```
import matplotlib.pyplot as plt
import random
import torch
import torch.optim as optim

torch.manual_seed(0)
random.seed(0)
```

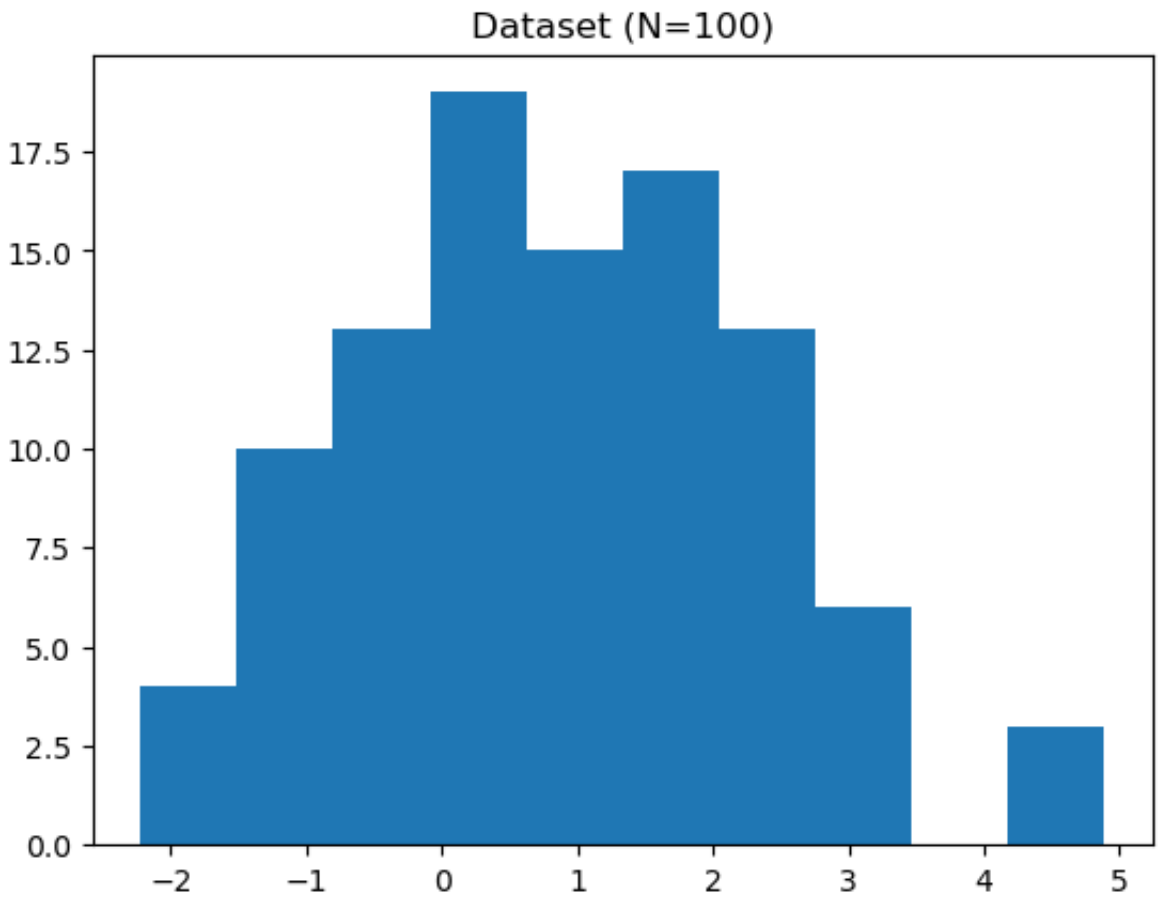
Assignment 1.2.5 - BBVI - Algorithm 2

BBVI algorithm II i.e. without Rao-Blackwellization

```
def generate_data(mu, tau, N):
  x = torch.linspace(-10, 10, N)
  # Insert your code here
  sigma = 1 / torch.sqrt(torch.tensor(tau))   # precision τ = 1/σ²
  torch.manual_seed(10)

  D = torch.normal(mu, sigma, size=(N,))

  return D
```

Set μ = 1, τ = 0.5 and generate a dataset with size N=100. Plot the histogram for the generated dataset.

```
mu = 1
tau = 0.5

dataset = generate_data(mu, tau, 100)

# Visulaize the datasets via histograms
plt.figure()
plt.hist(dataset)
plt.title("Dataset (N=100)")

plt.show()
```




```
class BlackBoxVI:
    """Black Box Variational Inference implementation."""

    def __init__(self, D, log_joint_distribution, variational_famil
        """
        Black Box Variational Inference implementation without any
            Args:
                D: dataset
```

```
                log_joint_distribution: function that computes the
                variational_family_q: variational family q with par
                S: number of samples for Monte Carlo estimation
                learning_rate: learning rate for the optimizer
        """
        self.D = D
        self.log_joint_distribution = log_joint_distribution
        self.variational_family_q = variational_family_q
        self.S = S
        self.learning_rate = learning_rate


    def fit(self, max_iterations = 1000):
        """Fit the variational parameters using BBVI algorithm.
        SGD optimization
        Args:
            threshold: convergence threshold
            max_iterations: maximum number of iterations
        """
        history = {
            "elbo" : [],
            "final_params" : None,
            'mu_expected': [],
            'tau_expected': [],
        }
        lr_sgd = self.learning_rate      # A small learning rate (S
        optimizer_sgd = optim.SGD([self.variational_family_q.parame

        for t in range(1,max_iterations+1):
            elbo=0
            loss=0
            optimizer_sgd.zero_grad()

            number_of_variational_parameters = self.variational_fam

            f = torch.zeros((number_of_variational_parameters, self
            h = torch.zeros((number_of_variational_parameters, self

            for s in range(self.S):
                z_s = self.variational_family_q.sample()

                #compute log(q(z[s]))
                log_q_z_s = self.variational_family_q.log_prob(z_s)

                # Compute log p(x, z[s])
                log_p_z_s_D = self.log_joint_distribution(self.D, z

                # Compute the score function ∇_λ log q(z[s]; λ)
```

```
            learning_signal = (log_p_z_s_D - log_q_z_s).detach(

            #loss
            elbo += learning_signal

            # We need to compute ∇_λ log q(z[s]; λ) for each pa
            grad_log_q_z_s = torch.autograd.grad(log_q_z_s,
                                                 self.variational_
                                                 retain_graph=True
                                                 create_graph=Fals

             # For each variational parameter d
            for d in range(number_of_variational_parameters):
                # Control variate computation
                grad_d = grad_log_q_z_s[d]

                # Compute f_t and h_t for this sample
                f[d, s] += learning_signal * grad_d
                h[d, s] += grad_d

        elbo /= self.S


        # Compute gradient with control variates for each param
        final_gradient = torch.zeros(number_of_variational_para

        for d in range(number_of_variational_parameters):
            # a_d* = Cov(f_d, h_d) / Var(h_d)

            f_d_mean = f[d].mean()
            h_d_mean = h[d].mean()

            # Covariance
            cov_f_h = ((f[d] - f_d_mean) * (h[d] - h_d_mean)).m

            # Variance of h_d
            var_h = ((h[d] - h_d_mean) ** 2).mean()

            # We add small epsilon to avoid division by zero
            if var_h > 1e-8:
                a_d_star = cov_f_h / var_h
            else:
                a_d_star = 0.0

            # ∇_λd L ≈ (1/S) Σ[f_i[s] - a_d* h_i[s]]
            final_gradient[d] = (f[d] - a_d_star * h[d]).mean()

        with torch.no_grad():
```

```
                self.variational_family_q.parameters += self.learni

            #we compute the elb and we check the params every 10 it
            if t % 10 == 0 or t == 1:
                mu_N,lambda_N, alpha_N,beta_N = self.variational_fa
                history["elbo"].append((t,elbo))
                history['mu_expected'].append((t,mu_N.item()))
                history['tau_expected'].append((t,(alpha_N / beta_N

        history['final_params'] = self.variational_family_q.get_par

        return history
```

Gaussian Variational Family

```
class NormalGammaVariationalFamily():

    """Variational family for Normal-NormalGamma conjugate model.

        Variational distribution: q(μ, τ | λ) = q(μ | τ) q(τ)
        where:
            - μ | τ ~ Normal(μ_N, (λ_N * τ)^(-1))
            - τ ~ Gamma(α_N, β_N)

         Parameters: λ = [μ_N, λ_N, α_N, β_N]
    """
    def __init__(self):
        """Initialize with dimension of latent variable."""
        mu_N = torch.randn(1).item()
        lambda_N = torch.rand(1).item() * 2 + 0.5
        alpha_N = torch.rand(1).item() * 3 + 1.0
        beta_N = torch.rand(1).item() * 3 + 0.5

        self.parameters = torch.tensor([
            mu_N, torch.log(torch.tensor(lambda_N)), torch.log(torc
        ], requires_grad=True)

    def get_actual_parameters(self):
        mu_N = self.parameters[0]

        lambda_N = torch.exp(self.parameters[1])
        alpha_N = torch.exp(self.parameters[2])
        beta_N = torch.exp(self.parameters[3])
```

```
        return mu_N, lambda_N, alpha_N, beta_N

    def get_parameters(self):
        return self.parameters.clone()

    def get_number_of_parameters(self):
        return len(self.parameters)

    def set_parameters(self,new_params):
        self.parameters = new_params.clone()


    def sample(self):
        """Sample from the variational distribution.
            z[s] ~ q(μ, τ | λ)
        """

        mu_N, lambda_N, alpha_N, beta_N = self.get_actual_parameter

        tau = torch.distributions.Gamma(alpha_N, beta_N).sample()


        precision = lambda_N * tau

        sigma_mu = 1.0 / torch.sqrt(precision)

        mu = torch.distributions.Normal(mu_N, sigma_mu).sample()

        return (mu,tau)

    def log_prob(self, z):
        """Compute log probability of z under the variational distr
            Compute log q(μ, τ | λ).
         """
        mu, tau = z
        mu_N, lambda_N, alpha_N, beta_N = self.get_actual_parameter


        log_q_tau = torch.distributions.Gamma(alpha_N, beta_N).log_


        precision = lambda_N * tau

        sigma_mu = 1.0 / torch.sqrt(precision)

        log_q_mu_given_tau = torch.distributions.Normal(mu_N,sigma_

        return log_q_tau + log_q_mu_given_tau
```

```
    def score_function_handmade_computed(self,z):
        """
        I coomputed the score function manually ( it needs to clamp
        Compute the score function ∇_λ log q(z; λ).

          z = (μ^(s), τ^(s)) is a sample from the variational distr
          λ = [μ_N, λ_N, α_N, β_N] are the variational parameters
        """

        mu, tau = z
        mu_N, lambda_N, alpha_N, beta_N = self.get_actual_parameter
        eps = self.eps

        lambda_N_safe = lambda_N.clamp(min=eps)
        alpha_N_safe = alpha_N.clamp(min=eps)
        beta_N_safe = beta_N.clamp(min=eps)
        tau_safe = tau.clamp(min=eps)

        # Gradient log q(μ | λ) with respect to λ
        grad_mu_wrt_mu_N = (mu - mu_N) / lambda_N_safe
        grad_mu_wrt_lambda_N = ((mu - mu_N)**2 - lambda_N) / (2*lam

        # Gradient log q(τ | λ) with respect to λ
        grad_tau_wrt_alpha_N = torch.log(beta_N_safe) - torch.digam
        grad_tau_wrt_beta_N = (alpha_N_safe / beta_N_safe) - tau


         # === Apply chain rule for log-parametrization ===

        grad_log_mu_N = grad_mu_wrt_mu_N

        grad_log_lambda_N = lambda_N * grad_mu_wrt_lambda_N

        grad_log_alpha_N = alpha_N * grad_tau_wrt_alpha_N

        grad_log_beta_N = beta_N * grad_tau_wrt_beta_N

        return torch.stack([grad_log_mu_N, grad_log_lambda_N,
                            grad_log_alpha_N, grad_log_beta_N])
```

Our Model : the initial parameters have been taken from the last assignment.

```
def log_joint_distribution(D, z):
        """Compute the log joint distribution log p(D, Z).
        Args:
            D: dataset
            Z: latent variables
        Returns:
            log p(D, Z)
        """
        # log p(D, Z) = log p(D|Z) + log p(Z)
        # Z = (mu, tau)

        mu, tau = z
        sigma = 1 / torch.sqrt(tau)

        mu_0 = 1.0
        lambda_0 = 0.1
        a_0 = 1.0
        b_0 = 2.0

        #log P(D|Z)
        log_likelihood = torch.distributions.Normal(mu, sigma).log_

        #log P(mu , tau) = log P(mu | tau) + log P(tau)

        # Log prior p(μ | τ)
        precision_mu = lambda_0 * tau
        sigma_mu = 1.0 / torch.sqrt(precision_mu)
        log_prior_mu = torch.distributions.Normal(mu_0, sigma_mu).l

        # Log prior p(τ)
        log_prior_tau = torch.distributions.Gamma(a_0, b_0).log_pro


        return log_likelihood + log_prior_mu + log_prior_tau
```

Application

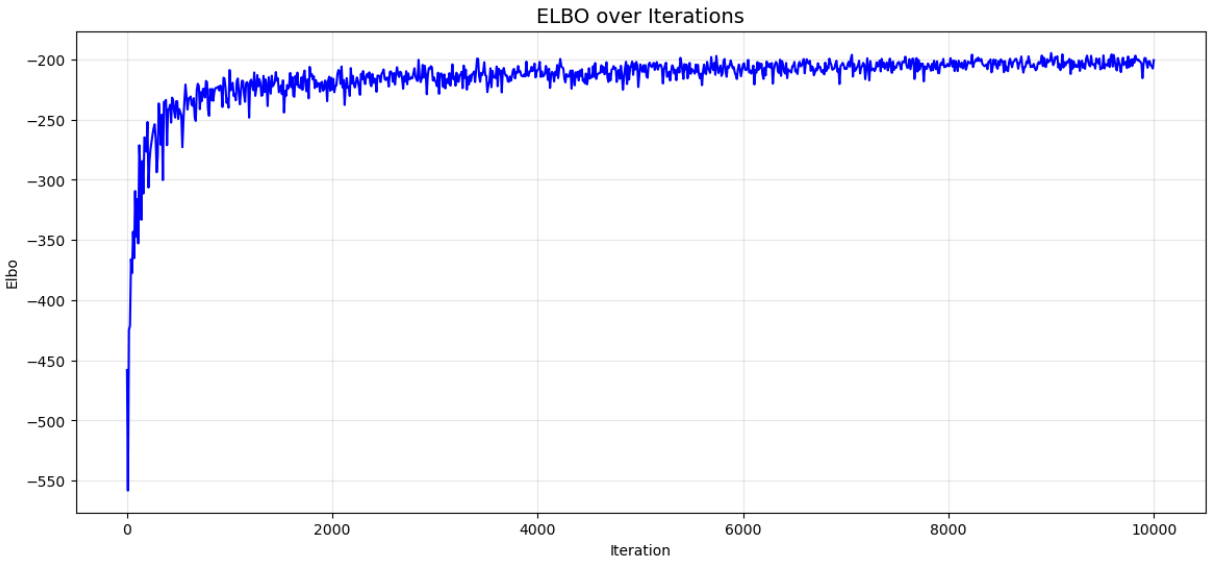
```
q = NormalGammaVariationalFamily()

bbvi = BlackBoxVI(dataset,log_joint_distribution=log_joint_distribu
results = bbvi.fit(max_iterations=10**4)
```

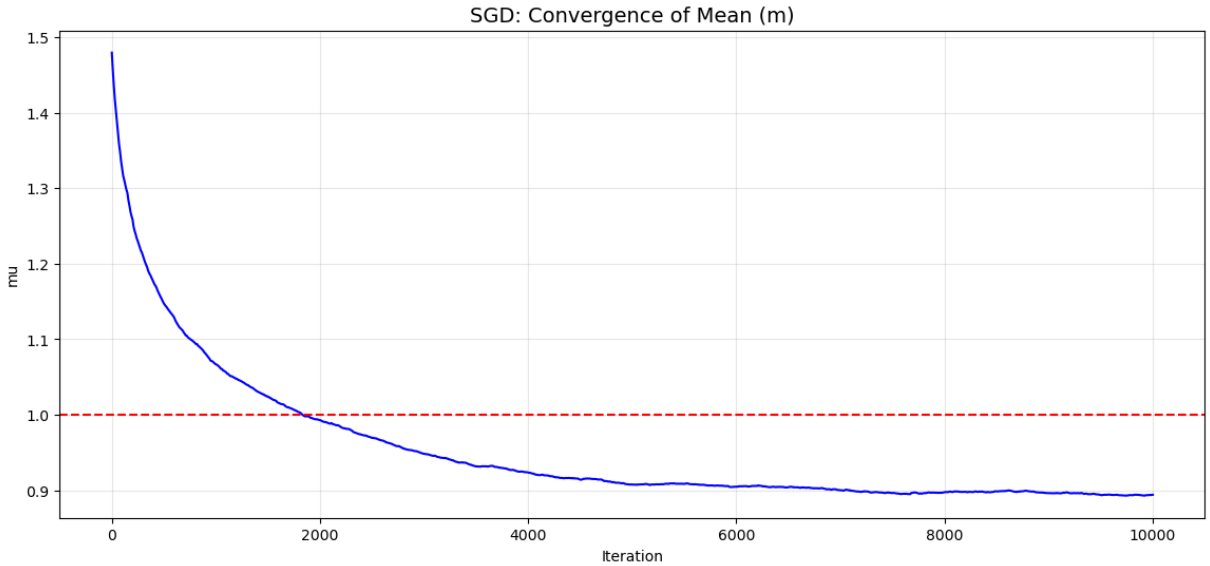
Visualizing the results

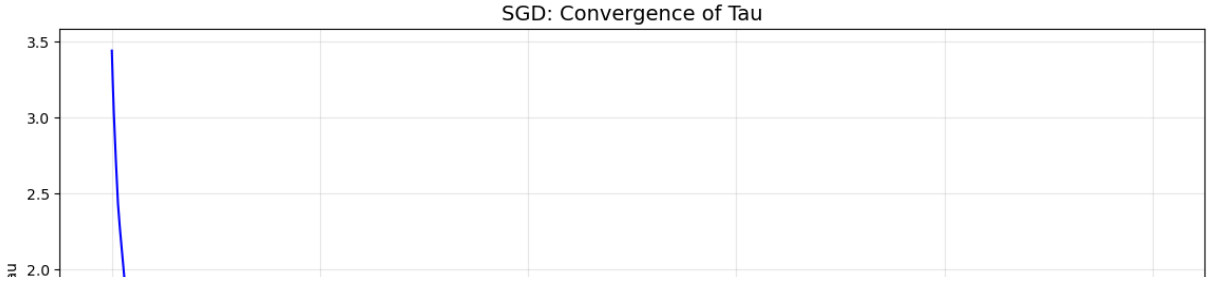
```
"""
results = {
            "elbo" : [],
            "final_params" : float,
            'mu_expected': float,
            'tau_expected': float,
        }
"""
print(results['mu_expected'][-1])
print(f"\n{'='*60}")
print(f"Final Results:")
print(f"E_q[μ] = {results['mu_expected'][-1][1]:.4f} (true: {mu})")
print(f"E_q[τ] = {results['tau_expected'][-1][1]:.4f} (true: {tau})
print(f"Data mean: {dataset.mean():.4f}")
print(f"Data precision: {1.0/dataset.var():.4f}")
print(f"{'='*60}")

### Plot of the ELBO
plt.figure(figsize=(14, 6))
iterations, elbos = zip(*results["elbo"])
plt.plot(iterations,elbos,'b-')
plt.xlabel("Iteration")
plt.ylabel("Elbo")
plt.title('ELBO over Iterations', fontsize=14)
plt.grid(alpha=0.3)


### Plot for m
plt.figure(figsize=(14, 6))
iterations, mu_expected = zip(*results["mu_expected"])
plt.plot(iterations,mu_expected,'b-')
plt.axhline(mu, color='r', linestyle='--', label="real value")
plt.xlabel("Iteration")
plt.ylabel("mu")
plt.title('SGD: Convergence of Mean (m)', fontsize=14)
plt.grid(alpha=0.3)




### Plot for tau
plt.figure(figsize=(14, 6))
iterations, tau_expected = zip(*results["tau_expected"])
plt.plot(iterations,tau_expected,'b-')
plt.axhline(tau, color='r', linestyle='--', label="real value")
```

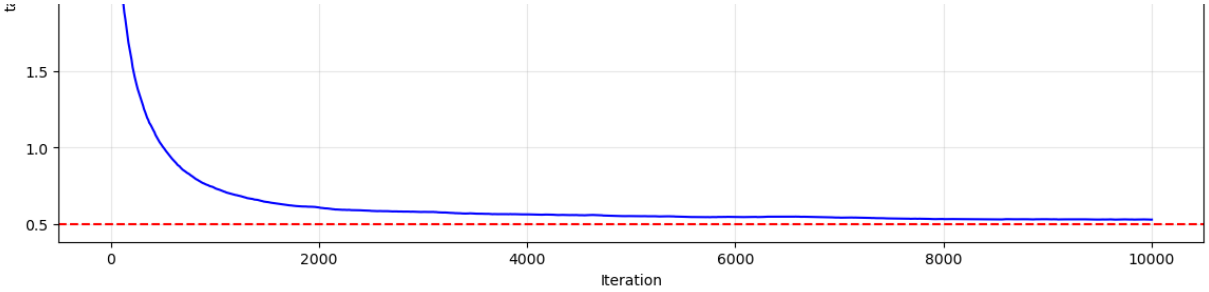
```
plt.xlabel("Iteration")
plt.ylabel("tau")
plt.title('SGD: Convergence of Tau', fontsize=14)
plt.grid(alpha=0.3)
```

```
(10000, 0.8939169645309448)

============================================================
Final Results:
E_q[μ] = 0.8939 (true: 1)
E_q[τ] = 0.5269 (true: 0.5)
Data mean: 0.8901
Data precision: 0.4580
============================================================
```








1.5
1.0
0.5
0
2000
4000
6000
8000
10000
Iteration

```
import matplotlib.pyplot as plt
import random
import torch
import torch.optim as optim

torch.manual_seed(0)
random.seed(0)
```

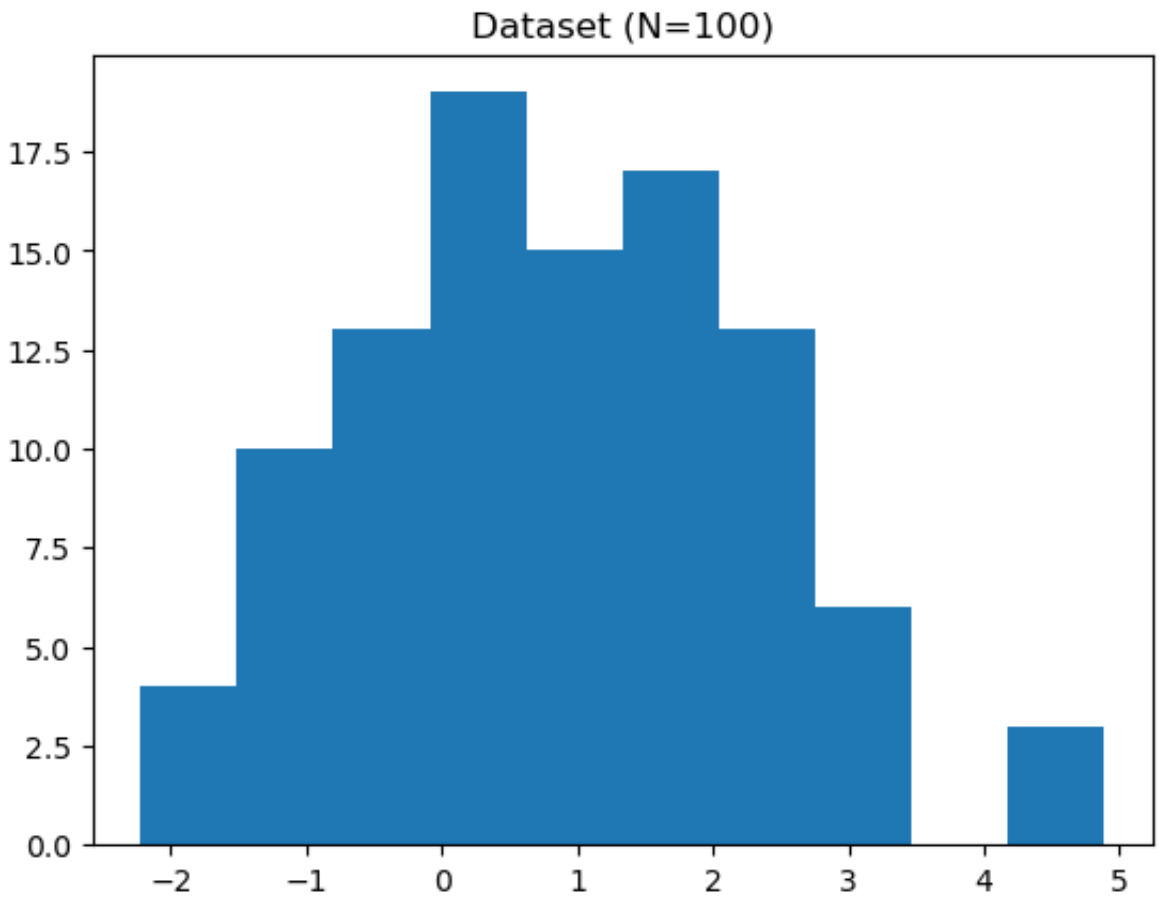
Assignment 1.2.5 - BBVI - Algorithm 1

Naive BBVI algorithm, i.e., without Rao-Blackwellization and Control Variates, called Algorithm 1 in the BBVI paper.

```
def generate_data(mu, tau, N):
  x = torch.linspace(-10, 10, N)
  # Insert your code here
  sigma = 1 / torch.sqrt(torch.tensor(tau))   # precision τ = 1/σ²
  torch.manual_seed(10)

  D = torch.normal(mu, sigma, size=(N,))

  return D
```

Set μ = 1, τ = 0.5 and generate a dataset with size N=100. Plot the histogram for the generated dataset.

```
mu = 1
tau = 0.5

dataset = generate_data(mu, tau, 100)

# Visulaize the datasets via histograms
plt.figure()
plt.hist(dataset)
plt.title("Dataset (N=100)")

plt.show()
```




```
class BlackBoxVI:
    """Black Box Variational Inference implementation."""

    def __init__(self, D, log_joint_distribution, variational_famil
        """
        Black Box Variational Inference implementation without any
            Args:
                D: dataset
                log_joint_distribution: function that computes the
```

```
                variational_family_q: variational family q with par
                S: number of samples for Monte Carlo estimation
                learning_rate: learning rate for the optimizer
        """
        self.D = D
        self.log_joint_distribution = log_joint_distribution
        self.variational_family_q = variational_family_q
        self.S = S
        self.learning_rate = learning_rate


    def fit(self, max_iterations = 1000):
        """Fit the variational parameters using BBVI algorithm.
        SGD optimization
        Args:
            threshold: convergence threshold
            max_iterations: maximum number of iterations
        """
        history = {
            "elbo" : [],
            "final_params" : None,
            'mu_expected': [],
            'tau_expected': [],
        }
        lr_sgd = self.learning_rate      # A small learning rate (S
        optimizer_sgd = optim.SGD([self.variational_family_q.parame

        for t in range(1,max_iterations+1):
            elbo=0
            loss=0
            optimizer_sgd.zero_grad()
            for s in range(self.S):
                z_s = self.variational_family_q.sample()

                #compute log(q(z[s]))
                log_q_z_s = self.variational_family_q.log_prob(z_s)

                # Compute log p(x, z[s])
                log_p_z_s_D = self.log_joint_distribution(self.D, z

                # Compute the score function ∇_λ log q(z[s]; λ)
                learning_signal = (log_p_z_s_D - log_q_z_s).detach(

                #loss
                elbo += learning_signal
                loss += (- log_q_z_s * learning_signal)

            elbo /= self.S
```

```
            loss /= self.S

            loss.backward()
            optimizer_sgd.step()


            #we compute the elb and we check the params every 10 it
            if t % 10 == 0 or t == 1:
                mu_N,lambda_N, alpha_N,beta_N = self.variational_fa
                history["elbo"].append((t,elbo))
                history['mu_expected'].append((t,mu_N.item()))
                history['tau_expected'].append((t,(alpha_N / beta_N

        history['final_params'] = self.variational_family_q.get_par

        return history
```

Gaussian Variational Family

```
class NormalGammaVariationalFamily():

    """Variational family for Normal-NormalGamma conjugate model.

        Variational distribution: q(μ, τ | λ) = q(μ | τ) q(τ)
        where:
            - μ | τ ~ Normal(μ_N, (λ_N * τ)^(-1))
            - τ ~ Gamma(α_N, β_N)

     Parameters: λ = [μ_N, λ_N, α_N, β_N]
    """
    def __init__(self):
        """Initialize with dimension of latent variable."""
        mu_N = torch.randn(1).item()
        lambda_N = torch.rand(1).item() * 2 + 0.5
        alpha_N = torch.rand(1).item() * 3 + 1.0
        beta_N = torch.rand(1).item() * 3 + 0.5

        self.parameters = torch.tensor([
            mu_N, torch.log(torch.tensor(lambda_N)), torch.log(torc
        ], requires_grad=True)

    def get_actual_parameters(self):
        mu_N = self.parameters[0]
```

```
        lambda_N = torch.exp(self.parameters[1])
        alpha_N = torch.exp(self.parameters[2])
        beta_N = torch.exp(self.parameters[3])

        return mu_N, lambda_N, alpha_N, beta_N

    def get_parameters(self):
        return self.parameters.clone()

    def set_parameters(self,new_params):
        self.parameters = new_params.clone()


    def sample(self):
        """Sample from the variational distribution.
            z[s] ~ q(μ, τ | λ)
        """

        mu_N, lambda_N, alpha_N, beta_N = self.get_actual_parameter

        tau = torch.distributions.Gamma(alpha_N, beta_N).sample()


        precision = lambda_N * tau

        sigma_mu = 1.0 / torch.sqrt(precision)

        mu = torch.distributions.Normal(mu_N, sigma_mu).sample()

        return (mu,tau)

    def log_prob(self, z):
        """Compute log probability of z under the variational distr
            Compute log q(μ, τ | λ).
         """
        mu, tau = z
        mu_N, lambda_N, alpha_N, beta_N = self.get_actual_parameter


        log_q_tau = torch.distributions.Gamma(alpha_N, beta_N).log_


        precision = lambda_N * tau

        sigma_mu = 1.0 / torch.sqrt(precision)

        log_q_mu_given_tau = torch.distributions.Normal(mu_N,sigma_
```

```
        return log_q_tau + log_q_mu_given_tau


    def score_function_handmade_computed(self,z):
        """
        I coomputed the score function manually ( it needs to clamp
        Compute the score function ∇_λ log q(z; λ).

          z = (μ^(s), τ^(s)) is a sample from the variational distr
          λ = [μ_N, λ_N, α_N, β_N] are the variational parameters
        """

        mu, tau = z
        mu_N, lambda_N, alpha_N, beta_N = self.get_actual_parameter
        eps = self.eps

        lambda_N_safe = lambda_N.clamp(min=eps)
        alpha_N_safe = alpha_N.clamp(min=eps)
        beta_N_safe = beta_N.clamp(min=eps)
        tau_safe = tau.clamp(min=eps)

        # Gradient log q(μ | λ) with respect to λ
        grad_mu_wrt_mu_N = (mu - mu_N) / lambda_N_safe
        grad_mu_wrt_lambda_N = ((mu - mu_N)**2 - lambda_N) / (2*lam

        # Gradient log q(τ | λ) with respect to λ
        grad_tau_wrt_alpha_N = torch.log(beta_N_safe) - torch.digam
        grad_tau_wrt_beta_N = (alpha_N_safe / beta_N_safe) - tau


         # === Apply chain rule for log-parametrization ===

        grad_log_mu_N = grad_mu_wrt_mu_N

        grad_log_lambda_N = lambda_N * grad_mu_wrt_lambda_N

        grad_log_alpha_N = alpha_N * grad_tau_wrt_alpha_N

        grad_log_beta_N = beta_N * grad_tau_wrt_beta_N

        return torch.stack([grad_log_mu_N, grad_log_lambda_N,
                            grad_log_alpha_N, grad_log_beta_N])
```

Our Model : the initial parameters have been taken from the last assignment.

```
def log_joint_distribution(D, z):
        """Compute the log joint distribution log p(D, Z).
        Args:
            D: dataset
            Z: latent variables
        Returns:
            log p(D, Z)
        """
        # log p(D, Z) = log p(D|Z) + log p(Z)
        # Z = (mu, tau)

        mu, tau = z
        sigma = 1 / torch.sqrt(tau)

        mu_0 = 1.0
        lambda_0 = 0.1
        a_0 = 1.0
        b_0 = 2.0

        #log P(D|Z)
        log_likelihood = torch.distributions.Normal(mu, sigma).log_

        #log P(mu , tau) = log P(mu | tau) + log P(tau)

        # Log prior p(μ | τ)
        precision_mu = lambda_0 * tau
        sigma_mu = 1.0 / torch.sqrt(precision_mu)
        log_prior_mu = torch.distributions.Normal(mu_0, sigma_mu).l

        # Log prior p(τ)
        log_prior_tau = torch.distributions.Gamma(a_0, b_0).log_pro


        return log_likelihood + log_prior_mu + log_prior_tau
```

Application

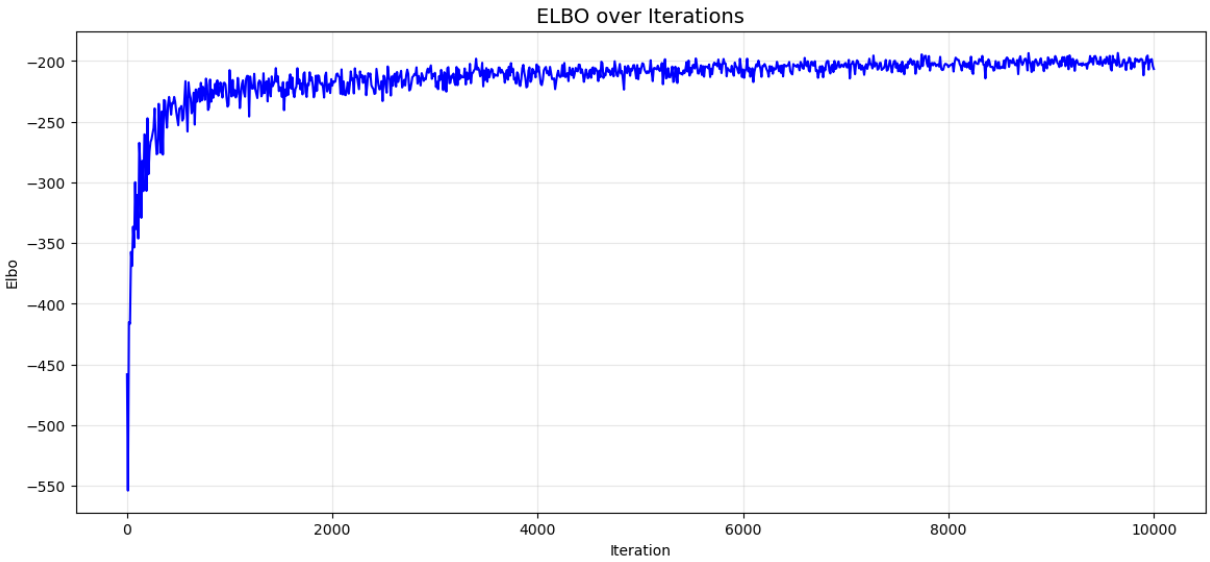
```
q = NormalGammaVariationalFamily()

bbvi = BlackBoxVI(dataset,log_joint_distribution=log_joint_distribu
results = bbvi.fit(max_iterations=10**4)
```

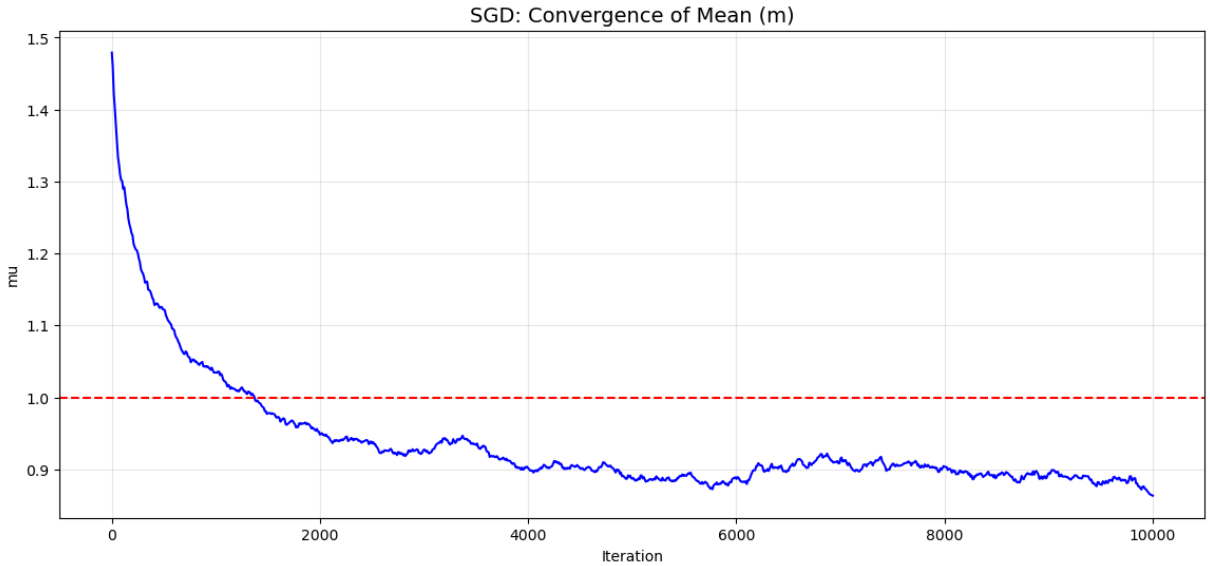
Visualizing the results

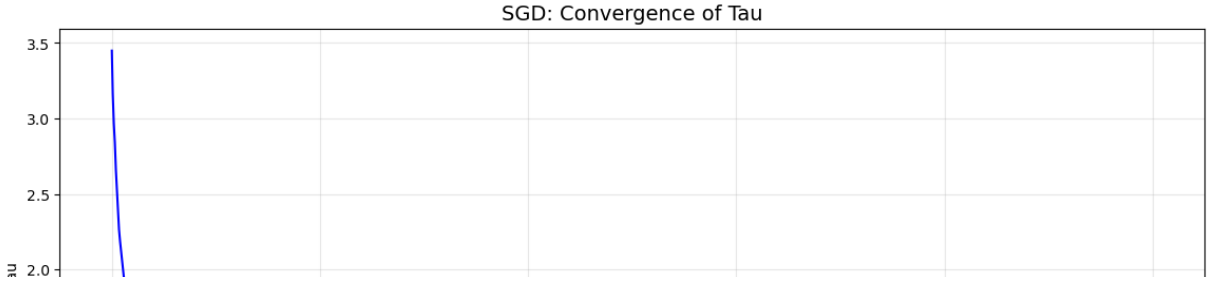
```
"""
results = {
            "elbo" : [],
            "final_params" : float,
            'mu_expected': float,
            'tau_expected': float,
        }
"""
print(results['mu_expected'][-1])
print(f"\n{'='*60}")
print(f"Final Results:")
print(f"E_q[μ] = {results['mu_expected'][-1][1]:.4f} (true: {mu})")
print(f"E_q[τ] = {results['tau_expected'][-1][1]:.4f} (true: {tau})
print(f"Data mean: {dataset.mean():.4f}")
print(f"Data precision: {1.0/dataset.var():.4f}")
print(f"{'='*60}")

### Plot of the ELBO
plt.figure(figsize=(14, 6))
iterations, elbos = zip(*results["elbo"])
plt.plot(iterations,elbos,'b-')
plt.xlabel("Iteration")
plt.ylabel("Elbo")
plt.title('ELBO over Iterations', fontsize=14)
plt.grid(alpha=0.3)


### Plot for m
plt.figure(figsize=(14, 6))
iterations, mu_expected = zip(*results["mu_expected"])
plt.plot(iterations,mu_expected,'b-')
plt.axhline(mu, color='r', linestyle='--', label="real value")
plt.xlabel("Iteration")
plt.ylabel("mu")
plt.title('SGD: Convergence of Mean (m)', fontsize=14)
plt.grid(alpha=0.3)




### Plot for tau
plt.figure(figsize=(14, 6))
iterations, tau_expected = zip(*results["tau_expected"])
plt.plot(iterations,tau_expected,'b-')
plt.axhline(tau, color='r', linestyle='--', label="real value")
```

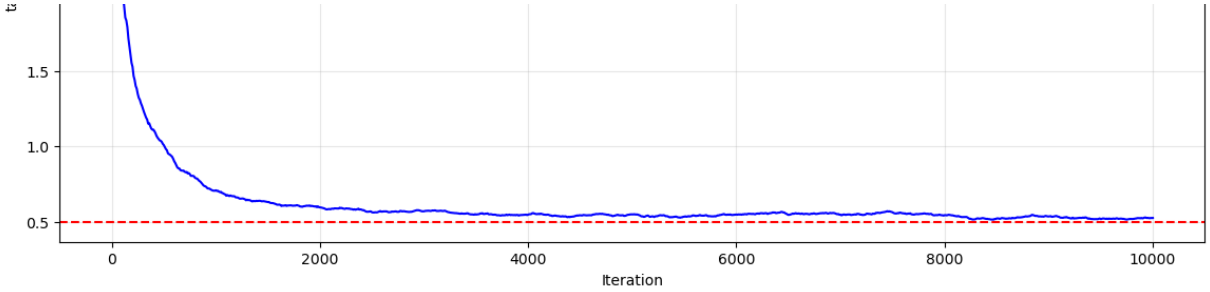
```
plt.xlabel("Iteration")
plt.ylabel("tau")
plt.title('SGD: Convergence of Tau', fontsize=14)
plt.grid(alpha=0.3)
```

```
(10000, 0.8634085655212402)

============================================================
Final Results:
E_q[μ] = 0.8634 (true: 1)
E_q[τ] = 0.5255 (true: 0.5)
Data mean: 0.8901
Data precision: 0.4580
============================================================
```








1.5
1.0
0.5
0
2000
4000
6000
8000
10000
Iteration

Q 2.4.17. Standard Gradient Descent vs. Natural Gradient Descent

In this notebbook, we compare standard gradient descent (GD) and natural gradient descent (NGD) for estimating the parameters of a 1D Gamma distribution from observed data. We will generate synthetic data from a known Gamma distribution and then attempt to recover its shape and scale parameters using both optimization methods.

1. Configuration

We define the true parameters of the Gamma distribution, the number of data points to generate, the learning rate, the number of epochs for optimization, and the initial guesses for the parameters. We also set a random seed for reproducibility.

```
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 = 150

# 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)
```

```
<torch._C.Generator at 0x122105750>
```

2. Data Generation

We generate synthetic data from the true Gamma distribution using PyTorch's `Gamma` distribution class. We sample `N_DATA` points and print the sample mean and variance to verify the generated data.

```
# Our parameterisation is shape–scale, but PyTorch's Gamma uses sha
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
print(f"Sample mean:     {data.mean().item():.4f}")
print(f"Sample variance: {data.var().item():.4f}\n")
```

```
Generated 1000 data points from Gamma(alpha=3.0, beta=2.0)
Sample mean:     6.0692
Sample variance: 11.9481
```

3. Loss function & parameter init

We define the negative log-likelihood loss function for the Gamma distribution. We also initialize the parameters for both standard gradient descent and natural gradient descent, and set up history trackers to record the parameter values over epochs.

```
# ToDo: Loss function
def gamma_nll(alpha, beta, data_points):
    """
    ToDo:
        Implement the average negative log-likelihood for Gamma dis

        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)

    # Convert data_points to tensor if not already
    data_tensor = torch.tensor(data_points) if not torch.is_tensor(

    empirical_mean_x = data_tensor.mean()
    empirical_mean_log_x = torch.log(data_tensor).mean()
    nll = +alpha*torch.log(beta) + torch.lgamma(alpha) - (alpha-1)*
    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 = []
history_loss_gd =[]
history_loss_ngd = []
```

4. Fisher Information inverse

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

      Theory:
        F(α, β) =
            [ ψ1(α)        1/β      ]
            [  1/β     α/β^2        ]

        F^{-1}(α, β) =
            1 / (α ψ1(α) - 1) *
            [  α           -β             ]
            [  -β       β^2 ψ1(α)         ]

      Hints:
      - Use torch.polygamma(1, alpha) for ψ1(α) (trigamma).
      - Make sure to detach alpha, beta so F^{-1} is not part of th
    """
    alpha_detached = alpha.detach()
    beta_detached = beta.detach()

    external_factor = 1/(alpha_detached* torch.polygamma(1, alpha_d
    inv11 = alpha_detached * external_factor
    inv12 = -beta_detached * external_factor
    inv22 = beta_detached**2 *  torch.polygamma(1, alpha_detached)
    return inv11, inv12, inv22
```

4. Optimization Loop

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.

```
Inizia a programmare o genera codice con l'IA.
```

```
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_loss_gd.append(loss_gd.item())
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}(α, β)
#  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_loss_ngd.append(loss_ngd.item())
        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.i
                  f"Loss={loss_gd.item():.4f}")
            print(f"  NGD: alpha={alpha_ngd.item():.4f}, beta={beta_ngd
                  f"Loss={loss_ngd.item():.4f}")


    print("\nOptimization finished.")
```

```
Optimizing with LR=0.1 for 150 epochs...

--- Epoch 1 ---
  GD:  alpha=0.6516, beta=8.0032, Loss=3.1866
  NGD: alpha=0.5340, beta=7.8695, Loss=3.1866

--- Epoch 15 ---
  GD:  alpha=1.0504, beta=7.9743, Loss=2.8331
  NGD: alpha=1.2770, beta=4.3331, Loss=2.7370

--- Epoch 30 ---
  GD:  alpha=1.0853, beta=7.9160, Loss=2.8296
  NGD: alpha=2.3102, beta=2.5556, Loss=2.5688

--- Epoch 45 ---
  GD:  alpha=1.0933, beta=7.8551, Loss=2.8270
  NGD: alpha=2.8686, beta=2.1013, Loss=2.5450

--- Epoch 60 ---
  GD:  alpha=1.0989, beta=7.7937, Loss=2.8245
  NGD: alpha=3.0273, beta=2.0018, Loss=2.5435

--- Epoch 75 ---
  GD:  alpha=1.1045, beta=7.7319, Loss=2.8219
  NGD: alpha=3.0627, beta=1.9810, Loss=2.5434

--- Epoch 90 ---
  GD:  alpha=1.1101, beta=7.6698, Loss=2.8193
  NGD: alpha=3.0701, beta=1.9768, Loss=2.5434

--- Epoch 105 ---
```

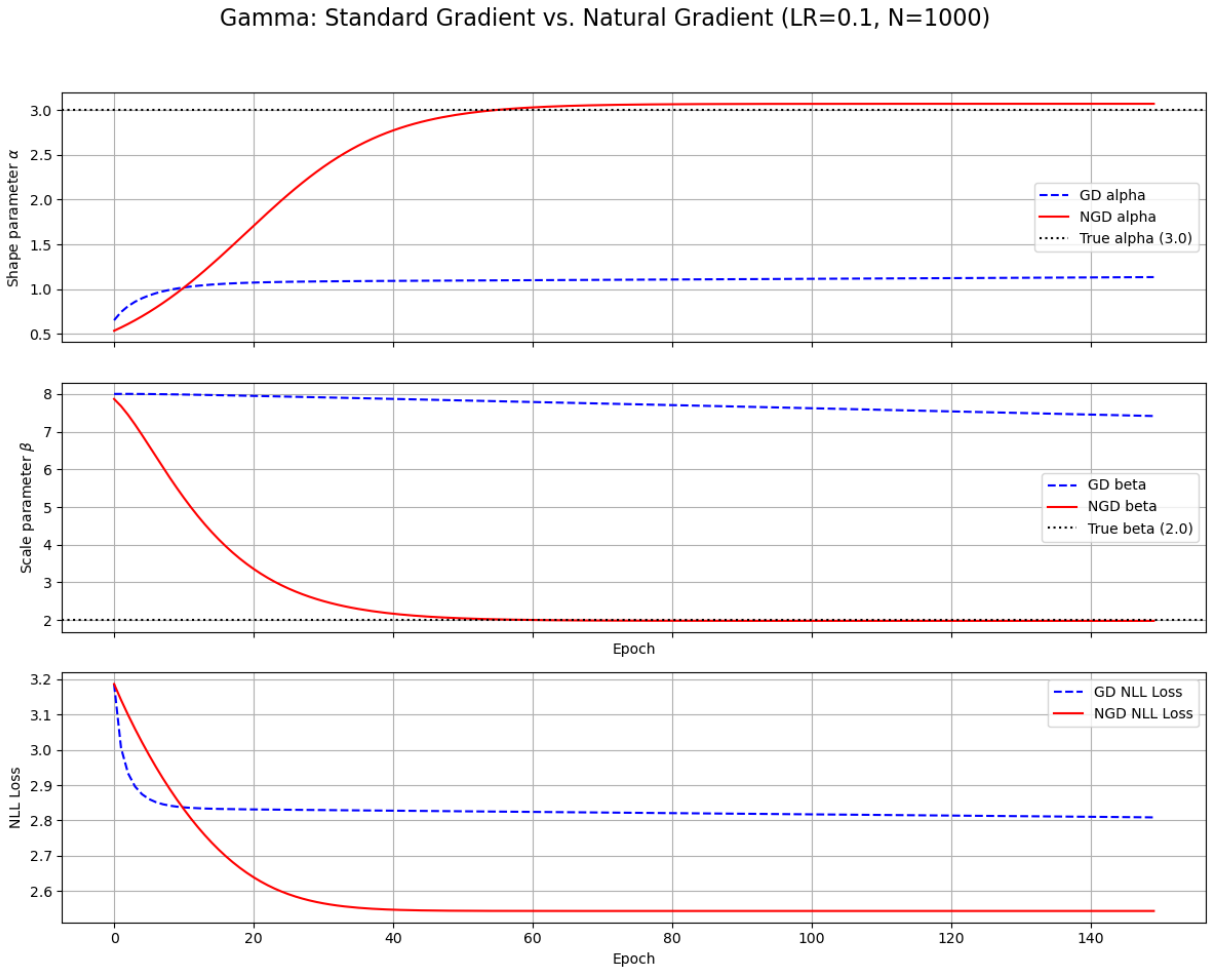
```
  GD:  alpha=1.1159, beta=7.6072, Loss=2.8167
  NGD: alpha=3.0716, beta=1.9759, Loss=2.5434

--- Epoch 120 ---
  GD:  alpha=1.1217, beta=7.5444, Loss=2.8141
  NGD: alpha=3.0719, beta=1.9757, Loss=2.5434

--- Epoch 135 ---
  GD:  alpha=1.1277, beta=7.4811, Loss=2.8114
  NGD: alpha=3.0720, beta=1.9757, Loss=2.5434

--- Epoch 150 ---
  GD:  alpha=1.1339, beta=7.4175, Loss=2.8086
  NGD: alpha=3.0720, beta=1.9756, Loss=2.5434

Optimization finished.
```

Q 2.4.18. Plotting Results

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

```
hist_gd_np  = np.array(history_gd)
hist_ngd_np = np.array(history_ngd)
hist_loss_gd = np.array(history_loss_gd)
hist_loss_ngd = np.array(history_loss_ngd)


fig, (ax1, ax2,ax3) = plt.subplots(3, 1, figsize=(12, 10), sharex=T
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', linest
ax1.plot(hist_ngd_np[:, 0], label="NGD alpha", color='red')
ax1.axhline(ALPHA_TRUE, color='black', linestyle=':', label=f"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', linesty
ax2.plot(hist_ngd_np[:, 1], label="NGD beta", color='red')
ax2.axhline(BETA_TRUE, color='black', linestyle=':', label=f"True b
ax2.set_xlabel("Epoch")
```

```
ax2.set_ylabel("Scale parameter $\\beta$")
ax2.legend()
ax2.grid(True)

# Plot 3: Negative Log-Likelihood (NLL) Loss
ax3.plot(hist_loss_gd,  label="GD NLL Loss",  color='blue', linesty
ax3.plot(hist_loss_ngd, label="NGD NLL Loss", color='red')
ax3.set_xlabel("Epoch")
ax3.set_ylabel("NLL Loss")
ax3.legend()
ax3.grid(True)



plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



Reparameterization of the categorical distribution

We will work with Torch throughout this notebook.

```
import torch
from torch.distributions import Beta #, ...  import the distributions
from torch.nn import functional as F
```

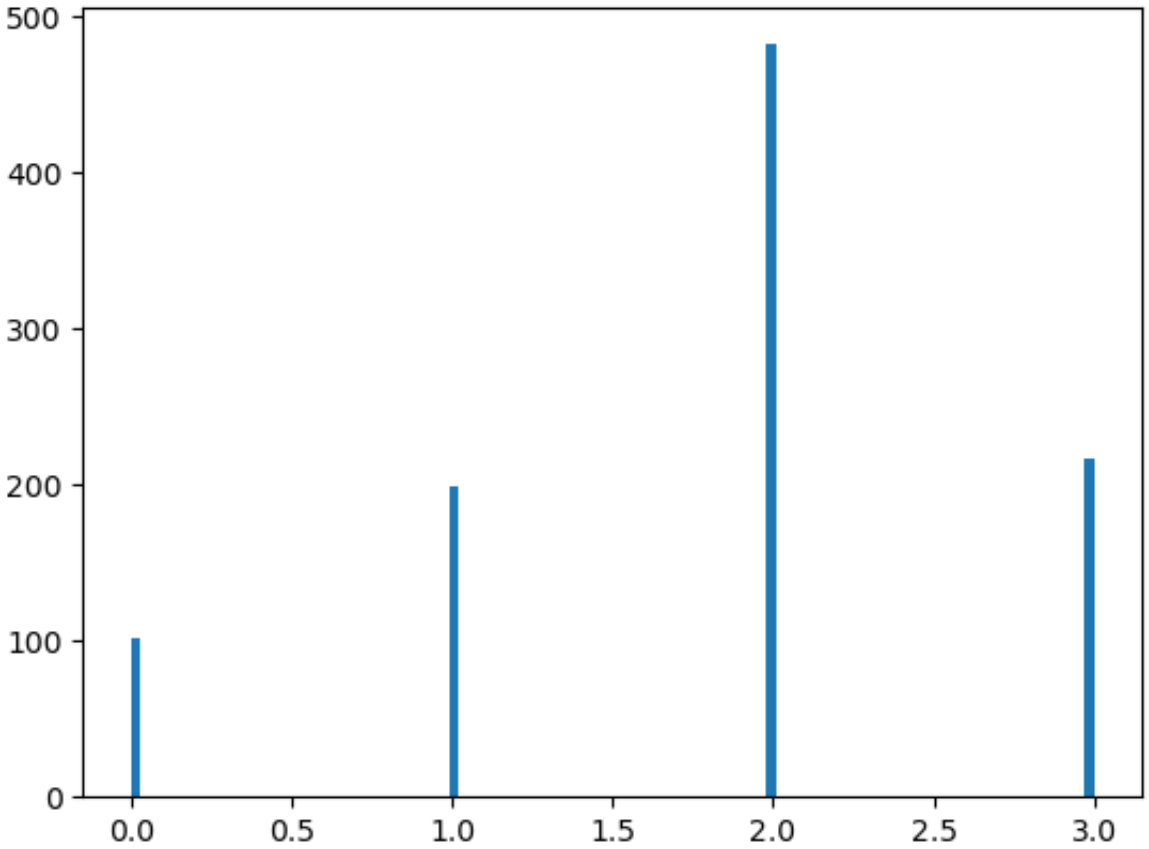
A helper function to visualize the generated samples:

```
import matplotlib.pyplot as plt
def compare_samples (samples_1, samples_2, bins=100, range=None):
  fig = plt.figure()
  if range is not None:
    plt.hist(samples_1, bins=bins, range=range)
    plt.hist(samples_2, bins=bins, range=range)
  else:
    plt.hist(samples_1, bins=bins)
    plt.hist(samples_2, bins=bins)
  plt.xlabel('value')
  plt.ylabel('number of samples')
  plt.legend(['direct','via reparameterization'])
  plt.show()
```

Categorical Distribution

Below write a function that generates N samples from Categorical (**a**), where **a** = $[a_0, a_1, a_2, a_3]$.

```
def categorical_sampler(a, N):
  samples = torch.distributions.Categorical(a).sample((N,))

  return samples  # should be N-by-1

## TEST
plt.figure()
samples = categorical_sampler(torch.tensor([0.1,0.2,0.5,0.2]), 1000)
plt.hist(samples, bins=100)
plt.show()
```

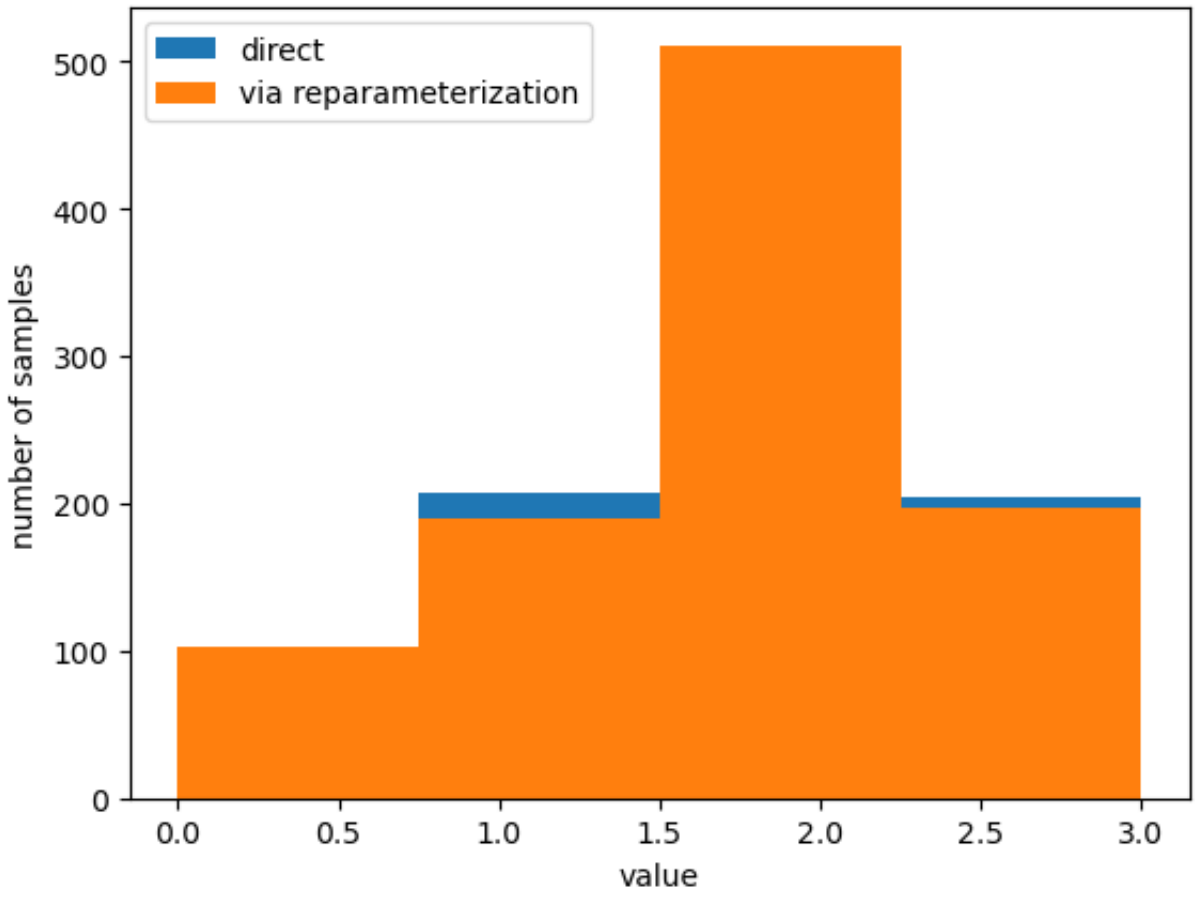


Now write a function that generates samples from Categorical (**a**) via reparameterization:

```
# Hint: approximate the Categorical distribution with the Gumbel-So
def categorical_reparametrize(a, N, temp=0.1, eps=1e-20):  # temp a
  g_gumbel_0_1 = torch.distributions.Gumbel(0,1).sample((N,len(a)))
  a.requires_grad = True
  x = torch.log(a + eps) + g_gumbel_0_1

  samples = torch.nn.functional.softmax(x / temp, dim=-1)

  return samples # make sure that your implementation allows the gr
```

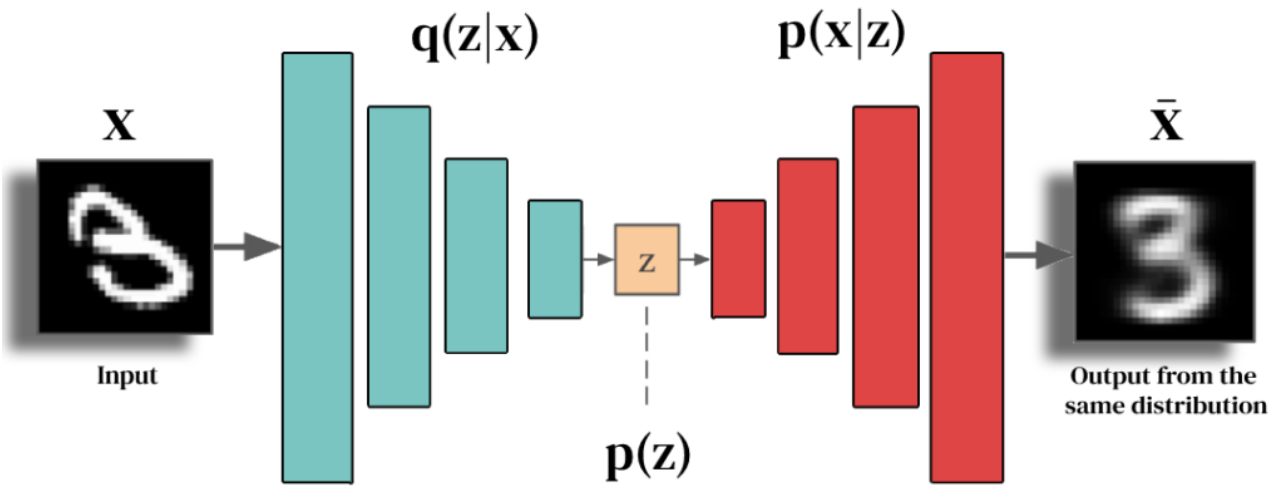
Generate samples when $a = [0.1,0.2,0.5,0.2]$$a = [0.1,0.2,0.5,0.2]$ and visualize them:

```
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,

hard_samples = reparametrized_samples.argmax(dim=1,keepdim=True)
compare_samples(direct_samples, hard_samples, bins=4)
```



Algorithm 1 VAE

This is the first algorithm of the paper for Variational AutoEncoder



We model each pixel value $\in \{0,1\}$ as a sample drawn from a Bernoulli distribution. Through a decoder, the latent random variable z_n associated with an image n is mapped to the success parameters of the Bernoulli distributions associated with the pixels of that image. Our generative model is described as follows:

$ 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 normal distribution, for computational convenience.

Inference model: We infer the posterior distribution of z_n via variational inference. The variational distribution $ q(z_n|x_n) $ is chosen to be multivariate Gaussian with a diagonal covariance matrix. The mean and covariance of this distribution are obtained by applying an encoder to x_n.

$ 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.

```
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}")


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-4

# Number of epoch
epochs = 20
```

```
Using device: mps
```

```
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

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


train_dataset = MNIST(dataset_path, transform=mnist_transform, trai
test_dataset  = MNIST(dataset_path, transform=mnist_transform, trai
test_labels = test_dataset.targets

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_s
test_loader  = DataLoader(dataset=test_dataset,  batch_size=batch_s
```

```
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 = full
        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):
    # it wrap the encoder and the decoder
    def __init__(self, Encoder, Decoder):
        super(Model, self).__init__()
        self.Encoder = Encoder
        self.Decoder = Decoder

    def reparameterization(self, mean, std):
        # 1. Sample standard normal noise epsilon: epsilon ~ N(0, I
        # Use torch.randn_like(mean) to ensure 'eps' has the same s
        # as 'mean' (or 'std').
        eps = torch.randn_like(mean)
        z = mean + eps * std
        return z

    def forward(self, x):
        # insert your code here
        mean, log_var = self.Encoder.forward(x)

        std = torch.sqrt(torch.exp(log_var))

        z = self.reparameterization(mean, std)

        theta = self.Decoder.forward(z)
        return theta, mean, log_var, z
```

```
encoder = Encoder(input_dim=x_dim, hidden_dim=hidden_dim, latent_di
decoder = Decoder(latent_dim=latent_dim, hidden_dim = hidden_dim, o

model = Model(Encoder=encoder, Decoder=decoder)

model.to(device)
```

```
Model(
  (Encoder): Encoder(
    (FC_enc1): Linear(in_features=784, out_features=200, bias=True)
    (FC_enc2): Linear(in_features=200, out_features=200, bias=True)
    (FC_mean): Linear(in_features=200, out_features=20, bias=True)
    (FC_std): Linear(in_features=200, out_features=20, bias=True)
    (LeakyReLU): LeakyReLU(negative_slope=0.2)
  )
  (Decoder): Decoder(
    (FC_dec1): Linear(in_features=20, out_features=200, bias=True)
    (FC_dec2): Linear(in_features=200, out_features=200, bias=True)
    (FC_output): Linear(in_features=200, out_features=784,
bias=True)
    (LeakyReLU): LeakyReLU(negative_slope=0.2)
  )
)
```

```
def loss_function(x, theta, mean, log_var): # should return the los
    eps = 1e-9

    log_px_z = x * torch.log(theta + eps) + (1 - x) * torch.log(1 -
    log_px_z = log_px_z.sum(dim=1)
    recon_term = log_px_z.mean(dim=0)

    var = torch.exp(log_var)
    m_kl_per_sample = 0.5 * torch.sum(1 + log_var -var - mean.pow(2
    m_kl_term = m_kl_per_sample.mean(dim=0)

    elbo = recon_term + m_kl_term
    loss = -elbo
    return loss
```

```
from torch.optim import Adam

print("Start training VAE...")
model.train()

# optimizer
optimizer = Adam(model.parameters(), lr=lr)
pbar = tqdm(range(epochs))
elbo_1 = []
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()

        # insert your code here
        theta, mean, log_var,_= model.forward(x)

        loss = loss_function(x,theta,mean,log_var)
        loss.backward()
        optimizer.step()

        # loss.item() is the mean. Multiply by batch size to get th
        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_1.append(-avg_loss)

print("Finish!!")
```

```
Start training VAE...
Epoch 20/20, Loss: 106.7578, ELBO: -106.7578: 100%|██████████| 20/20
```

Algorithm 2 VAE - Sticking the Landing

```
class Model(nn.Module):
    # it wrap the encoder and the decoder
    def __init__(self, Encoder, Decoder):
        super(Model, self).__init__()
        self.Encoder = Encoder
        self.Decoder = Decoder

    def reparameterization(self, mean, std):
        # 1. Sample standard normal noise epsilon: epsilon ~ N(0, I
        # Use torch.randn_like(mean) to ensure 'eps' has the same s
        # as 'mean' (or 'std').
        eps = torch.randn_like(mean)
        z_pathwise = mean + eps * std

        z_stop_grad = z_pathwise.detach()

        return z_pathwise,z_stop_grad

    def forward(self, x):
        # insert your code here
        mean, log_var = self.Encoder.forward(x)

        std = torch.sqrt(torch.exp(log_var))

        z_pathwise,z_stop_grad = self.reparameterization(mean, std)

        theta = self.Decoder.forward(z_stop_grad)
        return theta, mean, log_var, z_pathwise,z_stop_grad
```

```
encoder = Encoder(input_dim=x_dim, hidden_dim=hidden_dim, latent_di
decoder = Decoder(latent_dim=latent_dim, hidden_dim = hidden_dim, o

model= Model(Encoder=encoder, Decoder=decoder)

model.to(device)
```

```
Model(
  (Encoder): Encoder(
    (FC_enc1): Linear(in_features=784, out_features=200, bias=True)
    (FC_enc2): Linear(in_features=200, out_features=200, bias=True)
    (FC_mean): Linear(in_features=200, out_features=20, bias=True)
    (FC_std): Linear(in_features=200, out_features=20, bias=True)
    (LeakyReLU): LeakyReLU(negative_slope=0.2)
  )
  (Decoder): Decoder(
    (FC_dec1): Linear(in_features=20, out_features=200, bias=True)
    (FC_dec2): Linear(in_features=200, out_features=200, bias=True)
    (FC_output): Linear(in_features=200, out_features=784,
bias=True)
    (LeakyReLU): LeakyReLU(negative_slope=0.2)
  )
)
```

```
def loss_function(x, theta, mean, log_var): # should return the los
    eps = 1e-9

    log_px_z = x * torch.log(theta + eps) + (1 - x) * torch.log(1 -
    log_px_z = log_px_z.sum(dim=1)
    recon_term = log_px_z.mean(dim=0)

    var = torch.exp(log_var)
    m_kl_per_sample = 0.5 * torch.sum(1 + log_var -var - mean.pow(2
    m_kl_term = m_kl_per_sample.mean(dim=0)

    elbo = recon_term + m_kl_term
    loss = -elbo
    return loss
```

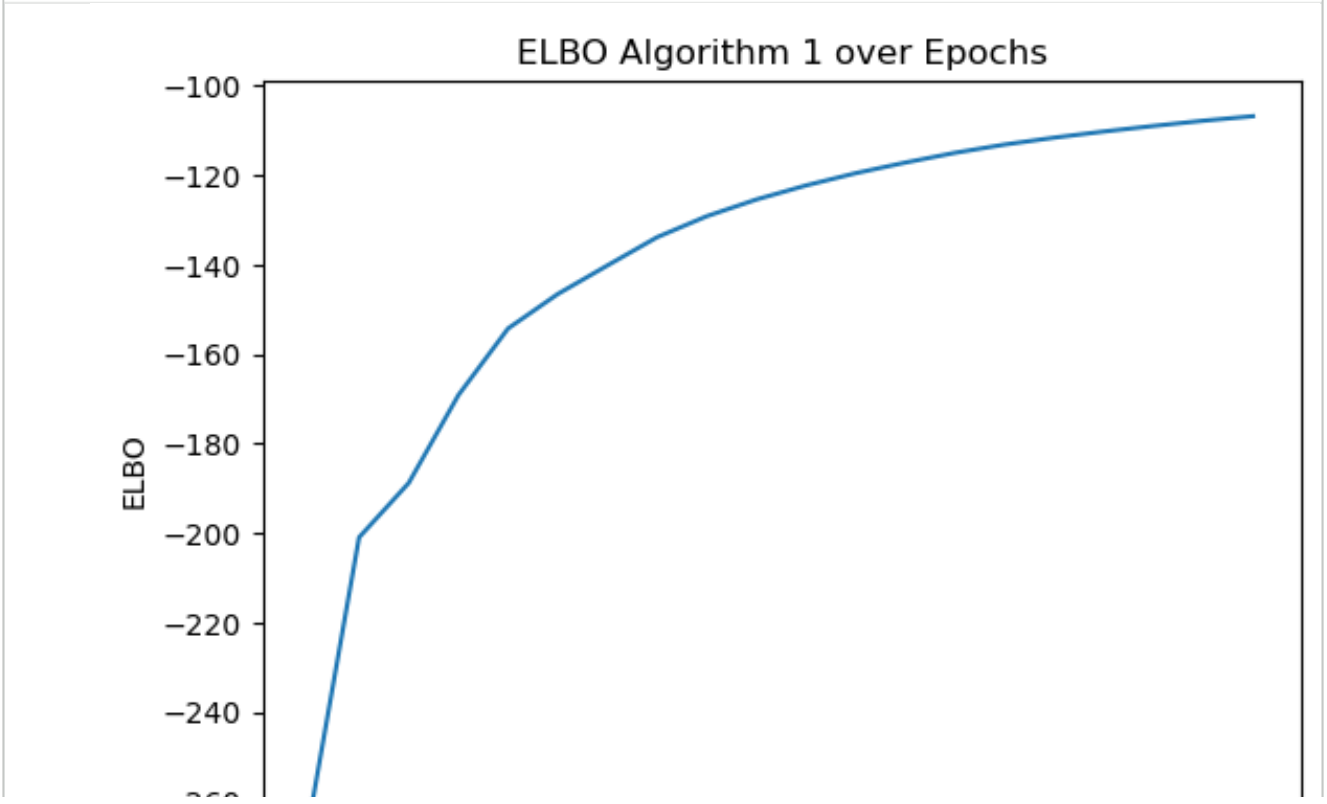
```
from torch.optim import Adam

print("Start training VAE...")
model.train()

# optimizer
optimizer = Adam(model.parameters(), lr=lr)
pbar = tqdm(range(epochs))
elbo_2= []
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()

        # insert your code here
        theta_ng, mean, log_var,_,_= model.forward(x)

        loss = loss_function(x,theta_ng,mean,log_var) # it is compu
        loss.backward()
        optimizer.step()

        # loss.item() is the mean. Multiply by batch size to get th
        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_2.append(-avg_loss)

print("Finish!!")
```

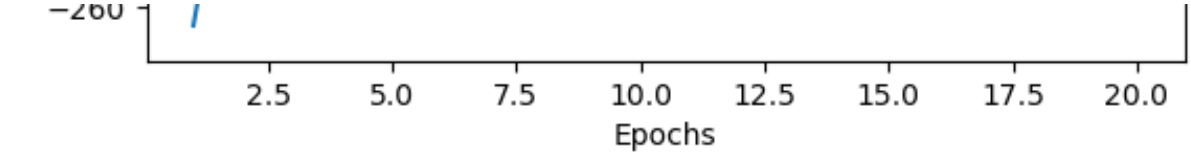
```
Start training VAE...
Epoch 20/20, Loss: 206.3120, ELBO: -206.3120: 100%|██████████| 20/20
```

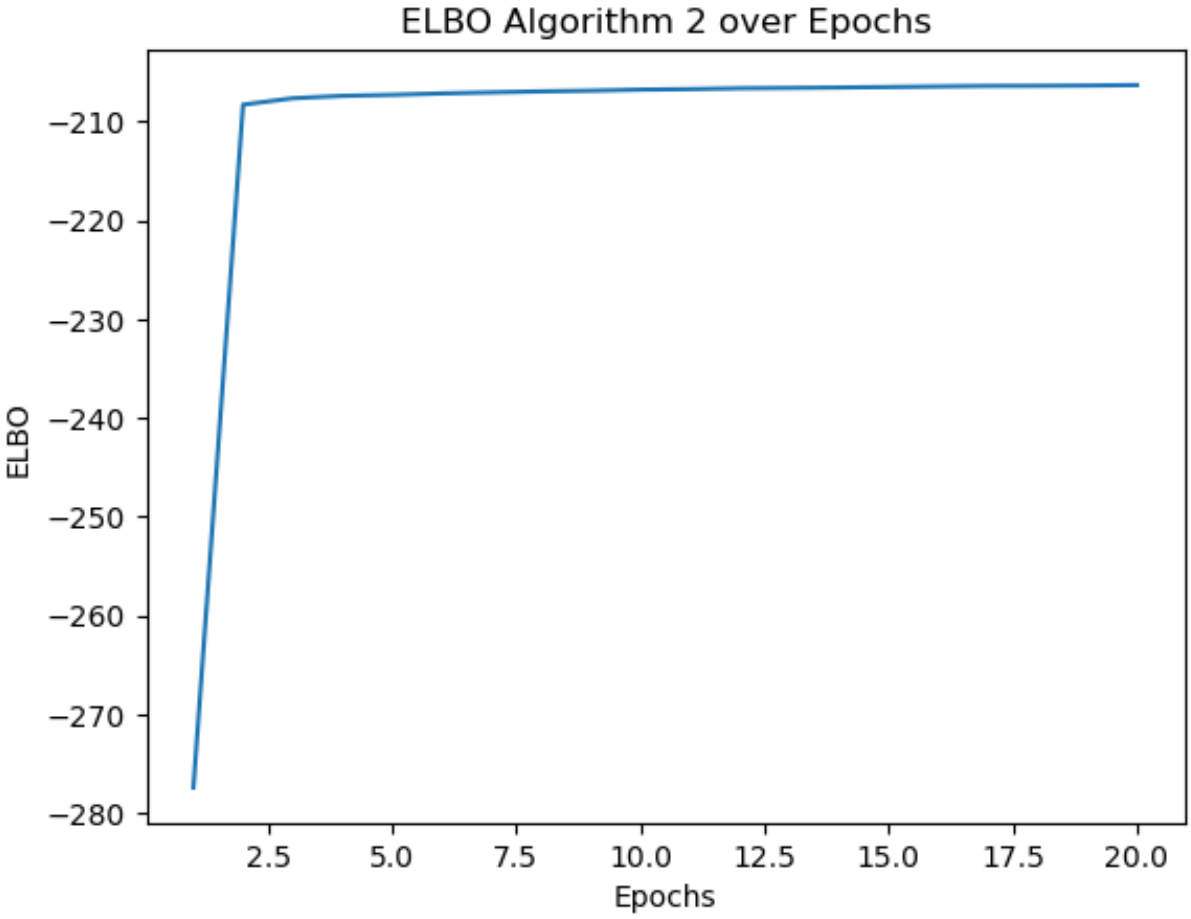
Comparison

Comparing the ELBO over iterations for the standard VAE and the modified VAE of algorithm 2.

```
# Plot ELBO curve
import matplotlib.pyplot as plt
plt.plot(range(1, epochs + 1), elbo_1)
plt.xlabel('Epochs')
plt.ylabel('ELBO')
plt.title('ELBO Algorithm 1 over Epochs')
plt.show()

# Plot ELBO curve
import matplotlib.pyplot as plt
plt.plot(range(1, epochs + 1), elbo_2)
plt.xlabel('Epochs')
plt.ylabel('ELBO')
plt.title('ELBO Algorithm 2 over Epochs')
plt.show()


# Final ELBO
print(f"Final ELBO for Algorithm 1: {elbo_1[-1]}")
print(f"Final ELBO for Algorithm 2: {elbo_2[-1]}")
```








```
Final ELBO for Algorithm 1: -106.75783197428386
Final ELBO for Algorithm 2: -206.31198584798176
```

Fai doppio clic (o premi Invio) per modificare

```
Inizia a programmare o genera codice con l'IA.
```