

```
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  $\tau = 1/\sigma^2$ 
    torch.manual_seed(10)

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

    return D
```

Set  $\mu = 1$ ,  $\tau = 0.5$  and generate a dataset with size  $N=100$ . Plot the histogram for the generated dataset.

```

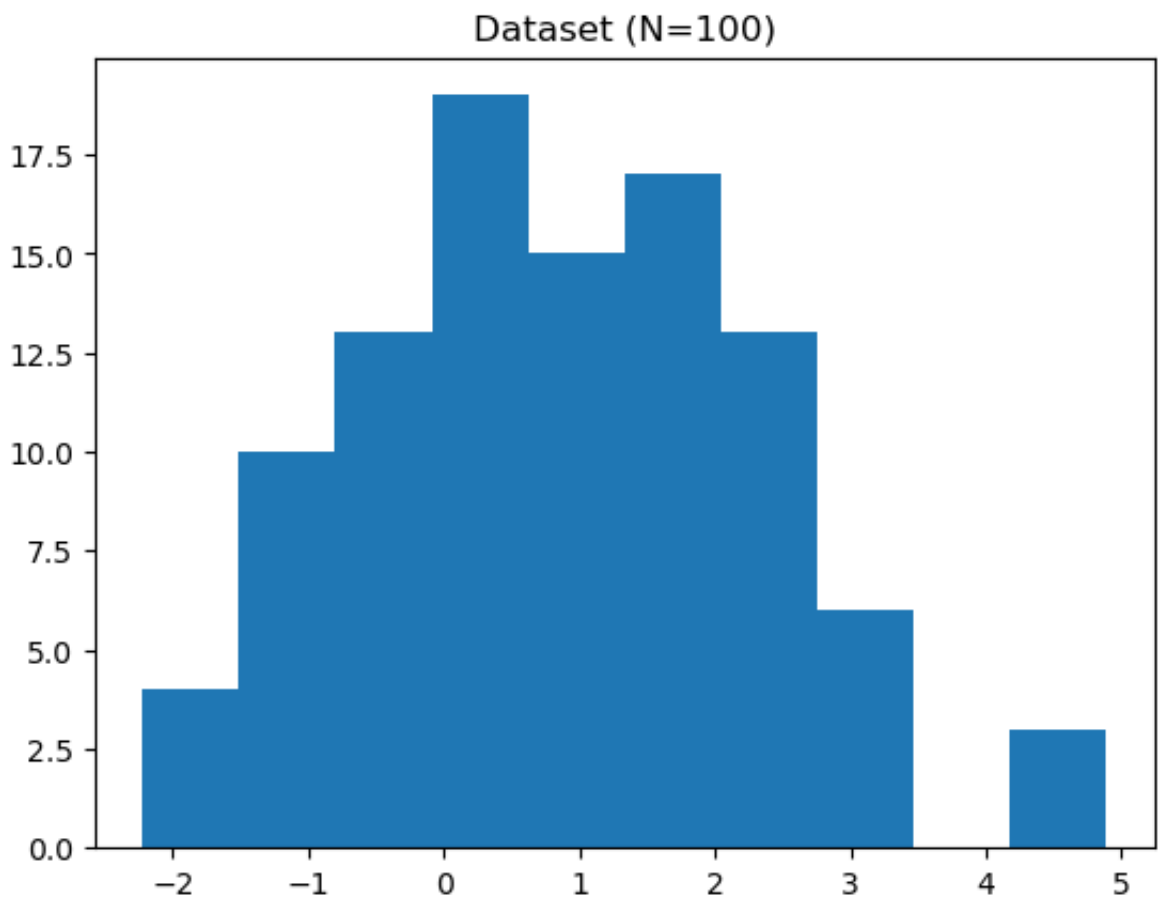
mu = 1
tau = 0.5

dataset = generate_data(mu, tau, 100)

# Visualize 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  $\nabla_{\lambda} \log q(z[s]; \lambda)$ 
            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:  $q(\mu, \tau \mid \lambda) = q(\mu \mid \tau) q(\tau)$ 
    where:
        -  $\mu \mid \tau \sim \text{Normal}(\mu_N, (\lambda_N * \tau)^{-1})$ 
        -  $\tau \sim \text{Gamma}(\alpha_N, \beta_N)$ 

    Parameters:  $\lambda = [\mu_N, \lambda_N, \alpha_N, \beta_N]$ 
    """

    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(torch.tensor(beta_N)),
        ], 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.
         $z[s] \sim q(\mu, \tau \mid \lambda)$ 
    """

    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  $q(\mu, \tau \mid \lambda)$ .
    """
    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  $\nabla_{\lambda} \log q(z; \lambda)$ .

     $z = (\mu^s, \tau^s)$  is a sample from the variational distr
     $\lambda = [\mu_N, \lambda_N, \alpha_N, \beta_N]$  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  $\log q(\mu | \lambda)$  with respect to  $\lambda$ 
    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  $\log q(\tau | \lambda)$  with respect to  $\lambda$ 
    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( $\mu$  |  $\tau$ )
    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( $\tau$ )
    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
```

```
=====
```

