd_np  = np.array(history_gd)
2 hist_ngd_np = np.array(history_ngd)
3 hist_loss_gd = np.array(history_loss_gd)
4 hist_loss_ngd = np.array(history_loss_ngd)
5
6
7 fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(12, 10), sharex=
    True)
8 fig.suptitle(f"Gamma: Standard Gradient vs. Natural Gradient ")

```

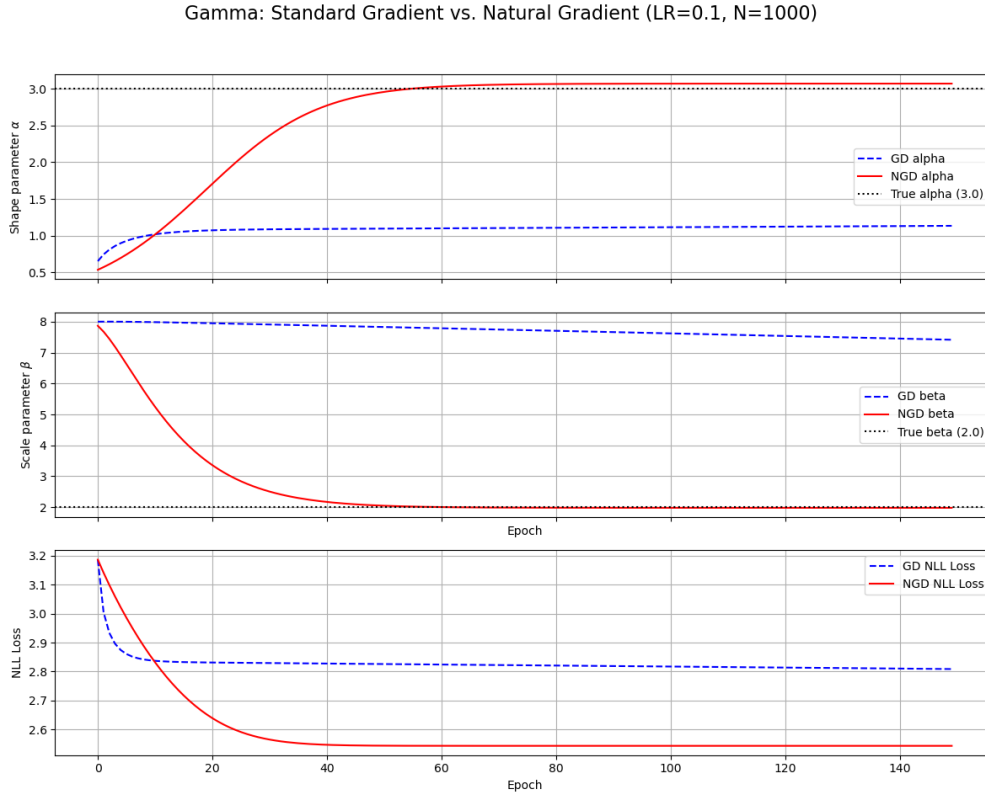
```

9         f"(LR={LEARNING_RATE}, N={N_DATA})", fontsize=16)
10
11 # Plot 1: alpha (shape)
12 ax1.plot(hist_gd_np[:, 0], label="GD alpha", color='blue',
13         linestyle='--')
14 ax1.plot(hist_ngd_np[:, 0], label="NGD alpha", color='red')
15 ax1.axhline(ALPHA_TRUE, color='black', linestyle=':', label=f"True
16         alpha ({ALPHA_TRUE})")
17 ax1.set_ylabel("Shape parameter  $\alpha$ ")
18 ax1.legend()
19 ax1.grid(True)
20
21 # Plot 2: beta (scale)
22 ax2.plot(hist_gd_np[:, 1], label="GD beta", color='blue',
23         linestyle='--')
24 ax2.plot(hist_ngd_np[:, 1], label="NGD beta", color='red')
25 ax2.axhline(BETA_TRUE, color='black', linestyle=':', label=f"True
26         beta ({BETA_TRUE})")
27 ax2.set_xlabel("Epoch")
28 ax2.set_ylabel("Scale parameter  $\beta$ ")
29 ax2.legend()
30 ax2.grid(True)
31
32 # Plot 3: Negative Log-Likelihood (NLL) Loss
33 ax3.plot(hist_loss_gd, label="GD NLL Loss", color='blue',
34         linestyle='--')
35 ax3.plot(hist_loss_ngd, label="NGD NLL Loss", color='red')
36 ax3.set_xlabel("Epoch")
37 ax3.set_ylabel("NLL Loss")
38 ax3.legend()
39 ax3.grid(True)
40
41 plt.tight_layout(rect=[0, 0.03, 1, 0.95])
42 plt.show()

```

Question 2.2.19

We plot and compare the convergence trajectories of GD and NGD for both parameters (α_t, β_t) , as well as the evolution of the negative log-likelihood



Introduction: The natural gradient is the fastest decreasing direction of the error defined in the parameter space in the Riemannian space

The Fisher information matrix (positive definite) defines the local curvature of the model distribution space

Answer: The natural gradient leads to faster and more stable convergence because it corrects the direction of the standard gradient by taking into account the curvature of the probability distribution's manifold (via the Fisher Information Matrix, G). By multiplying the standard gradient with the inverse of the FIM, NGD scales down updates along shallow directions and amplifies updates along directions that cause a large change in the distribution (even if the standard gradient is small), effectively making the optimization landscape appear more like a sphere

B Section

B.1

Question 3.1.20

We know that the categorical distribution is discrete and non-differentiable, making it impossible to backpropagate gradients through samples during training. We can use the Gumbel-Max trick which provides a simple and efficient way to draw samples z from a categorical distribution

$$z = \text{one hot}(\arg \max_i [g_i + \log \pi_i])$$

Where

- $\pi_1 \dots \pi_K$ are the class probabilities (K is the number of class probabilities)
- $g_1 \dots g_K$ are i.i.d. samples drawn from $\text{Gumbel}(0, 1)$

But it is not differentiable: we can approximate the arg max function with a differentiable continuous function called softmax in order to generate k-dimensional sample vectors k-dimensional sample vectors $y \in \Delta^{k-1}$ (one hot encoding)

$$y_i = \frac{e^{\frac{\log \pi_i + g_i}{\tau}}}{\sum_{j=1}^k e^{\frac{\log \pi_j + g_j}{\tau}}}$$

We can define the density of the Gumbel-Softmax distribution as

$$p_{\pi, \tau}(y_1, \dots, y_k) = \Gamma(k) \tau^{k-1} \left(\sum_{i=1}^K \frac{\pi_i}{y_i^\tau} \right)^{-k} \prod_{i=1}^K \frac{\pi_i}{y_i^{\tau+1}}$$

As the softmax temperature τ approaches 0 $\tau \rightarrow 0$, samples from the Gumbel Softmax distribution become one-hot and the Gumbel-Softmax distribution becomes identical to the categorical distribution $p(z)$. As the temperature increase $\tau \rightarrow \infty$, the samples becomes uniform. Below write a function that generates N samples from Categorical a , where $a = [a_0, a_1, a_2, a_3]$.

```
1 def categorical_sampler(a, N):
2     samples = torch.distributions.Categorical(a).sample((N,))
3
4     return samples # should be N-by-1
```

The option `dim=-1` normalizes across the K categories for each sample (each row (representing one sample) will sum to 1)

```
1 # Hint: approximate the Categorical distribution with the Gumbel-
   Softmax distribution
2 def categorical_reparametrize(a, N, temp=0.1, eps=1e-20): # temp
   and eps are hyperparameters for Gumbel-Softmax
```

```

3  g_gumbel_0_1 = torch.distributions.Gumbel(0,1).sample((N, len(a)))
4  )
5  a.requires_grad = True
6  x = torch.log(a + eps) + g_gumbel_0_1
7
8  samples = torch.nn.functional.softmax(x / temp, dim=-1)
9
10 return samples # make sure that your implementation allows the
11                gradient to backpropagate

```

Comparing the samples

```

1  a = torch.tensor([0.1,0.2,0.5,0.2])
2  N = 1000
3  direct_samples = categorical_sampler(a, N)
4  reparametrized_samples = categorical_reparametrize(a, N, temp
5  =0.1, eps=1e-20)
6
7  hard_samples = reparametrized_samples.argmax(dim=1, keepdim=
8  True)
9  compare_samples(direct_samples, hard_samples, bins=4)

```

Results

The graph shows that both methods produce similar distributions (mostly orange visible means they overlap well), which validates that our Gumbel-Softmax implementation is working correctly!

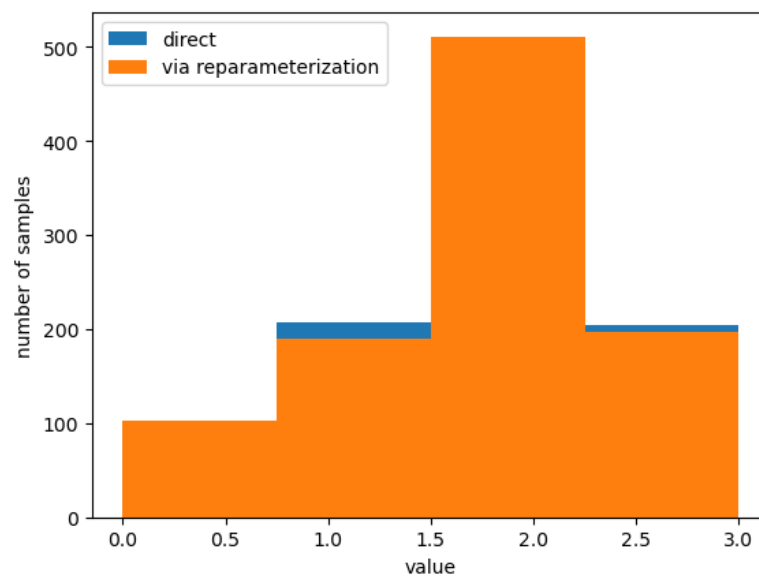


Figure 10: Comparison of samples from direct categorical sampling vs. Gumbel-Softmax reparameterization

B.2

Question 3.1.21

Following the methodology from the Sticking the Landing paper, I derived the gradient of the ELBO under the reparameterization trick and showed how it can be decomposed into multiple terms, one of which is the score function. This decomposition reveals the connection between the reparameterization gradient and the score function gradient.