DD2434 Advanced Machine Learning
Assignment 1AD

Reuben Gezang

December 2025

D-Level

Theory 1.D.1

Question 1.1.1

We start by stating the definition of the Kullback-Leibler divergence:

$$KL(q(Z)\|p(Z|X)) = \mathbb{E}_{q(\boldsymbol{Z})}[\log(\frac{q(Z)}{p(Z|X)})] = \int q(Z)\log(\frac{q(Z)}{p(Z|X)})dZ \tag{1}$$

Now we separate the fraction inside the logarithm:

$$KL(q(Z)\|p(Z|X)) = \int q(Z)\log(\frac{q(Z)p(X)}{p(X,Z)})dZ = \int q(Z)(\log(q(Z)) - \log(p(Z|X))dZ \tag{2}$$

Note that $p(Z|X) = \frac{p(X,Z)}{p(X)}$ and using this we can rewrite the equation as:

$$KL(q(Z)\|p(Z|X)) = \int q(Z)\log(q(Z))dZ - \int q(Z)\log(p(X,Z))dZ + \log(p(X))\int q(Z)dZ \tag{3}$$

Since $q(Z)$ is a probability distribution, we know that $\int q(Z)dZ = 1$. Now we can solve for $\log(p(X))$:

$$\log(p(X)) = KL(q(Z)\|p(Z|X)) - \int q(Z)\log(q(Z))dZ + \int q(Z)\log(p(X,Z))dZ \tag{4}$$

Combining the logarithm terms, we get:

$$\log(p(X)) = KL(q(Z)\|p(Z|X)) + \mathbb{E}_{q(\boldsymbol{Z})}[\frac{\log(p(X,Z))}{q(Z)}] \tag{5}$$

Now we can identify the Evidence lower bound (ELBO)

$$\mathcal{L}(q) = \mathbb{E}_{q(\boldsymbol{Z})}[\frac{\log(p(X,Z))}{q(Z)}] \tag{6}$$

and with this we have shown that:

$$\log(p(X)) = \mathcal{L}(q) + KL(q(Z)\|p(Z|X)) \tag{7}$$

concluding the proof.

Question 1.1.2

In this question we are to describe (in one sentence) how the choice of variational family $q(Z)$ affects

- (i) The tightness of the ELBO
- (ii) The accuracy of the posterior approximation

(i) A more expressive variational family can lead to a tighter ELBO as it can better approximate (and better match) the true posterior, reducing the KL divergence term.

(ii) The choice of variational family directly impacts the accuracy of the posterior approximation, as a limited family may not capture the true posterior's complexity, leading to a less accurate approximation.

Question 1.D.2

1.1.3

For a mean field assumption and joint distribution

$$q(Z_1, Z_2, Z_3) = q_1(Z_1)q_2(Z_2)q_3(Z_3), \quad p(X, Z)$$

Let $q_1^*(Z_1)$ be the q_1 that maximizes the ELBO. We want to show that q_1^* satisfies

$$\log q_1^*(Z_1) = \mathbb{E}_{-Z_1}[\log p(X, Z)]$$

We can start by inspecting the ELBO:

$$\mathcal{L}(q) = \mathcal{L}(q) = \mathbb{E}_{q(\boldsymbol{Z})}[\log(\frac{p(X, Z)}{q(Z)})] = \mathbb{E}_q[\log p(X, Z)] - \mathbb{E}_q[\log q(Z)]$$

and using the mean field assumption we can rewrite this as:

$$\mathcal{L}(q) = \mathbb{E}_{q(Z)}[\log p(X, Z)] - \mathbb{E}_{q(Z)}[\log(q_1(Z_1)q_2(Z_2)q_3(Z_3))]$$

and by separating the logarithm and taking expectations over the relevant distribution (z_i is independent of z_j for $i \neq j$) we get:

$$\mathbb{E}_{q(Z)}[\log(q_1(Z_1)q_2(Z_2)q_3(Z_3))] = \mathbb{E}_{q_1(Z_1)}[\log(q_1(Z_1))] + \mathbb{E}_{q_2(Z_2)}[\log(q_2(Z_2))] + \mathbb{E}_{q_3(Z_3)}[\log(q_3(Z_3))]$$

Since we are maximizing w.r.t $q_1(Z_1)$ we can ignore the terms that do not depend on it. Thus we can rewrite the ELBO as:

$$\mathcal{L}(q) = \mathbb{E}_{q(Z)}[\log p(X, Z)] - \mathbb{E}_{q_1(Z_1)}[\log(q_1(Z_1))] + C$$

where C is a constant w.r.t $q_1(Z_1)$ (and can thus be ignored). Now we can rewrite the expectation over $q(Z)$ as:

$$\mathbb{E}_{q(Z)}[\log p(X, Z)] = \mathbb{E}_{q_1(Z_1)}[\mathbb{E}_{q_2(Z_2)q_3(Z_3)}[\log p(X, Z)]]$$

meaning that we can rewrite the ELBO as:

$$\mathcal{L}(q) = \mathbb{E}_{q_1(Z_1)}[\mathbb{E}_{q_2(Z_2)q_3(Z_3)}[\log p(X, Z)]] - \mathbb{E}_{q_1(Z_1)}[\log(q_1(Z_1))] + C$$

Using the fact that $\int_{Z_1} q(Z_1)dZ_1 = 1$ we will now optimize the ELBO w.r.t $q_1(Z_1)$ and with a lagrange multiplier λ.

$$\frac{\partial}{\partial q_1(Z_1)}\left(\mathbb{E}_{q_1(Z_1)}[\mathbb{E}_{q_2(Z_2)q_3(Z_3)}[\log p(X,Z)]] - \mathbb{E}_{q_1(Z_1)}[\log(q_1(Z_1))] + \lambda(\int_{Z_1} q(Z_1)dZ_1 - 1)\right) = 0 \tag{8}$$

giving that

$$\mathbb{E}_{q_2(Z_2)q_3(Z_3)}[\log p(X,Z)] - \log(q_1(Z_1)) - 1 + \lambda = 0 \rightarrow \log(q_1^*(Z_1)) = \mathbb{E}_{q_2(Z_2)q_3(Z_3)}[\log p(X,Z)] + \lambda - 1 \tag{9}$$

where $\lambda - 1$ is a additive constant that can be ignored when normalizing $q_1^*(Z_1)$. Thus we have shown that:

$$\log q_1^*(Z_1) = \mathbb{E}_{-Z_1}[\log p(X,Z)] \tag{10}$$

as required.

Practice/Implementation - D level (I have chosen 1.D.3)

1.2.4

The log likelihood of the data (D) is:

$$\log(P(D|\mu,\tau)) = \frac{N}{2}\log(\tau) - \frac{N}{2}\log(2\pi) - \frac{\tau}{2}\sum_{i=1}^{N}(x_i - \mu)^2 \tag{11}$$

The log prior for μ and τ is: (Note that $\mu|\tau \sim \mathcal{N}(\mu_0, (\lambda_0\tau)^{-1})$ and $\tau \sim \text{Gamma}(a_0, b_0)$)

$$\log(P(\mu,\tau)) = \frac{1}{2}\log(\lambda_0\tau) - \frac{1}{2}\log(2\pi) - \frac{\lambda_0\tau}{2}(\mu-\mu_0)^2 + a_0\log(b_0) - \log(\Gamma(a_0)) + (a_0-1)\log(\tau) - b_0\tau \tag{12}$$

The log-variational distribution is (Where $q(\mu) \sim \mathcal{N}(\mu_N, \lambda_N^{-1})$ and $q(\tau) \sim \text{Gamma}(a_N, b_N)$):

$$\log(q(\mu,\tau)) = \frac{1}{2}\log(\lambda_N) - \frac{1}{2}\log(2\pi) - \frac{\lambda_N}{2}(\mu-\mu_N)^2 + a_N\log(b_N) - \log(\Gamma(a_N)) + (a_N-1)\log(\tau) - b_N\tau \tag{13}$$

Finally we state the score functions of the variational distributions. Let

$$\omega = (\mu_N, \lambda_N, a_N, b_N)$$

be the variational parameters.

$$\nabla_\omega \log(q(\mu,\tau)) = \begin{bmatrix} \lambda_N(\mu - \mu_N) \\ \frac{1}{2\lambda_N} - \frac{1}{2}(\mu-\mu_N)^2 \\ \psi(a_N) - \log(b_N) + \log(\tau) \\ \frac{a_N}{b_N} - \tau \end{bmatrix} \tag{14}$$

where ψ is the digamma function. Note that we can also express the score function w.r.t λ_N as using the variance/standard deviation. We have that the relation between variance and precision is given by $\sigma^2 = \frac{1}{\lambda_N}$, meaning that $d\lambda = -\frac{2}{\sigma^3}d\sigma$. Then we have that the score function w.r.t σ (standard deviation) is given by

$$\frac{\partial \log q(\mu,\tau)}{\partial\sigma} = \frac{\partial \log q(\mu,\tau)}{\partial\lambda_N}\frac{\partial\lambda_N}{\partial\sigma} = \left(\frac{1}{2\lambda_N} - \frac{1}{2}(\mu-\mu_N)^2\right)\left(-\frac{2}{\sigma^3}\right) = -\frac{1}{\sigma} + \frac{(\mu-\mu_N)^2}{\sigma^3}$$

1.2.5

In this exercise we implement Algorithm 1 of the BBVI paper [Ranganath et al., 2014]. We reuse the data sampling script from 1.E.3 and prior parameters, meaning that

- $\mu_0 = 1.0$
- $\lambda_0 = 0.1$
- $a_0 = 1.0$
- $b_0 = 2.0$

The following plots show the ELBO over iterations and the expected value of μ and τ over iterations for dataset 2 with 100 samples.

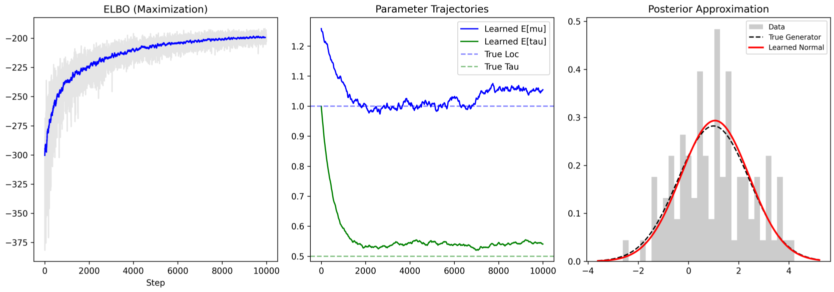

Figure 1: ELBO over iterations for dataset with 100 samples.

C-level

Theory 1.C.1

Question 2.1.10

For this exercise we want to show that that the IWELBO (Importance-Weighted ELBO) is a valid lower bound on the log marginal likelihood $\log p(X)$. First of, we state the IWELBO:

$$\mathcal{L}_K(q) := \mathbb{E}_{Z_1,\ldots,Z_K}\left[\log\left(\frac{1}{K}\sum_{k=1}^{K}\frac{p(X,Z_k)}{q(Z_k|X)}\right)\right] \tag{15}$$

Now, note that we can rewrite the marginal likelihood as:

$$p(X) = \int p(X,Z)dZ = \int q(Z|X)\frac{p(X,Z)}{q(Z|X)}dZ = \mathbb{E}_{q(Z|X)}\left[\frac{p(X,Z)}{q(Z|X)}\right]$$

and we can extent this to K samples:

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

Now, by applying Jensen's inequality that says that for a concave function f and random variable X, $\mathbb{E}[f(X)] \leq f(\mathbb{E}[X])$, (note that log is concave) we get:

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

Thus, the IWELBO is a valid lower bound on the log marginal likelihood $\log p(X)$, as required.

2.1.11

Let $W_K = \frac{p(X, Z_K)}{q(Z_K|X)}$. The IWELBO can then be rewritten as:

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

We can also see that the standard ELBO can be rewritten as:

$$\mathcal{L}_1(q) = \mathbb{E}_{Z_1}\left[\log\left(W_1\right)\right] = \mathbb{E}_{Z_1,\ldots,Z_K}\left[\frac{1}{K}\sum_{k=1}^{K}\log\left(W_k\right)\right]$$

This is because the W_k are i.i.d. Now we can see that

$$\log\left(\frac{1}{K}\sum_{k=1}^{K} W_k\right) \geq \frac{1}{K}\sum_{k=1}^{K}\log\left(W_k\right)$$

This follows from the fact that the logarithm is strictly concave and according to Jensens inequality the logarithm of an average is greater than the average of the logaritms. Taking expectations of both sides, we get the final expression:

$$\mathcal{L}_K(q) = \mathbb{E}_{Z_1,\ldots,Z_K}\left[\log\left(\frac{1}{K}\sum_{k=1}^{K} W_k\right)\right] \geq \mathbb{E}_{Z_1,\ldots,Z_K}\left[\frac{1}{K}\sum_{k=1}^{K}\log\left(W_k\right)\right] = \mathcal{L}_1(q)$$

Theory 1.C.2

Question 2.1.12

In this exercise we derive the Rao-Blackwellized partial gradient of the ELBO w.r.t λ_3. We have a total of $n+4$ latent variables, $v, z, y_n, \omega_1, \omega_2$. The variational distribution is given by:

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

The formula for the gradient of the ELBO w.r.t λ_3 is:

$$\nabla_{\lambda_3}\mathcal{L}(\lambda) = \mathbb{E}_{q(Z|X)}[\nabla_{\lambda_3}\log q_{\lambda_3}(z)(\log p_3(x,z) - \log q(z|\lambda_3))]$$

In this formula, as defined in the paper by Ranganath we have that $p_3(x,z)$ is the terms in the joint that depend on those variables. Meaning that

$$p_3(x,z) = p(x|z,y_n)p(z|v)$$

This gives the final expression for the Rao-Blackwellized partial gradient of the ELBO w.r.t λ_3:

$$\nabla_{\lambda_3}\mathcal{L}(\lambda) = \mathbb{E}_{q(Z|X)}[\nabla_{\lambda_3}\log q_{\lambda_3}(z)(\log p(x|z,y_n) + \log p(z|v) - \log q_{\lambda_3}(z))]$$

Practice/implementation - 1.C.3

Question 2.2.13

To implement BBVI with Control variates, but without Rao-Blackwellization, we reuse the code from 1.D.5 and add control variates. The implementation can be found in the notebook BBVI.ipynb. The results were as follows:

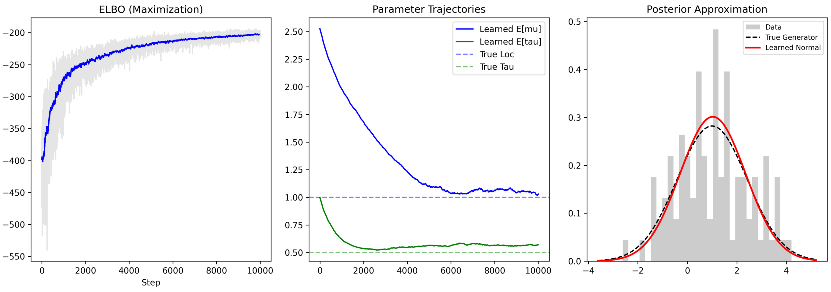

Figure 2: ELBO over iterations for dataset with 100 samples using BBVI with control variates.

Practice/implementation - 1.C.4

Question 2.2.14

The gamma distribution pdf is defined as

$$p(x|\alpha,\beta) = \frac{x^{\alpha-1}}{\Gamma(\alpha)\beta^{\alpha}}\exp(-x/\beta) = \exp((\alpha-1)log(x) - \frac{x}{\beta} - log(\Gamma(\alpha)) - \alpha log(\beta))$$

In, canonical exponential-family form we can use the following identifications:

- $\boldsymbol{\eta}(\boldsymbol{\theta}) = [\alpha - 1, -1/\beta]$
- $h(x) = 1$
- $\boldsymbol{T}(x) = [\log x, x]$
- $A(\boldsymbol{\eta}) = \log\Gamma(\eta_1 + 1) + (\eta_1 + 1)\log(-1/\eta_2) = \log\Gamma(\eta_1 + 1) - (\eta_1 + 1)\log(-\eta_2)$

With this we can identify

$$\nabla_{\boldsymbol{\eta}} A(\boldsymbol{\eta}) = \begin{bmatrix} \psi(\eta_1 + 1) - \log(-\eta_2) \\ -\frac{\eta_1+1}{\eta_2} \end{bmatrix}$$

Question 2.2.15

Let J be the Jacobian matrix of η w.r.t $\theta = (\alpha, \beta)$.

$$J = \begin{bmatrix} \frac{\partial \eta_1}{\partial \alpha} & \frac{\partial \eta_1}{\partial \beta} \\ \frac{\partial \eta_2}{\partial \alpha} & \frac{\partial \eta_2}{\partial \beta} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\beta^2} \end{bmatrix}$$

Furthermore, we have that $F(\eta)$ is the Fisher information matrix w.r.t the natural parameters η and $F(\theta) = J^T F(\eta) J$. According to the Question description, we do not need to specify the explicit entries in F.

Using this, we have that the natural gradient is given by:

$$\tilde{\nabla}_{\theta}\mathcal{L}(\theta) = F(\alpha,\beta)^{-1}\nabla_{\theta}\mathcal{L}(\theta) = J^T F(\eta)^{-1} J \nabla_{\theta}\mathcal{L}(\theta)$$

This follows from the fact that $J^{-1} = J$.

Question 2.2.16

Now we compute the fisher information matrix $F(\theta)$ explicitly. We have that $F(\eta) = \nabla_\eta \nabla_\eta^T A(\eta)$. Thus,

$$F(\eta) = \begin{bmatrix} \psi_1(\eta_1 + 1) & -\frac{1}{\eta_2} \\ -\frac{1}{\eta_2} & \frac{\eta_1+1}{\eta_2^2} \end{bmatrix}$$

where ψ_1 is the trigamma function. Now we can compute $F(\theta)$, and get the result:

$$F(\theta) = J^T F(\eta) J = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\beta^2} \end{bmatrix} \begin{bmatrix} \psi_1(\alpha) & \beta \\ \beta & \alpha\beta^2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\beta^2} \end{bmatrix} = \begin{bmatrix} \psi_1(\alpha) & \frac{1}{\beta} \\ \frac{1}{\beta} & \frac{\alpha}{\beta^2} \end{bmatrix}$$

Question 2.2.17

The average negative log likelihood is given by:

$$\mathcal{L}(\alpha, \beta) = -\frac{1}{N} \log p(x_{1:N}|\alpha, \beta) = -\frac{1}{N} \sum_{i=1}^{N} \log p(x_i|\alpha, \beta)$$

(All samples are i.i.d). Since we have samples from a Gamma distribution we can rewrite this as

$$\mathcal{L}(\alpha, \beta) = -\frac{1}{N} \sum_{n=1}^{N} ((\alpha - 1)\log x_n - \beta x_n - \log \Gamma(\alpha) + \alpha \log \beta)$$

The gradient of the negative log likelihood is then given by:

$$\nabla_{\alpha,\beta} \mathcal{L}(\alpha, \beta) = \begin{bmatrix} -\frac{1}{N} \sum_{n=1}^{N} (\log x_n - \psi(\alpha) + \log \beta) \\ -\frac{1}{N} \sum_{n=1}^{N} \left(-x_n + \frac{\alpha}{\beta}\right) \end{bmatrix}$$

Question 2.2.18

This question is primarily solved in a python notebook. We sample 1000 data points from $Gamma(\alpha^* = 3.0, \beta^* = 2.0)$. Using this data, we implement both gradient descent and natural gradient descent to estimate α and β. We initilize both methods with a poor guess of $(\alpha, \beta) = (0.5, 8)$ and ensure that α, β stay positive during optimizaiton. The code can be found in ngd_vs_gd_1D_gamma_AD.ipynb notebook.

Question 2.2.19

The results of running both gradient descent and natural gradient descent can be seen in the following plots.

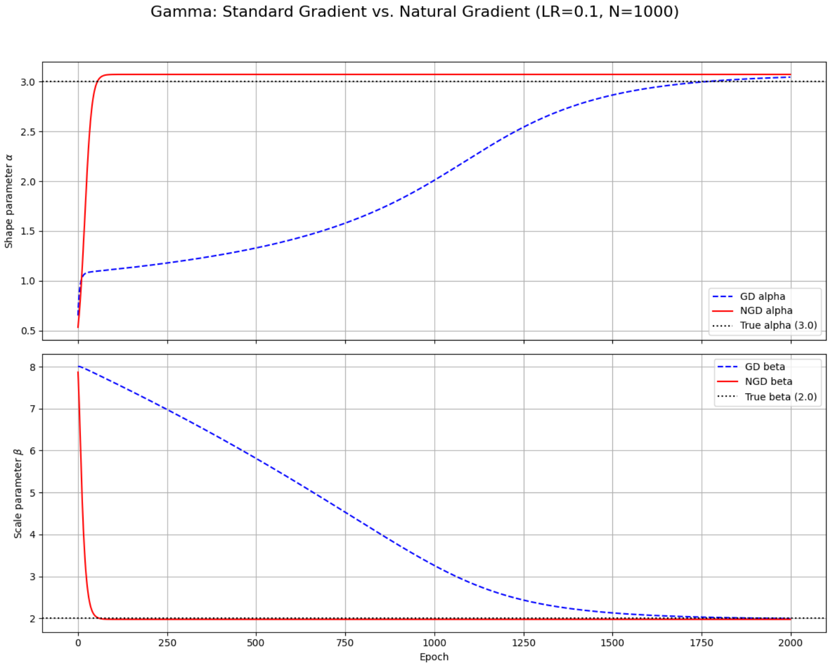

Figure 3: Gradient descent: Estimated α and β over iterations.

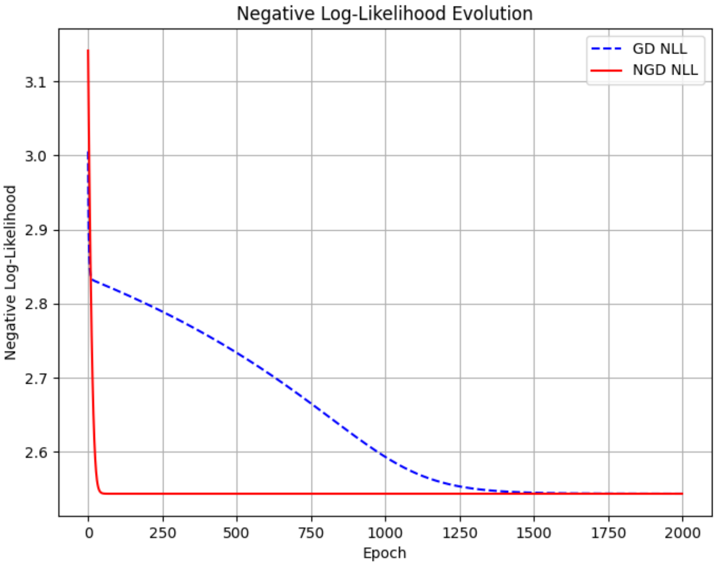

Figure 4: Natural gradient descent: Estimated α and β over iterations.

Based on the plots, we can see that natural gradient descent converges faster to the true parameters $(\alpha^*, \beta^*) = (3.0, 2.0)$ compared to standard gradient descent. Furthermore, natural gradient descent does not reach a good approximation, after 150 epochs but is in need of more epochs. This is because natural gradient descent takes into account the geometry of the parameter space, leading to more efficient updates and avoids getting 'stuck'.

B-level

1.B.1

Question 3.1.20

To approximate and reparameterize the categorical distribution we can use the Gumbel-Softmax distribution as an approximation. Let $g_i \sim \mathit{Gumbel}(0, 1)$ for $i = 1, ..., K$, probabilities $\pi_1, ..., \pi_K$ and temperature τ. A sample from the Gumbel-Softmax distribution is then given by:

$$y_i = \frac{\exp((\log(\pi_i) + g_i)/\tau)}{\sum_{j=1}^{K} \exp((\log(\pi_j) + g_j)/\tau)} \quad \text{for } i = 1, ..., K$$

This is the approximation of a one-hot encoded sample from a categorical distribution with class probabilities $\pi_1, ..., \pi_K$. As $\tau \to 0$, the Gumbel-Softmax distribution approaches the categorical distribution, and with $\tau \to \infty$, it approaches a uniform distribution. Note that the approximation is continuous and differentiable w.r.t the parameters π_i, allowing for gradient-based optimization.

For evaluation, we can use the argmax function to obtain a one-hot encoded sample from the Gumbel-Softmax distribution:

$$z = \text{one\_hot}(\text{argmax}_i(g_i + \log(\pi_i)))$$

The following code was implemented:

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

  dist = Gumbel(0,1)
  u = dist.sample((N, a.shape[0]))
  samples = F.softmax((torch.log(a + eps) + u) / temp, dim=1)

  return samples # make sure that your implementation allows the gradient to backpropagate

```

and the resulting output plot is shown below:

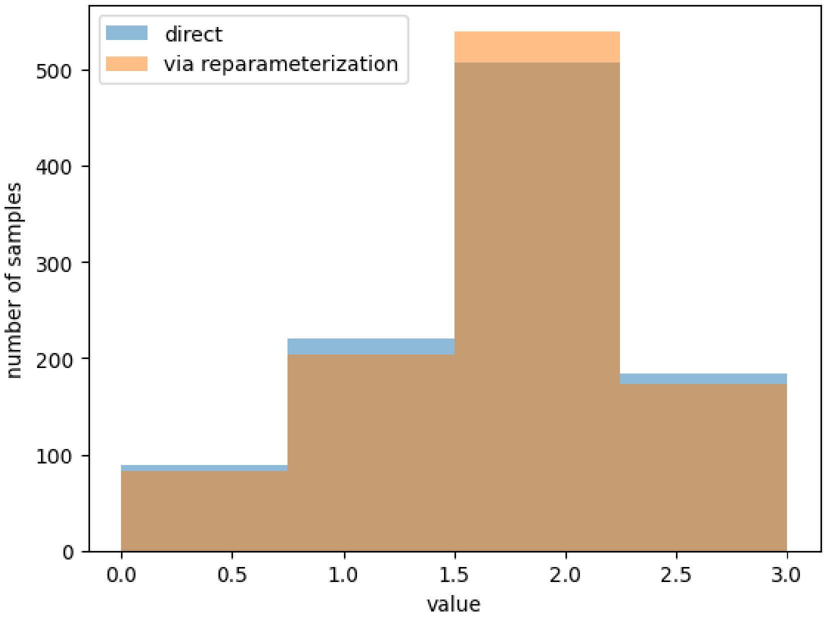

Figure 5: Output samples from the Gumbel-Softmax reparameterization.

1.B.2

Question 3.1.21

Using the reparameterization trick we can express a sample z from a parametric distribution $q(z)$ as a deterministic function of a random variable ϵ, with some fixed distribution and the parameters ϕ of $q_\phi(z)$, $(z = t(\epsilon, \phi))$. The paper gives the example of q_ϕ being a diagonal gaussian, and for $\epsilon \sim N(0, \mathbb{I})$, $z = \mu + \sigma\epsilon$ gives a sample from $q_\phi(z) = N(z|\mu, \sigma^2)$. Under such a parametrization of z we can decompose the total derivative of the integrand of the estimator, w.r.t trainable parameters ϕ as:

$$\begin{aligned} \hat{\nabla}_{TD}(\epsilon, \phi) = \nabla_\phi \left[\log p(\boldsymbol{x}|\boldsymbol{z}) + \log p(\boldsymbol{z}) - \log q_\phi(\boldsymbol{z}|\boldsymbol{x})\right] = \quad (16) \\ \nabla_\phi \left[\log p(\boldsymbol{z}|\boldsymbol{x}) + \log p(\boldsymbol{x}) - \log q_\phi(\boldsymbol{z}|\boldsymbol{x})\right] = \quad (17) \\ \nabla_{\boldsymbol{z}} \left[\log p(\boldsymbol{z}|\boldsymbol{x}) - \log q_\phi(\boldsymbol{z}|\boldsymbol{x})\right] \nabla_\phi t(\epsilon, \phi) - \nabla_\phi \log q_\phi(\boldsymbol{z}|\boldsymbol{x}) \quad (18) \end{aligned}$$

The reparameterized gradient estimator thus consists of two terms, the first is the path derivative and the second is the score function component.

Question 3.1.22

$$\mathbb{E}_{q_\phi(z|x)}[\nabla_\phi \log q_\phi(z|x)] = \int q_\phi(z|x) \nabla_\phi \log q_\phi(z|x) dz = \int \nabla_\phi q_\phi(z|x) dz = \nabla_\phi \int q_\phi(z|x) dz = \nabla_\phi 1 = 0$$

The expectation of the score function is zero because the integral of the probability density function over its entire support is equal to 1, and the gradient of a constant (1) w.r.t any parameter is zero.

Question 3.1.23

The authors propose that we can remove the score function term from the gradient estimate by setting ϕ' to the stop gradient.

Question 3.1.24

The authors of the paper bring up the concept that if the score function is positively correlated with the remaining terms in the total derivative estimator, then the variance of the estimator can be reduced by subtracting the score function term. (The score function then acts as a control variate)

Question 3.1.25

After extending the VAE implementation of 2E according to Algorithm 2, the following results were obtained for the ELBO over epochs on the MNIST dataset:

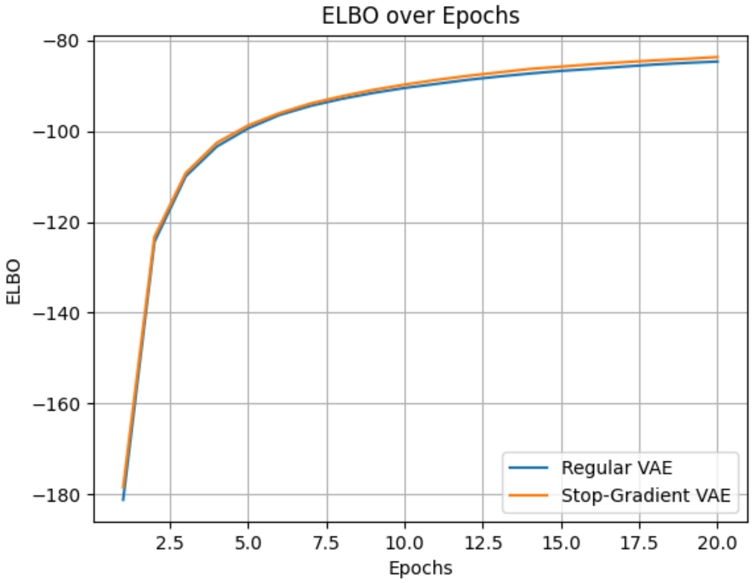

Figure 6: ELBO over epochs on MNIST using path derivative estimator.

We can see that both models preform similarly, with the path derivative estimator having a slight edge. This is expected as the score function term has an expectation of zero, meaning that removing it does not introduce bias. However, removing the score function term can reduce variance in the gradient estimates, leading to more stable and efficient training. The code used for implementing this can be found in the notebook 2E-VAE-HT25.ipynb

A-level

1.A.1 - Theory

Question 4.1.26

From the paper Denoising Diffusion Probabilistic Models we have that the left most side of equation 3 is

$$\mathbb{E}\left[-\log p_\theta(\mathbf{x}_0)\right] \leq \mathbb{E}_q\left[-\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right] := L$$

Where the left hand side is the expecation taken over the data distribution and the right hand side is the ELBO (expectation taken over whole mean field assumption q). Following the derivation in the paper we can rewrite this as:

$$\begin{aligned}L &= \mathbb{E}_q\left[-\log p(\mathbf{x}_T) - \sum_{t\geq 1}\log\frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1})}\right]\\
&= \mathbb{E}_q\left[-\log p(\mathbf{x}_T) - \sum_{t>1}\log\frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1})} - \log\frac{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)}\right]\\
&= \mathbb{E}_q\left[-\log p(\mathbf{x}_T) - \sum_{t>1}\log\frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)}\cdot\frac{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} - \log\frac{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)}\right]\\
&= \mathbb{E}_q\left[-\log\frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_0)} - \sum_{t>1}\log\frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)} - \log p_\theta(\mathbf{x}_0|\mathbf{x}_1)\right]\\
&= \mathbb{E}_q\left[D_{KL}(q(\mathbf{x}_T|\mathbf{x}_0)||p(\mathbf{x}_T)) + \sum_{t>1}D_{KL}(q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)||p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)) - \log p_\theta(\mathbf{x}_0|\mathbf{x}_1)\right]\end{aligned}$$

The expectations are all taken over the full distribution $q(\mathbf{x}_{0:T})$. The final step, to arrive at the final equation comes from separating the terms in the expectations.

$$L = \mathbb{E}_q\left[D_{KL}(q(\mathbf{x}_T|\mathbf{x}_0)||p(\mathbf{x}_T))\right] + \sum_{t>1}\mathbb{E}_q\left[D_{KL}(q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)||p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))\right] - \mathbb{E}_q\left[\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)\right]$$

In this final expression we can identify the three terms as L_T, L_{t-1} and L_0 respectively.

$$L_T = D_{KL}(q(\mathbf{x}_T|\mathbf{x}_0)||p(\mathbf{x}_T)) \rightarrow (1) = \mathbb{E}_q\left[L_T\right]$$

$$L_{t-1} = D_{KL}(q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)||p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)) \rightarrow (2) = \mathbb{E}_q\left[L_{t-1}\right]$$
$$L_0 = \log p_\theta(\mathbf{x}_0|\mathbf{x}_1) \rightarrow (3) = \mathbb{E}_q\left[L_0\right]$$

By inspecting the final expectations we find what distributions the expectations are taken over. We have that

$$(1) = \mathbb{E}_q\left[L_T\right] = \mathbb{E}_{q(\mathbf{x}_0)}\left[D_{KL}(q(\mathbf{x}_T|\mathbf{x}_0)||p(\mathbf{x}_T))\right]$$

Meaning that we only take the expectation over $\mathbf{x}_0$. This is true since the Kullback-leibler divergence integrates out $\mathbf{x}_T$.

$$(2) = \mathbb{E}_q\left[L_{t-1}\right] = \mathbb{E}_{q(\mathbf{x}_t,\mathbf{x}_0)}\left[D_{KL}(q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)||p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))\right]$$

Showing that we take the expectation over $\mathbf{x}_t$ and $\mathbf{x}_0$. This is once again true since the Kullback-leibler divergence integrates out $\mathbf{x}_{t-1}$. Finally, we have that

$$(3) = \mathbb{E}_q\left[L_0\right] = \mathbb{E}_{q(\mathbf{x}_1,\mathbf{x}_0)}\left[\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)\right]$$

Taking the expectation over $\mathbf{x}_1$ and $\mathbf{x}_0$. Thus we have specified the random variables of the expectations.

1.A.2 - Importance Weighted Autoencoder

Question 4.2.27

For this question we extend the VAE implementation from 2E to the IWAE model. To implement this, the loss function is modified to use the IWELBO with K samples. The modified loss function is as follows:

$$\mathcal{L}_K = \mathbb{E}_{Z_1,\ldots,Z_K}\left[\log\left(\frac{1}{K}\sum_{k=1}^{K}\frac{p_\theta(X,Z_k)}{q_\phi(Z_k|X)}\right)\right]$$

Let $W_k = \frac{p_\theta(X,Z_k)}{q_\phi(Z_k|X)}$. The loss function can then be rewritten as:

$$\mathcal{L}_K = \mathbb{E}_{Z_1,\ldots,Z_K}\left[\log\left(\frac{1}{K}\sum_{k=1}^{K} W_k\right)\right]$$

In practice, we approximate the expectation using Monte Carlo sampling.

The sampling also changes, and we now sample K latent variables. This is done by modifying the reparametrization function to sample K times from the latent distribution. The following code snippets show the modified parts of the VAE implementation: The reparametrization function:

```python
def reparameterization(self, mean, std, K):
    # set z = mean + std * epsilon, where epsilon ~ N(0, I)
    batch_size, latent_dim = mean.size()
    eps = torch.randn(batch_size, K, latent_dim).to(device)  # shape: (batch_size, K, latent_dim)
    z = mean.unsqueeze(1) + std.unsqueeze(1) * eps  # shape: (batch_size, K, latent_dim)
    return z
```

The loss function:

```python
def loss_IWAE(x, theta, mean, log_var, z):
    # theta: (batch, K, x_dim), z: (batch, K, latent_dim)
    #Expanded x to match theta's shape
    x_expanded = x.unsqueeze(1).expand_as(theta)
    #Calculate log p(x|z)
    log_px_z= -nn.functional.binary_cross_entropy(theta, x_expanded,
      reduction='none').sum(dim=2)

    # Parameters for calculating log probabilities
    log2pi = float(np.log(2 * np.pi))
    #Expand mean and log_var to match z's shape
    mean_exp = mean.unsqueeze(1)
    log_var_exp = log_var.unsqueeze(1)

    #prior p(z) ~ N(0, I)
    log_pz = -0.5 * (z.pow(2) + log2pi).sum(dim=2)

    #posterior q(z|x) ~ N(mean, var)
    log_qz_x = -0.5 * ((((z - mean_exp) ** 2) / log_var_exp.exp()) +
     log_var_exp + log2pi).sum(dim=2)

    log_w = log_px_z + log_pz - log_qz_x

    # Loss for each batch element
    log_w_mean = torch.logsumexp(log_w, dim=1) - np.log(log_w.size(1))

    #mean over batch:
    loss = -log_w_mean.mean()
    return loss
```

Question 4.2.28

Following the experimental setup from 2E (section 5.1) we use the 1 stochastic layer model on MNIST. We train the regular VAE model, together with the IWAE model with $K = 1$, $K = 5$ and $K = 50$. For feasible runtime, numer of epochs $= 20$ and learning rate $= 1e - 3$. The results for the ELBO over epochs are shown in the following plot:

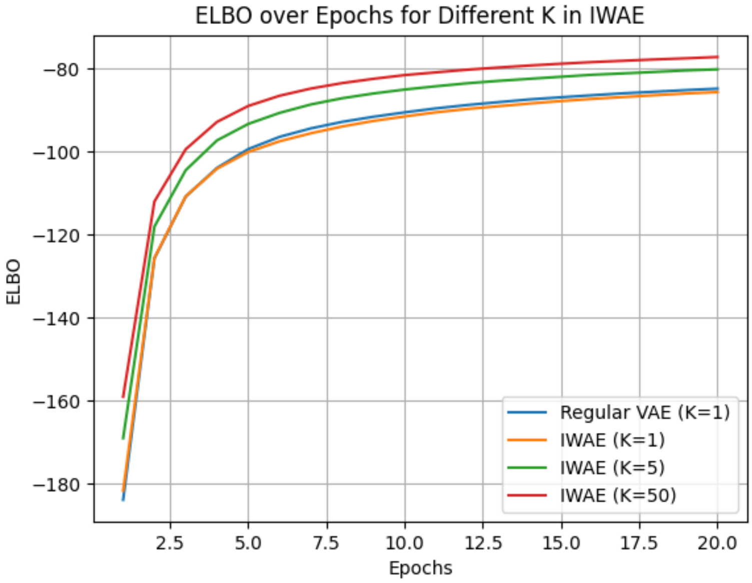

Figure 7: ELBO over epochs on MNIST using IWAE with different K values.

We can clearly see that increasing K leads to better ELBO values. This is inline with the theory, since the IWELBO is a tighter bound. For the model architecure, I left the encoder and decoder the same as in 2E, but created a new model for IWAE. The code for the model architecture is as follows:

```
class Encoder(nn.Module):
  # encoder outputs the parameters of variational distribution "q"
  def __init__(self, input_dim, hidden_dim, latent_dim):
      super(Encoder, self).__init__()

      self.FC_enc1 = nn.Linear(input_dim, hidden_dim) # FC = fully connected layer
      self.FC_enc2 = nn.Linear(hidden_dim, hidden_dim)
      self.FC_mean = nn.Linear(hidden_dim, latent_dim)
      self.FC_std  = nn.Linear (hidden_dim, latent_dim)

      self.LeakyReLU = nn.LeakyReLU(0.2)

      self.training = True

  def forward(self, x):
      h_1     = self.LeakyReLU(self.FC_enc1(x))
      h_2     = self.LeakyReLU(self.FC_enc2(h_1))
      mu     = self.FC_mean(h_2)  # mean / location
      log_var = self.FC_std(h_2)    # log variance

      return mu, log_var

class Decoder(nn.Module):
    # decoder generates the success parameter of each pixel
    def __init__(self, latent_dim, hidden_dim, output_dim):
        super(Decoder, self).__init__()
        self.FC_dec1   = nn.Linear(latent_dim, hidden_dim)
        self.FC_dec2   = nn.Linear(hidden_dim, hidden_dim)
        self.FC_output = nn.Linear(hidden_dim, output_dim)

        self.LeakyReLU = nn.LeakyReLU(0.2)

    def forward(self, z):
        h_out_1  = self.LeakyReLU(self.FC_dec1(z))
        h_out_2  = self.LeakyReLU(self.FC_dec2(h_out_1))

        theta = torch.sigmoid(self.FC_output(h_out_2))
        return theta

class Model(nn.Module):
    def __init__(self, Encoder, Decoder):
        super(Model, self).__init__()
        self.Encoder = Encoder
        self.Decoder = Decoder

    def reparameterization(self, mean, std):
        # set z = mean + std * epsilon, where epsilon ~ N(0, I)
        z = mean + std * torch.randn_like(std)
        return z

    def forward(self, x):
        mean, log_var = self.Encoder(x)
        std = torch.exp(0.5 * log_var)
        z = self.reparameterization(mean, std)
        theta = self.Decoder(z)
        return theta, mean, log_var, z

class Model_IWAE(nn.Module):
    def __init__(self, Encoder, Decoder):
        super(Model_IWAE, self).__init__()
        self.Encoder = Encoder
        self.Decoder = Decoder

    #Sample K times
    def reparameterization(self, mean, std, K):
        # set z = mean + std * epsilon, where epsilon ~ N(0, I)
        batch_size, latent_dim = mean.size()
        eps = torch.randn(batch_size, K, latent_dim).to(device)  # shape: (batch_size, K, la
        z = mean.unsqueeze(1) + std.unsqueeze(1) * eps  # shape: (batch_size, K, latent_dim
        return z

    def forward(self, x, K):
        mean, log_var = self.Encoder(x)
        std = torch.exp(0.5 * log_var)
        z = self.reparameterization(mean, std, K)  # shape: (batch_size, K, latent_dim)
        theta = self.Decoder(z.view(-1, z.size(-1)))  # reshape z to (batch_size * K, laten
        theta = theta.view(x.size(0), K, -1)  # reshape theta to (batch_size, K, x_dim)
        return theta, mean, log_var, z
```

Appendix

(For a better overview of the code, please refer to the respective notebooks and scripts)

Code - BBVI.ipynb

```
# %%
import tqdm
from scipy.special import digamma, gammaln
from scipy.stats import norm
import matplotlib.pyplot as plt
import numpy as np
import torch.distributions as dist
import torch.nn as nn
import torch
import os
# Force the threading layer to be compatible with Jupyter on macOS
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"
os.environ["OMP_NUM_THREADS"] = "1"


# %%
# Set random seeds for reproducibility
torch.manual_seed(0)
np.random.seed(0)

# %%


def generate_data(mu, tau, N):
    """Draw N samples from N(mu, tau^{-1}) and return them as a NumPy array.
    """
    if tau <= 0:
        raise ValueError("Precision tau must be positive.")

    std = np.sqrt(1.0 / tau)
    D = np.random.normal(loc=mu, scale=std, size=N)
    return D


# %%
mu_true = 1
tau_true = 0.5
Dataset_1 = generate_data(mu_true, tau_true, 10)
Dataset_2 = generate_data(mu_true, tau_true, 100)
Dataset_3 = generate_data(mu_true, tau_true, 1000)

# Make them into tensors:
dataset_1 = torch.tensor(Dataset_1, dtype=torch.float32)
dataset_2 = torch.tensor(Dataset_2, dtype=torch.float32)
dataset_3 = torch.tensor(Dataset_3, dtype=torch.float32)
```

```
# Visualize the datasets/true distribution:
# Plot histograms of the data:
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.hist(dataset_1, bins=10, alpha=0.7, color='blue')
plt.title('Histogram of 10 samples')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.subplot(1, 3, 2)
plt.hist(dataset_2, bins=20, alpha=0.7, color='green')
plt.title('Histogram of 100 samples')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.subplot(1, 3, 3)
plt.hist(dataset_3, bins=30, alpha=0.7, color='red')
plt.title('Histogram of 1000 samples')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.tight_layout()
plt.show()


# %%
class Normal_BBVI(nn.Module):
    def __init__(self, mu_0=1, lambda_0=0.1, a_0=1, b_0=2):
        super(Normal_BBVI, self).__init__()
        # Initialize mu and log_sigma as learnable parameters
        self.mu_0 = mu_0
        self.lambda_0 = lambda_0
        self.a_0 = a_0
        self.b_0 = b_0

        # Parameters for variatioalnal distributions:
        # q(mu) - Normal(loc,scale)
        self.q_mu_loc = nn.Parameter(torch.tensor(np.random.randn()))
        self.q_mu_scale_raw = nn.Parameter(torch.tensor((1.0)))

        # q(tau) - Gamma(concentration, rate)
        self.q_tau_concentration_raw = nn.Parameter(torch.tensor(1.0))
        self.q_tau_rate_raw = nn.Parameter(torch.tensor(1.0))

    @property
    def q_mu_scale(self):
        return torch.nn.functional.softplus(self.q_mu_scale_raw)

    @property
    def q_tau_conc(self):
        return torch.nn.functional.softplus(self.q_tau_concentration_raw)

    @property
```

```
def q_tau_rate(self):
    return torch.nn.functional.softplus(self.q_tau_rate_raw)

def sample_q(self, num_samples):
    q_mu = dist.Normal(self.q_mu_loc, self.q_mu_scale)
    q_tau = dist.Gamma(self.q_tau_conc, self.q_tau_rate)

    mu_samples = q_mu.rsample((num_samples,)).detach()
    tau_samples = q_tau.rsample((num_samples,)).detach()

    log_q_mu = q_mu.log_prob(mu_samples)
    log_q_tau = q_tau.log_prob(tau_samples)

    return mu_samples, tau_samples, log_q_mu, log_q_tau

def log_joint(self, x, mu, tau):
    """Compute the log joint probability log p(x, mu, tau).
    """
    # Get parameters:
    mu_0, lambda_0 = self.mu_0, self.lambda_0
    a_0, b_0 = self.a_0, self.b_0
    # Prior p(mu)
    mu_prior_scale = torch.sqrt(1.0 / (lambda_0 * tau))
    prior_mu = dist.Normal(mu_0, mu_prior_scale)
    log_p_mu = prior_mu.log_prob(mu)

    # Prior p(tau)
    prior_tau = dist.Gamma(a_0, b_0)
    log_p_tau = prior_tau.log_prob(tau)

    # Scale:
    scale = 1.0 / torch.sqrt(tau)

    # Reshape for broadcasting:
    mu_exp = mu.unsqueeze(1)  #
    scale_exp = scale.unsqueeze(1)  #
    x_exp = x.unsqueeze(0)  # Shape: (1, N)

    # Likelihood p(x | mu, tau)
    likelihood = dist.Normal(mu_exp, scale_exp)
    log_p_x_given_mu_tau = likelihood.log_prob(x_exp).sum(dim=1)

    return log_p_mu + log_p_tau + log_p_x_given_mu_tau

def compute_scores(self, mu_s, tau_s):
    """Compute the score functions for q(mu) and q(tau).
    """
    # Parameters:
    mu_loc = self.q_mu_loc.detach()
    mu_scale = self.q_mu_scale.detach()
    tau_conc = self.q_tau_conc.detach()
```

```
        tau_rate = self.q_tau_rate.detach()

        score_mu_wrt_loc = (mu_s - mu_loc) / (mu_scale ** 2)
        score_mu_wrt_scale = -1/mu_scale + \
            ((mu_s - mu_loc) ** 2) / (mu_scale ** 3)

        score_wrt_a = - digamma(tau_conc) + \
            torch.log(tau_rate) + torch.log(tau_s)
        score_wrt_b = tau_conc / tau_rate - tau_s

        return score_mu_wrt_loc, score_mu_wrt_scale, score_wrt_a, score_wrt_b


# %% [markdown]
# ### Naive BBVI Implementation:
#

# %%
def train_model(model, optimizer, x_data, steps=2500, n_mc_samples=50):
    history = {'ELBO': [], 'mu': [], 'tau': []}
    print(f"Training regular BBVI:")
    # Add so tqdm writes elbo:
    with tqdm.tqdm(total=steps, desc="Training") as pbar:
        for step in range(steps):
            optimizer.zero_grad()

            # 1. Sample and evaluuate q(mu) and q(tau)
            mu_samples, tau_samples, log_q_mu, log_q_tau = model.sample_q(
                n_mc_samples)
            log_q = log_q_mu + log_q_tau

            # 2. Compute Log P (Reward)
            with torch.no_grad():
                log_p = model.log_joint(x_data, mu_samples, tau_samples)

            # 3. Algorithm 1 Loss Naive REINFORCE
            rewards = log_p - log_q.detach()
            loss = - torch.mean(rewards * log_q)

            loss.backward()
            optimizer.step()

            # Track progress
            with torch.no_grad():
                elbo = torch.mean(log_p - log_q).item()
                e_mu = model.q_mu_loc.item()
                e_tau = (model.q_tau_conc / model.q_tau_rate).item()

                history['ELBO'].append(elbo)
                history['mu'].append(e_mu)
                history['tau'].append(e_tau)
            if step % 1000 == 0:
                pbar.set_postfix(
                    {'ELBO': elbo, 'E[mu]': e_mu, 'E[tau]': e_tau})
            pbar.update(1)
            # print(f"Step {step}: ELBO = {elbo:.4f}, E[mu] = {e_mu:.4f}, E[tau] = {e_tau:.
    return history


# %%
n_iter = 1e4
n_mc_samples = 50
learning_rate = 1e-3
all_histories = []
for dataset in [dataset_1, dataset_2, dataset_3]:
    model = Normal_BBVI()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    history = train_model(model, optimizer, dataset,
                          steps=int(n_iter), n_mc_samples=n_mc_samples)
    all_histories.append(history)


# %%
def plot_results(history, x_data, true_params, i=0, cv=False):
    true_loc_gen, true_tau_gen, = true_params

    plt.figure(figsize=(14, 5))

    # --- A. ELBO ---
    plt.subplot(1, 3, 1)
    elbo_ma = np.convolve(history['ELBO'], np.ones(50) / 50, mode='valid')
    plt.plot(history['ELBO'], alpha=0.2, color='gray')
    plt.plot(elbo_ma, color='blue')
    plt.title("ELBO (Maximization)")
    plt.xlabel("Step")

    # --- B. Trajectories ---
    plt.subplot(1, 3, 2)
    plt.plot(history['mu'], label='Learned E[mu]', color='blue')
    plt.plot(history['tau'], label='Learned E[tau]', color='green')

    plt.axhline(true_loc_gen, color='blue', linestyle='--',
                alpha=0.5, label='True Loc')
    plt.axhline(true_tau_gen, color='green',
                linestyle='--', alpha=0.5, label='True Tau')

    plt.title("Parameter Trajectories")
    plt.legend()

    # --- C. Distribution Fit ---
    plt.subplot(1, 3, 3)
    # Data Histogram
    plt.hist(x_data.numpy(), bins=30, density=True,
             alpha=0.4, color='gray', label='Data')

    x_range = np.linspace(x_data.min() - 1, x_data.max() + 1, 500)

    # True Generating PDF
    true_scale_gen = 1.0 / np.sqrt(true_tau_gen)
    pdf_true = norm.pdf(x_range, loc=true_loc_gen, scale=true_scale_gen)
    plt.plot(x_range, pdf_true, 'k--', label='True Generator')

    # Learned Model PDF (Skew Normal)
    final_mu = history['mu'][-1]
    final_tau = history['tau'][-1]
    final_scale = 1.0 / np.sqrt(final_tau)

    pdf_learned = norm.pdf(x_range, loc=final_mu, scale=final_scale)
    plt.plot(x_range, pdf_learned, 'r-', linewidth=2, label='Learned Normal')
    plt.title("Posterior Approximation")
    plt.legend(fontsize='small')

    plt.tight_layout()
    if i == 2:
        if not cv:
            # Save figure in Plots/
            plt.savefig(
                f'Plots/BBVI_Normal_Fit_{x_data.shape[0]}_samples.png', dpi=300)
        if cv:
            # Save figure in Plots/
            plt.savefig(
                f'Plots/BBVI_Normal_Fit_CV_{x_data.shape[0]}_samples.png', dpi=300)
    plt.show()


# %%
i = 1
for history, x_data in zip(all_histories, [dataset_1, dataset_2, dataset_3]):
    plot_results(history, x_data, (mu_true, tau_true), i=i)
    i += 1

# %% [markdown]
# ## BBVI with Control Variate (CV):
# ### first we create the training loop:

# %%


def train_model_cv_per_d(model: Normal_BBVI, optimizer, x_data, steps=2500, n_mc_samples=50
    history = {'ELBO': [], 'mu': [], 'tau': []}

    print(f"Training BBVI with Control Variates:")
```

```
# Add so tqdm writes elbo:

with tqdm.tqdm(total=steps, desc="Training") as pbar:
    for step in range(steps):
        optimizer.zero_grad()

        # 1. Sample and evaluuate q(mu) and q(tau)
        mu_samples, tau_samples, log_q_mu, log_q_tau = model.sample_q(
            n_mc_samples)

        # Compute Score functions:
        scores = model.compute_scores(mu_samples, tau_samples)
        log_q = torch.stack([log_q_mu, log_q_tau], dim=0)

        # 2. Compute Log P (Reward)
        with torch.no_grad():
            log_p = model.log_joint(x_data, mu_samples, tau_samples)

        # 3. Algorithm 2 (Control Variates) loss
        R_mu = log_p - log_q_mu.detach()
        R_tau = log_p - log_q_tau.detach()
        R_d_list = [R_mu, R_mu, R_tau, R_tau]

        f_d_list = []
        for d in range(4):
            f_d = R_d_list[d] * scores[d]
            f_d_list.append(f_d)

        # get h (score functions)
        h_d = []
        for d in range(4):
            h_d.append(scores[d])

        a_d_list = []
        for d in range(4):
            cat_fh_d = torch.cat(
                [f_d_list[d].unsqueeze(1), h_d[d].unsqueeze(1)], dim=1)
            a_d = torch.cov(cat_fh_d) / torch.var(h_d[d])
            a_d_list.append(a_d.detach())

        loss_list = []
        tot_loss = 0.0
        for d in range(4):
            loss_d = - \
                torch.mean(f_d_list[d] - torch.matmul(a_d_list[d], h_d[d]))
            loss_list.append(loss_d)
            tot_loss += loss_d

        model.q_mu_loc.grad = loss_list[0]
        model.q_mu_scale_raw.grad = loss_list[1]
        model.q_tau_concentration_raw.grad = loss_list[2]
```

```
            model.q_tau_rate_raw.grad = loss_list[3]

            optimizer.step()

            # Track progress
            with torch.no_grad():
                elbo = torch.mean(log_p - log_q).item()
                e_mu = model.q_mu_loc.item()
                e_tau = (model.q_tau_conc / model.q_tau_rate).item()

                history['ELBO'].append(elbo)
                history['mu'].append(e_mu)
                history['tau'].append(e_tau)
            if step % 1000 == 0:
                pbar.set_postfix(
                    {'ELBO': elbo, 'E[mu]': e_mu, 'E[tau]': e_tau})
            pbar.update(1)
    return history


# %%
n_iter = 1e4
n_mc_samples = 50
learning_rate = 1e-3
all_histories_cv = []
for dataset in [dataset_1, dataset_2, dataset_3]:
    model_cv = Normal_BBVI()
    optimizer_cv = torch.optim.Adam(model_cv.parameters(), lr=learning_rate)
    history_cv = train_model_cv_per_d(
        model_cv, optimizer_cv, dataset, steps=int(n_iter), n_mc_samples=n_mc_samples)
    all_histories_cv.append(history_cv)

# %%
# plot results for CV model
i = 1
for history, x_data in zip(all_histories_cv, [dataset_1, dataset_2, dataset_3]):
    plot_results(history, x_data, (mu_true, tau_true), i=i, cv=True)
    i += 1
```

Code - 2E-VAE-HT25.ipynb

```
# %% [markdown]
# # ***VAE for image generation***
# Consider VAE model from
# *Auto-Encoding Variational Bayes (2014, D.P. Kingma et. al.)*.
# We will implement a VAE model using Torch and apply it to the MNIST dataset.
#

# %% [markdown]
# ![MNIST_VAE.png](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAABAAAAAJJCAIAAAC7+NJKAAC+IE

# %% [markdown]
```

```
#
# **Generative model:** We model each pixel value $\in \{0,1\}$ as a sample drawn from a Be
#
# $ z_n \sim N(0,I) $
#
# $ \theta_n = g(z_n) $
#
# $ x_n \sim Bern(\theta_n) $
#
# where $g$ is the decoder. We choose the prior on $z_n$ to be the standard multivariate no
#
# **Inference model:** We infer the posterior distribution of $z_n$ via variational inferen
#
# $ q(z_n|x_n) \sim q( \mu_n, \sigma^2_n) $
#
# where $ \mu_n, \sigma^2_n = f(x_n)  $ and $f$ is the encoder.
#
#
#
#
#

# %% [markdown]
# **Implementation:**
# Let's start with importing Torch and other necessary libraries:

# %%
import matplotlib.pyplot as plt
from torch.optim import Adam
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
from torchvision.datasets import MNIST
import torch
import torch.nn as nn
import torch.distributions as dist

import numpy as np

from tqdm import tqdm

# %%
# Do not change the seeds
torch.manual_seed(0)
np.random.seed(0)

# %%
if torch.cuda.is_available():
    device = torch.device("cuda:0")
elif torch.backends.mps.is_available():
    device = torch.device("mps")
else:
    device = torch.device("cpu")
print(f"Using device: {device}")

# %% [markdown]
# ### ***Step1: Model Hyperparameters***
#
#

# %%
dataset_path = '~/datasets'

batch_size = 128

# Dimensions of the input, the hidden layer, and the latent space.
x_dim = 784
hidden_dim = 200
latent_dim = 20

# Learning rate
lr = 1e-3

# Number of epoch
epochs = 20

# %% [markdown]
# ### ***Step2: Load Dataset***
#
#

# %%

mnist_transform = transforms.Compose([
    transforms.ToTensor(),
])


train_dataset = MNIST(
    dataset_path, transform=mnist_transform, train=True, download=True)
test_dataset = MNIST(dataset_path, transform=mnist_transform,
                     train=False, download=True)
test_labels = test_dataset.targets

train_loader = DataLoader(dataset=train_dataset,
                          batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset,
                         batch_size=batch_size, shuffle=False)

# %% [markdown]
# ### ***Step3: Define the model***
#
```

```
# %%


class Encoder(nn.Module):
    # encoder outputs the parameters of variational distribution "q"
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(Encoder, self).__init__()

        # FC = fully connected layer
        self.FC_enc1 = nn.Linear(input_dim, hidden_dim)
        self.FC_enc2 = nn.Linear(hidden_dim, hidden_dim)
        self.FC_mean = nn.Linear(hidden_dim, latent_dim)
        self.FC_std = nn.Linear(hidden_dim, latent_dim)

        self.LeakyReLU = nn.LeakyReLU(0.2)

        self.training = True

    def forward(self, x):
        h_1 = self.LeakyReLU(self.FC_enc1(x))
        h_2 = self.LeakyReLU(self.FC_enc2(h_1))
        mu = self.FC_mean(h_2)  # mean / location
        log_var = self.FC_std(h_2)    # log variance

        return mu, log_var

# %%


class Decoder(nn.Module):
    # decoder generates the success parameter of each pixel
    def __init__(self, latent_dim, hidden_dim, output_dim):
        super(Decoder, self).__init__()
        self.FC_dec1 = nn.Linear(latent_dim, hidden_dim)
        self.FC_dec2 = nn.Linear(hidden_dim, hidden_dim)
        self.FC_output = nn.Linear(hidden_dim, output_dim)

        self.LeakyReLU = nn.LeakyReLU(0.2)

    def forward(self, z):
        h_out_1 = self.LeakyReLU(self.FC_dec1(z))
        h_out_2 = self.LeakyReLU(self.FC_dec2(h_out_1))

        theta = torch.sigmoid(self.FC_output(h_out_2))
        return theta


# %% [markdown]
# **Q3.1 (2 points)**  Below implement the reparameterization function.

# %%
```

```
class Model(nn.Module):
    def __init__(self, Encoder, Decoder):
        super(Model, self).__init__()
        self.Encoder = Encoder
        self.Decoder = Decoder

    def reparameterization(self, mean, std):
        # set z = mean + std * epsilon, where epsilon ~ N(0, I)
        z = mean + std * torch.randn_like(std)
        return z

    def forward(self, x):
        mean, log_var = self.Encoder(x)
        std = torch.exp(0.5 * log_var)
        z = self.reparameterization(mean, std)
        theta = self.Decoder(z)
        return theta, mean, log_var, z


class Model_IWAE(nn.Module):
    def __init__(self, Encoder, Decoder):
        super(Model_IWAE, self).__init__()
        self.Encoder = Encoder
        self.Decoder = Decoder

    # For IWAE, we need to sample K latent variables per input data point
    def reparameterization(self, mean, std, K):
        # set z = mean + std * epsilon, where epsilon ~ N(0, I)
        batch_size, latent_dim = mean.size()
        eps = torch.randn(batch_size, K, latent_dim).to(
            device)  # shape: (batch_size, K, latent_dim)
        # shape: (batch_size, K, latent_dim)
        z = mean.unsqueeze(1) + std.unsqueeze(1) * eps
        return z

    def forward(self, x, K):
        mean, log_var = self.Encoder(x)
        std = torch.exp(0.5 * log_var)
        # shape: (batch_size, K, latent_dim)
        z = self.reparameterization(mean, std, K)
        # reshape z to (batch_size * K, latent_dim)
        theta = self.Decoder(z.view(-1, z.size(-1)))
        # reshape theta to (batch_size, K, x_dim)
        theta = theta.view(x.size(0), K, -1)
        return theta, mean, log_var, z

# %% [markdown]
# ### ***Step4: Model initialization***
#
```

```
# %%
encoder = Encoder(input_dim=x_dim, hidden_dim=hidden_dim,
                  latent_dim=latent_dim)
decoder = Decoder(latent_dim=latent_dim,
                  hidden_dim=hidden_dim, output_dim=x_dim)

encoder2 = Encoder(input_dim=x_dim, hidden_dim=hidden_dim,
                   latent_dim=latent_dim)
decoder2 = Decoder(latent_dim=latent_dim,
                   hidden_dim=hidden_dim, output_dim=x_dim)

model = Model(Encoder=encoder, Decoder=decoder)
model2 = Model(Encoder=encoder2, Decoder=decoder2)
# model_IWAE = Model_IWAE(Encoder=encoder, Decoder=decoder)

model.to(device)
model2.to(device)
# model_IWAE.to(device)

# %% [markdown]
# ### ***Step5: Loss function and optimizer***
#

# %% [markdown]
# Our objective function is ELBO:
# $E_{q(z)}\big[\log \frac{p(x,z)}{q(z)}\big]$
#
# * **Q5.1 (1 point)** Show that ELBO can be rewritten as :
#
# $E_{q(z)}\big[\log p(x|z)\big] - D_{KL} \big( q(z) \lvert \rvert  p(z)\big)$
#

# %% [markdown]
# *5.1 Your answer*
#
# We can begin by inspecting and expanding the ELBO: $E_{q(z)}\big[\log \frac{p(x,z)}{q(z)}
#
# Now we can combine the terms with $p(z)$ and $q(z)$ to see that
# $E_{q(z)}\big[\log \frac{p(x,z)}{q(z)}\big] =  E_{q(z)}\big[\log p(x|z)\big] - E_{q(z)}\b
#
# Showing the wanted result
#
#

# %% [markdown]
# Consider the first term: $E_{q(z|x)}\big(\log p(x|z)\big)$
#
# $E_{q(z|x)}\big(\log p(x|z)\big) = \int q(z|x) \log p(x|z) dz $
#
# We can approximate this integral by Monte Carlo integration as following:
#
```

```
# $\approx \frac{1}{L} \sum_{l=1}^L \log p(x|z_l) $, where $z_l \sim q(z|x)$.
#
# Now we can compute this term using the analytic expression for $p(x|z)$. ( Remember we mo

# %% [markdown]
# Consider the second term: $- D_{KL} \big( q(z|x) \lvert \rvert  p(z)\big)$
#
# * **Q5.2 (2 points)** Kullback-Leibler divergence can be computed using the closed-form a

# %% [markdown]
#   *5.2 Solution:*
# Let $q(z) = \mathcal{N}(\mu_q, \Sigma_q)$ and the prior be $p(z) = \mathcal{N}(\mu_p, \Si
# The KL divergence is then
# $
# D_{KL}(q\|p)= \frac{1}{2}\left[\log \frac{|\Sigma_p|}{|\Sigma_q|}- K+ \mathrm{tr}\!\left(
#
# $D_{\mathrm{KL}}(q\|p)=\frac{1}{2}\sum_{k=1}^K\left[\log\frac{\sigma_{p,k}^2}{\sigma_{q,k
# +\frac{(\mu_{q,k}-\mu_{p,k})^2}{\sigma_{p,k}^2}\right]$.

# %% [markdown]
# **Q5.3 (2 points)** Now use your findings to implement the loss function, which is the ne

# %%


# should return the loss function (- ELBO)
def loss_function(x, theta, mean, log_var, z):
    # E_q[log(p(x|z))]:
    BCE = nn.functional.binary_cross_entropy(theta, x, reduction='sum')
    # KL divergence D_KL(q(z|x) || p(z)):
    KLD = -0.5 * torch.sum(1 + log_var - mean.pow(2) - log_var.exp())
    # sum over batch
    loss = BCE + KLD
    loss = loss / x.size(0)  # mean over batch
    return loss


def loss_function2(x, theta, mean, log_var, z):
    """
    Path-derivative / 'sticking the landing' loss:
    maximize E_q[ log p(x,z) - log q(z|x) ],
    but with stop-gradient on the parameters in log q.
    """
    #

    batch_size = x.size(0)
    log2pi = torch.log(torch.tensor(2.0 * np.pi, device=x.device))

    # --- log p(x|z) term (use decoder output theta as Bernoulli probs) ---
    # BCE = -log p(x|z)
    BCE = nn.functional.binary_cross_entropy(theta, x, reduction='sum')
    log_p_x_given_z = -BCE

    # --- log p(z) term (standard normal prior) ---
    log_pz = -0.5 * torch.sum(z.pow(2) + log2pi)

    # --- log q(z|x) term with STOP-GRADIENT on encoder params ---
    mean_det = mean.detach()
    log_var_det = log_var.detach()
    var_det = log_var_det.exp()

    log_qz_x = -0.5 * torch.sum(
        (z - mean_det).pow(2) / var_det + log_var_det + log2pi
    )

    # ELBO = E[ log p(x,z) - log q(z|x) ]
    elbo = log_p_x_given_z + log_pz - log_qz_x

    # We minimize -ELBO
    loss = -elbo / batch_size
    return loss


def loss_IWAE(x, theta, mean, log_var, z):
    # theta: (batch, K, x_dim), z: (batch, K, latent_dim)
    # Expanded x to match theta's shape
    x_expanded = x.unsqueeze(1).expand_as(theta)
    # Calculate log p(x|z)
    log_px_z = -nn.functional.binary_cross_entropy(theta,
                                                   x_expanded, reduction='none').sum(dim=2)

    # Parameters for calculating log probabilities
    log2pi = float(np.log(2 * np.pi))
    # Expand mean and log_var to match z's shape
    mean_exp = mean.unsqueeze(1)
    log_var_exp = log_var.unsqueeze(1)

    # prior p(z) ~ N(0, I)
    log_pz = -0.5 * (z.pow(2) + log2pi).sum(dim=2)

    # posterior q(z|x) ~ N(mean, var)
    log_qz_x = -0.5 * ((((z - mean_exp) ** 2) / log_var_exp.exp())
                       + log_var_exp + log2pi).sum(dim=2)

    log_w = log_px_z + log_pz - log_qz_x

    # Loss for each batch element
    log_w_mean = (torch.logsumexp(log_w, dim=1)
                  - np.log(log_w.size(1)))

    # mean over batch:
    loss = -log_w_mean.mean()
    return loss


# %% [markdown]
# ### ***Step6: Train the model***
#
# **Q6.1 (1 points)** Two lines of codes are missing in the training loop below, one to prop

# %%


def train_model(model, loss_function=loss_function):
    print("Start training VAE...")
    model.train()

    # optimizer
    optimizer = Adam(model.parameters(), lr=lr)
    pbar = tqdm(range(epochs))
    elbo = []
    for epoch in pbar:
        total_loss = 0
        total_samples = 0
        for batch_idx, (x, _) in enumerate(train_loader):
            x = x.to(device)
            x = x.view(-1, x_dim)
            x = torch.round(x)

            optimizer.zero_grad()

            theta, mean, log_var, z = model(x)
            loss = loss_function(x, theta, mean, log_var, z)

            loss.backward()
            optimizer.step()

            # loss.item() is the mean. Multiply by batch size to get the sum.
            total_loss += loss.item() * x.size(0)
            total_samples += x.size(0)

        # Correct global average
        avg_loss = total_loss / total_samples

        pbar.set_description(f"Epoch {epoch+1}/{epochs},"
                             f" Loss: {avg_loss:.4f},"
                             f" ELBO: {-avg_loss:.4f}")
        elbo.append(-avg_loss)

    print("Finish!!")
    return elbo
```

```
def train_model_IWAE(model, K=5):
    print("Start training IWAE...")
    model.train()

    # optimizer
    optimizer = Adam(model.parameters(), lr=lr)
    pbar = tqdm(range(epochs))
    elbo = []
    for epoch in pbar:
        total_loss = 0
        total_samples = 0
        for batch_idx, (x, _) in enumerate(train_loader):
            x = x.to(device)
            x = x.view(-1, x_dim)
            x = torch.round(x)

            optimizer.zero_grad()

            theta, mean, log_var, z = model(x, K)
            loss = loss_IWAE(x, theta, mean, log_var, z)

            loss.backward()
            optimizer.step()

            # loss.item() is the mean. Multiply by batch size to get the sum.
            total_loss += loss.item() * x.size(0)
            total_samples += x.size(0)

        # Correct global average
        avg_loss = total_loss / total_samples

        pbar.set_description(f"Epoch {epoch+1}/{epochs},"
                             f" Loss: {avg_loss:.4f},"
                             f" ELBO: {-avg_loss:.4f}")
        elbo.append(-avg_loss)

    print("Finish!!")
    return elbo

# %%


elbo_stop_gradient_VAE = train_model(model2, loss_function=loss_function2)
elbo_regular_VAE = train_model(model)
# IWAE_ELBO = train_model_IWAE(model_IWAE)


# %%
# Plot ELBO curve
plt.plot(range(1, epochs + 1), elbo_regular_VAE, label='Regular VAE')
plt.plot(range(1, epochs + 1), elbo_stop_gradient_VAE,
         label='Stop-Gradient VAE')
# plt.plot(range(1, epochs + 1), IWAE_ELBO, label='IWAE')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('ELBO')
plt.title('ELBO over Epochs')
plt.grid(True)
plt.savefig('Plots/Stop_gradient_VAE.png')
plt.show()

# %% [markdown]
# ### ***Step7: Generate images from test dataset***
# With our model trained, now we can start generating images.
#
# First, we will generate images from the latent representations of test data.
#
# Basically, we will sample $z$ from $q(z|x)$ and give it to the generative model (i.e., de
#
# **Q7.1 (1 points)** Fill in the script below to get the latent representations of each ba

# %%
model = model2  # Change this to select which model to evaluate

K = 5
model.eval()
# below we get decoder outputs for test data
with torch.no_grad():
    z_test = []
    x_test = []
    for batch_idx, (x, _) in enumerate(tqdm(test_loader)):
        x_test.append(x)
        x = x.to(device)
        x = x.view(-1, x_dim)
        x = torch.round(x)

        theta, mean, log_var, z = model(x)
        z_test.append(z.cpu().detach().numpy())

        # decode

    # Save the last batch theta for visualization
    theta_batch = model.Decoder(z)


# %%
theta_batch.shape
# keep mean over K
# theta_batch = theta_batch.mean(dim=1)

# %% [markdown]
# A helper function to display images:
```

```
# %%


def compare_images(x, theta, idx):
    # Reshape inputs to (Batch, 28, 28)
    # .cpu().detach().numpy() ensures we can handle tensors on GPU/with gradients
    x_img = x.view(-1, 28, 28)
    theta_img = theta.view(-1, 28, 28)

    # Create a figure with 1 row and 2 columns
    fig, axes = plt.subplots(1, 2, figsize=(8, 4))

    # Plot Original
    axes[0].imshow(x_img[idx].cpu().detach().numpy(), cmap='gray')
    axes[0].set_title("Original Image")
    axes[0].axis('off')

    # Plot Reconstruction
    axes[1].imshow(theta_img[idx].cpu().detach().numpy(), cmap='gray')
    axes[1].set_title("Reconstructed Image")
    axes[1].axis('off')

    plt.show()


# Call the function
theta_batch = theta_batch.cpu()
# theta_batch is the output of the decoder for the last batch in the test set
compare_images(x_test[-1], theta_batch, idx=0)
compare_images(x_test[-1], theta_batch, idx=1)
compare_images(x_test[-1], theta_batch, idx=2)
compare_images(x_test[-1], theta_batch, idx=3)
compare_images(x_test[-1], theta_batch, idx=4)
compare_images(x_test[-1], theta_batch, idx=5)
compare_images(x_test[-1], theta_batch, idx=6)
compare_images(x_test[-1], theta_batch, idx=7)
compare_images(x_test[-1], theta_batch, idx=8)
compare_images(x_test[-1], theta_batch, idx=9)

# %%
# run for k = 1, 5, 10
# First run regular VAE (k=1)
encoder = Encoder(input_dim=x_dim, hidden_dim=hidden_dim,
                  latent_dim=latent_dim)
decoder = Decoder(latent_dim=latent_dim,
                  hidden_dim=hidden_dim, output_dim=x_dim)

encoder1 = Encoder(input_dim=x_dim, hidden_dim=hidden_dim,
                   latent_dim=latent_dim)
decoder1 = Decoder(latent_dim=latent_dim,
                   hidden_dim=hidden_dim, output_dim=x_dim)

encoder5 = Encoder(input_dim=x_dim, hidden_dim=hidden_dim,
                   latent_dim=latent_dim)
decoder5 = Decoder(latent_dim=latent_dim,
                   hidden_dim=hidden_dim, output_dim=x_dim)

encoder50 = Encoder(input_dim=x_dim, hidden_dim=hidden_dim,
                    latent_dim=latent_dim)
decoder50 = Decoder(latent_dim=latent_dim,
                    hidden_dim=hidden_dim, output_dim=x_dim)

model = Model(Encoder=encoder, Decoder=decoder)
model_IWAE_1 = Model_IWAE(Encoder=encoder1, Decoder=decoder1)
model_IWAE_5 = Model_IWAE(Encoder=encoder5, Decoder=decoder5)
model_IWAE_50 = Model_IWAE(Encoder=encoder50, Decoder=decoder50)

model.to(device)
model_IWAE_1.to(device)
model_IWAE_5.to(device)
model_IWAE_50.to(device)

elbo_regular_VAE = train_model(model)
elbo_IWAE_1 = train_model_IWAE(model_IWAE_1, K=1)
elbo_IWAE_k5 = train_model_IWAE(model_IWAE_5, K=5)
elbo_IWAE_k50 = train_model_IWAE(model_IWAE_50, K=50)

# %%
# plot ELBO curves
plt.plot(range(1, epochs + 1), elbo_regular_VAE, label='Regular VAE (K=1)')
plt.plot(range(1, epochs + 1), elbo_IWAE_1, label='IWAE (K=1)')
plt.plot(range(1, epochs + 1), elbo_IWAE_k5, label='IWAE (K=5)')
plt.plot(range(1, epochs + 1), elbo_IWAE_k50, label='IWAE (K=50)')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('ELBO')
plt.title('ELBO over Epochs for Different K in IWAE')
plt.grid(True)
plt.savefig('Plots/IWAE_different_K.png')
plt.show()
```

Code - ngd_vs_gd_1D_gamma_AD.ipynb

```
# %% [markdown]
# # Q 2.4.17. Standard Gradient Descent vs. Natural Gradient Descent
# In this notebbook, we compare standard gradient descent (GD) and natural gradient descent

# %% [markdown]
# # 1. Configuration
# We define the true parameters of the Gamma distribution, the number of data points to gen

# %%
```

```python
import torch
import numpy as np
import matplotlib.pyplot as plt
from torch.distributions import Gamma
# ------------------------

# True parameters of the Gamma distribution we want to discover
ALPHA_TRUE = 3.0   # shape
BETA_TRUE = 2.0   # scale

# Number of observed data points
N_DATA = 1000

# Optimization parameters
LEARNING_RATE = 0.1
EPOCHS = 2000

# Initial "wrong" guess for our parameters (still positive)
ALPHA_INIT = 0.5
BETA_INIT = 8.0

# Fix random seed for reproducibility (optional)
torch.manual_seed(0)


# %% [markdown]
# # 2. Data Generation
# We generate synthetic data from the true Gamma distribution using PyTorch's `Gamma` distr

# %%
# Our parameterisation is shape-scale, but PyTorch's Gamma uses shape-rate.
rate_true = 1.0 / BETA_TRUE
dist_true = Gamma(concentration=torch.tensor(ALPHA_TRUE),
                  rate=torch.tensor(rate_true))

data = dist_true.sample((N_DATA,))

print(
    f"Generated {N_DATA} data points from Gamma(alpha={ALPHA_TRUE}, beta={BETA_TRUE})")
print(f"Sample mean:     {data.mean().item():.4f}")
print(f"Sample variance: {data.var().item():.4f}\n")


# %% [markdown]
# # 3. Loss function & parameter init
# We define the negative log-likelihood loss function for the Gamma distribution. We also i

# %%
# ToDo: Loss function
def gamma_nll(alpha, beta, data_points):
    """
    ToDo:
        Implement the average negative log-likelihood for Gamma distribution with shape=alp

        Hints:
        - Enforce positivity using clamp (e.g. min=1e-4).
        - PyTorch's Gamma takes (concentration=alpha, rate=1/beta).
        - Return the *mean* negative log-likelihood.

    """
    alpha = torch.clamp(alpha, min=1e-4)
    beta = torch.clamp(beta, min=1e-4)
    rate = 1.0 / beta
    dist = Gamma(concentration=alpha, rate=rate)
    nll = -dist.log_prob(data_points).mean()
    return nll


# %%

# Parameters for Standard Gradient Descent (GD)
alpha_gd = torch.tensor(ALPHA_INIT, requires_grad=True)
beta_gd = torch.tensor(BETA_INIT,  requires_grad=True)

# Parameters for Natural Gradient Descent (NGD)
alpha_ngd = torch.tensor(ALPHA_INIT, requires_grad=True)
beta_ngd = torch.tensor(BETA_INIT,  requires_grad=True)

# History trackers
history_gd = []
history_ngd = []

# %% [markdown]
# # 4. Fisher Information inverse

# %%


def fisher_inverse(alpha, beta):
    """
    TODO:
      Implement the inverse Fisher Information matrix F^{-1}(, )
      for the Gamma(shape=, scale=) distribution.

      Theory:
        F(, ) =
            [ 1()        1/      ]
            [  1/     /^2        ]

        F^{-1}(, ) =
            1 / ( 1() - 1) *
            [             -             ]
            [  -       ˜2 1()         ]

      Hints:
      - Use torch.polygamma(1, alpha) for 1() (trigamma).
      - Make sure to detach alpha, beta so F^{-1} is not part of the graph.
    """
    # complete
    factor = 1.0 / (alpha * torch.polygamma(1, alpha) - 1)
    inv11 = (factor * alpha).detach()
    inv12 = (factor * -beta).detach()
    inv22 = (factor * beta**2 * torch.polygamma(1, alpha)).detach()
    return inv11, inv12, inv22

# %% [markdown]
# # 4. Optimization Loop
# We run the optimization loop for a specified number of epochs. In each epoch, we perform 


# %%
print(f"Optimizing with LR={LEARNING_RATE} for {EPOCHS} epochs...")

for epoch in range(EPOCHS):

    # ===== A. Standard Gradient Descent (GD) =====

    if alpha_gd.grad is not None:
        alpha_gd.grad.zero_()
    if beta_gd.grad is not None:
        beta_gd.grad.zero_()

    loss_gd = gamma_nll(alpha_gd, beta_gd, data)
    loss_gd.backward()

    with torch.no_grad():
        alpha_gd -= LEARNING_RATE * alpha_gd.grad
        beta_gd -= LEARNING_RATE * beta_gd.grad

        alpha_gd.clamp_(min=1e-4)
        beta_gd.clamp_(min=1e-4)

    history_gd.append((alpha_gd.item(), beta_gd.item()))

    # ===== B. Natural Gradient Descent (NGD) =====

    if alpha_ngd.grad is not None:
        alpha_ngd.grad.zero_()
    if beta_ngd.grad is not None:
        beta_ngd.grad.zero_()

    loss_ngd = gamma_nll(alpha_ngd, beta_ngd, data)
    loss_ngd.backward()
    g_alpha = alpha_ngd.grad
    g_beta = beta_ngd.grad

    # ToDo : compute natural gradient using F^{-1}(, )
    #  1) Get F^{-1} entries using fisher_inverse(...)
    #  2) Compute:
    #       - ng_alpha
    #       - ng_beta

    inv11, inv12, inv22 = fisher_inverse(alpha_ngd, beta_ngd)

    ng_alpha = inv11 * g_alpha + inv12 * g_beta
    ng_beta = inv12 * g_alpha + inv22 * g_beta

    with torch.no_grad():
        alpha_ngd -= LEARNING_RATE * ng_alpha
        beta_ngd -= LEARNING_RATE * ng_beta

        alpha_ngd.clamp_(min=1e-4)
        beta_ngd.clamp_(min=1e-4)

    history_ngd.append((alpha_ngd.item(), beta_ngd.item()))

    if (epoch + 1) % 15 == 0 or epoch == 0:
        print(f"\n--- Epoch {epoch + 1} ---")
        print(f"  GD:  alpha={alpha_gd.item():.4f}, beta={beta_gd.item():.4f}, "
              f"Loss={loss_gd.item():.4f}")
        print(f"  NGD: alpha={alpha_ngd.item():.4f}, beta={beta_ngd.item():.4f}, "
              f"Loss={loss_ngd.item():.4f}")


print("\nOptimization finished.")


# %% [markdown]
# # Q 2.4.18. Plotting Results
# To illustrate the difference between standard gradients and natural gradients, we print o

# %%

hist_gd_np = np.array(history_gd)
hist_ngd_np = np.array(history_ngd)

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10), sharex=True)
fig.suptitle(f"Gamma: Standard Gradient vs. Natural Gradient "
             f"(LR={LEARNING_RATE}, N={N_DATA})", fontsize=16)

# Plot 1: alpha (shape)
ax1.plot(hist_gd_np[:, 0],  label="GD alpha",  color='blue', linestyle='--')
ax1.plot(hist_ngd_np[:, 0], label="NGD alpha", color='red')
```

```
ax1.axhline(ALPHA_TRUE, color='black', linestyle=':',
            label=f"True alpha ({ALPHA_TRUE})")
ax1.set_ylabel("Shape parameter $\\alpha$")
ax1.legend()
ax1.grid(True)

# Plot 2: beta (scale)
ax2.plot(hist_gd_np[:, 1],  label="GD beta",  color='blue', linestyle='--')
ax2.plot(hist_ngd_np[:, 1], label="NGD beta", color='red')
ax2.axhline(BETA_TRUE, color='black', linestyle=':',
            label=f"True beta ({BETA_TRUE})")
ax2.set_xlabel("Epoch")
ax2.set_ylabel("Scale parameter $\\beta$")
ax2.legend()
ax2.grid(True)

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
# Save to Plots directory:
plt.savefig("Plots/gamma_parameters_evolution.png")
plt.show()

# Plot the evolution of the negative log-likelihoods
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot([gamma_nll(torch.tensor(a), torch.tensor(b), data).item()
        for a, b in history_gd], label='GD NLL', color='blue', linestyle='--')
ax.plot([gamma_nll(torch.tensor(a), torch.tensor(b), data).item()
        for a, b in history_ngd], label='NGD NLL', color='red')
ax.set_xlabel('Epoch')
ax.set_ylabel('Negative Log-Likelihood')
ax.set_title('Negative Log-Likelihood Evolution')
ax.legend()
ax.grid(True)
# Save to Plots directory:
plt.savefig("Plots/gamma_nll_evolution.png")
plt.show()
```

Code - 1AD-B1.py

```
# %% [markdown]
# # ***Reparameterization of the categorical distribution***
#
#
#
#

# %% [markdown]
# We will work with Torch throughout this notebook.

# %%
import matplotlib.pyplot as plt
import torch
from torch.distributions import Beta, Categorical, Gumbel
from torch.nn import functional as F

# %%
torch.manual_seed(0)

# %% [markdown]
# A helper function to visualize the generated samples:

# %%


def compare_samples(samples_1, samples_2, bins=10, range=None):
    # Make sure both hist plots can be seen:
    fig = plt.figure()
    if range is not None:
        plt.hist(samples_1, bins=bins, range=range, alpha=0.5)
        plt.hist(samples_2, bins=bins, range=range, alpha=0.5)
    else:
        plt.hist(samples_1, bins=bins, alpha=0.5)
        plt.hist(samples_2, bins=bins, alpha=0.5)

    plt.xlabel('value')
    plt.ylabel('number of samples')
    plt.legend(['direct', 'via reparameterization'])
    plt.show()

# %% [markdown]
# ### ***Categorical Distribution***
# Below write a function that generates N samples from Categorical (**a**), where **a** = $

# %%


def categorical_sampler(a, N):
    # Generate N samples from categorical distribution with probs a

    dist = Categorical(probs=a)
    samples = dist.sample((N,))

    return samples  # should be size N

# %% [markdown]
# Now write a function that generates samples from Categorical (**a**) via reparameterizati
#
#
#

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

    dist = Gumbel(0, 1)
    u = dist.sample((N, a.shape[0]))
    samples = F.softmax((torch.log(a + eps) + u) / temp, dim=1)

    return samples  # make sure that your implementation allows the gradient to backpropaga


# %% [markdown]
# Generate samples when $a = [0.1,0.2,0.5,0.2]$ and visualize them:

# %%
a = torch.tensor([0.1, 0.2, 0.5, 0.2])
N = 1000
direct_samples = categorical_sampler(a, N)
reparametrized_samples = categorical_reparametrize(
    a, N, temp=0.1, eps=1e-20)  # N x 4
# Convert reparametrized samples to hard samples
hard_samples = torch.argmax(reparametrized_samples, dim=1)
compare_samples(direct_samples, hard_samples, bins=4)
```