CPSC 532W Assignment 4

Ali Seyfi - 97446637

Here is the link to the repository:
`https://github.com/aliseyfi75/Probabilistic-Programming/tree/master/Assignment_4`

write a black-box variational inference (VI) evaluator following section 4.4 of the book.

1 Code

Show code snippets that demonstrate the completeness and correctness of your BBVI implementation.

1.1 primitives

```
baseprimitives = {
    '+': lambda x: x[0] + x[1],
    '-': lambda x: x[0] - x[1],
    '*': lambda x: x[0] * x[1],
    '/': lambda x: x[0] / x[1],
    '>': lambda x: x[0] > x[1],
    '>=': lambda x: x[0] >= x[1],
    '<': lambda x: x[0] < x[1],
    '<=': lambda x: x[0] <= x[1],
    '==': lambda x: x[0] == x[1],
    '=': lambda x: torch.tensor([1]) if x[0] == x[1] else torch.tensor([0]),
    'and': lambda x: x[0] and x[1],
    'or': lambda x: x[0] or x[1],
    'sqrt': lambda x: torch.sqrt(x[0]),
    'exp': lambda x: torch.exp(x[0]),
    'log': lambda x: torch.log(x[0]),
    'abs': lambda x: torch.abs(x[0]),
    'vector': vector,
    'list': list,
    'get': get,
    'put': put,
    'hash-map': hash_map,
    'first': lambda x: x[0][0],
    'last': lambda x: x[0][-1],
    'nth': lambda x: x[0][int(x[1].item())],
    'second': lambda x: x[0][1],
    'rest': lambda x: x[0][1:],
    'append': append,
    'cons': lambda x: append([x[1],x[0]]),
    'conj': append,
    'mat-add': lambda x: x[0] + x[1],
    'mat-mul': lambda x: torch.matmul(x[0], x[1]),
    'mat-transpose': lambda x: x[0].T,
    'mat-tanh': lambda x: x[0].tanh(),
    'mat-repmat': lambda x: x[0].repeat((int(x[1].item()), int(x[2].item())))
}
```

Listing 1: primitives.py - Base primitives

```python
def vector(x):
    try:
        vector = torch.stack(x)
    except:
        vector = x
    return vector

def list(x):
    try:
        list = torch.stack(x)
    except:
        list = x
    return list

def get(x):
    if type(x[0]) == dict:
        value = x[0][x[1].item()]
    else:
        value = x[0][x[1].long()]
    return value

def put(x):
    if type(x[0]) == dict:
        x[0][x[1].item()] = x[2]
    else:
        x[0][x[1].long()] = x[2]
    return x[0]

def hash_map(x):
    keys = x[::2]
    value = x[1::2]
    new_keys = []
    for key in keys:
        try:
            new_keys.append(key.item())
        except:
            new_keys.append(key)
    result = dict(zip(new_keys, value))
    return result

def append(x):
    first = x[0]
    second = x[1]

    if first == 'vector':
        first = torch.tensor([])
    elif first.dim() == 0:
        first = first.unsqueeze(0)
    if second == 'vector':
        second = torch.tensor([])
    if second.dim() == 0:
        second = second.unsqueeze(0)
    return torch.cat((first, second))
```

Listing 2: primitives.py - Functions

```
import distributions as dist

class Dist:
    def __init__(self, name, distribution, num_par, *par):
        self.name = name
        self.distribution = distribution
        self.num_par = num_par
        self.pars = []
        for i in range(num_par):
            self.pars.append(par[i])

    def sample(self):
        return self.distribution.sample()

    def log_prob(self, c):
        return self.distribution.log_prob(c)

    def parameters(self):
        return self.distribution.Parameters()

    def make_copy_with_grads(self):
        temp_dist = self.distribution
        self.distribution = None
        dist_copy = copy.deepcopy(self)
        self.distribution = temp_dist
        dist_copy.distribution = temp_dist.make_copy_with_grads()
        return dist_copy

class normal(Dist):
    def __init__(self, pars):
        mean = pars[0]
        var = pars[1]
        normal_dist = dist.Normal(mean, var)
        super().__init__('normal', normal_dist, 2, mean, var)

class beta(Dist):
    def __init__(self, pars):
        alpha = pars[0]
        betta = pars[1]
        super().__init__('beta', distributions.Beta(alpha, betta), 2, alpha, betta)

class exponential(Dist):
    def __init__(self, par):
        lamda = par[0]
        super().__init__('exponential', distributions.Exponential(lamda), 1, lamda)

class uniform(Dist):
    def __init__(self, pars):
        a, b = pars[0], pars[1]
        if a > b:
            b = 5
        uniform_dist = dist.Uniform(a, b)
        super().__init__('uniform', uniform_dist, 2, a, b)

class discrete(Dist):
    def __init__(self, pars):
        prob = pars[0]
        discrete_dist = dist.Categorical(prob)
        super().__init__('discrete', discrete_dist, 0)

class bernoulli(Dist):
    def __init__(self, pars):
        p = pars[0]
        bernoulli_dist = dist.Bernoulli(p)
```

```python
        super().__init__('bernoulli', bernoulli_dist, 1, p)

class gamma(Dist):
    def __init__(self, pars):
        alpha, betta = pars[0], pars[1]
        gamma_dist = dist.Gamma(alpha, betta)
        super().__init__('gamma', gamma_dist, 2, alpha, betta)

class dirichlet(Dist):
    def __init__(self, pars):
        dirichlet_dist = dist.Dirichlet(pars[0])
        super().__init__('dirichlet', dirichlet_dist, len(pars), *pars)

class dirac(Dist):
    def __init__(self, value):
        mean = value[0]
        mean = torch.clip(mean, -1e5, 1e5)
        var = torch.tensor(1e-5)
        super().__init__('normal', distributions.Normal(mean, var), 2, mean, var)
```

Listing 3: primitives.py - Distributions

```python
distlist = {
    'normal' : normal,
    'beta' : beta,
    'exponential' : exponential,
    'uniform' : uniform,
    'discrete' : discrete,
    'bernoulli': bernoulli,
    'gamma': gamma,
    'dirichlet': dirichlet,
    'flip': bernoulli,
    'dirac': dirac,
    'uniform-continuous': uniform
}
```

Listing 4: primitives.py - distlist

1.2 evaluation based sampling

This part is same as last assignment.

1.3 topological sort

This part is same as last assignment.

1.4 BBVI evaluator

```
def BBVI_evaluator(order_node, graph, sigma):
    P = graph[1]['P']
    Y = graph[1]['Y']
    Q = sigma['Q']
    G = sigma['G']
    optimizer = sigma['optimizer']
    results = {}

    for node in order_node:
        link_function = P[node][0]

        if link_function == 'sample*':
            d = deterministic_eval(value_subs(P[node][1], results))
            if node not in Q:
                Q[node] = d.make_copy_with_grads()
                optimizer[node] = torch.optim.Adam(Q[node].parameters(), lr=0.01)
            result = Q[node].sample()
            G[node] = grad_log_prob(Q[node], result)
            try:
                sigma_temp = d.log_prob(result) - Q[node].log_prob(result)
                sigma['logW'] += sigma_temp
            except:
                sigma['logW'] += 0

        elif link_function == 'observe*':
            result = torch.tensor(Y[node])
            d = deterministic_eval(value_subs(P[node][1], results))
            sigma_temp = d.log_prob(result)
            sigma['logW'] += sigma_temp

        results[node] = result

    return results, sigma
```

Listing 5: graph_based_sampling.py - BBVI_evaluator

1.5 grad log prob

```
def grad_log_prob(dist, value):
    for param in dist.parameters():
        param = param.clone().detach()
        param.requires_grad = True
    log_prob = dist.log_prob(value)
    log_prob.backward()
    grad = [param.grad for param in dist.parameters()]
    return grad
```

Listing 6: graph_based_sampling.py - grad_log_prob

1.6 BBVI

```
def BBVI(graph, T, L):
    sigma = {'Q':{}, 'optimizer':{}}
    order_node = topological_sort(graph)

    results = []
    log_weights = []
    posteriers = []

    for t in range(T):
        sigma['G'] = {}
        gradients = []
        log_ws = []

        for l in range(L):
            sigma['logW'] = 0
            result, sigma = BBVI_evaluator(order_node, graph, sigma)
            gradients.append(copy.deepcopy(sigma['G']))
            log_ws.append(sigma['logW'])

        if t==0:
            posteriers.append(copy.deepcopy(sigma['Q']['sample2'].parameters()))

        ELBO_gradients(gradients, log_ws, sigma['Q'])

        for optimizer in sigma['optimizer'].values():
            optimizer.step()
            optimizer.zero_grad()

        post_temp = {}
        for q in sigma['Q']:
            post_temp[q] = sigma['Q'][q].parameters().copy()

        posteriers.append(post_temp)
        result_temp = deterministic_eval(value_subs(graph[2], result))
        results.append(result_temp)
        log_weights.append(log_ws[-1])
        wandb.log({'ELBO': torch.mean(torch.stack(log_weights)).detach().numpy()})

    return results, log_weights, posteriers
```

Listing 7: graph_based_sampling.py - BBVI

1.7 infinity skipper

```
def inf_skipper(gradients, log_ws):
    temp_gradients = []
    temp_log_ws = []

    for i in range(len(log_ws)):
        if log_ws[i] == float('-inf'):
            continue
        temp_gradients.append(gradients[i])
        temp_log_ws.append(log_ws[i])

    return temp_gradients, temp_log_ws
```

Listing 8: graph_based_sampling.py - inf_skipper

1.8 ELBO gradient

```python
def ELBO_gradients(gradients, log_ws, posteriors):

    gradients, log_ws = inf_skipper(gradients, log_ws)
    len_grads = len(gradients)

    var_union = list(set([var for grad in gradients for var in grad]))

    Fs = []
    Gs = []
    stack = {}

    for var in var_union:
        gradient_var = gradients[0][var]
        if len(gradient_var[0].shape) > 0 and len(gradient_var[0]) > 1:
            gradient_var = [grad.clone().detach().requires_grad_(True) for grad in gradient_var[0]]
            stack[var] = len(gradient_var)

        len_vars = len(gradient_var)

        G_var = torch.zeros((len_grads, len_vars))
        F_var = torch.zeros((len_grads, len_vars))

        for lg in range(len_grads):
            G_var[lg, :] = torch.stack(gradients[lg][var])
            F_var[lg, :] = G_var[lg, :] * log_ws[lg]
        Gs.append(G_var.detach().numpy())
        Fs.append(F_var.detach().numpy())

    Gs = np.column_stack(Gs)
    Fs = np.column_stack(Fs)

    num = np.sum([np.cov(Fs[:, v], Gs[:, v])[0, 1] for v in range(Gs.shape[1])])
    denum = np.sum([np.var(Gs[:, v]) for v in range(Gs.shape[1])])
    b_hat = 0.
    if not denum == 0. and not np.isnan(num):
        b_hat = num/denum

    counter_1 = 0
    for var in var_union:
        gradient_var = gradients[0][var]
        counter_2 = len(gradient_var)
        if var in stack:
            counter_2 = stack[var]
        g_hat = np.array([np.sum(Fs[:, v] - b_hat * Gs[:, v]) / len_grads for v in range(counter_1, counter_1+counter_2)])
        if var in stack:
            g_hat = [g_hat]
        for i, parameter in enumerate(posteriors[var].parameters()):
            parameter.grad = torch.tensor(-g_hat[i], dtype=parameter.grad.dtype)
        counter_1 += counter_2
    return
```

Listing 9: graph_based_sampling.py - ELBO_gradient

2 Results

2.1 Task 1

$T = 10^4$ and $L = 50$ for this task.
Time of drawing samples: **410.81 seconds**
Posterior expected value of mu is: **7.3007**
Parameters of the posterior distribution of mu: $mu = 7.2742$ and $\sigma = 0.4931$.

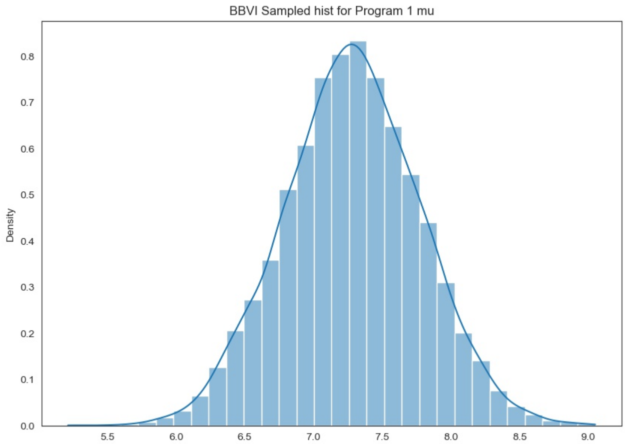

Figure 1: Histogram of posterior distribution of mu

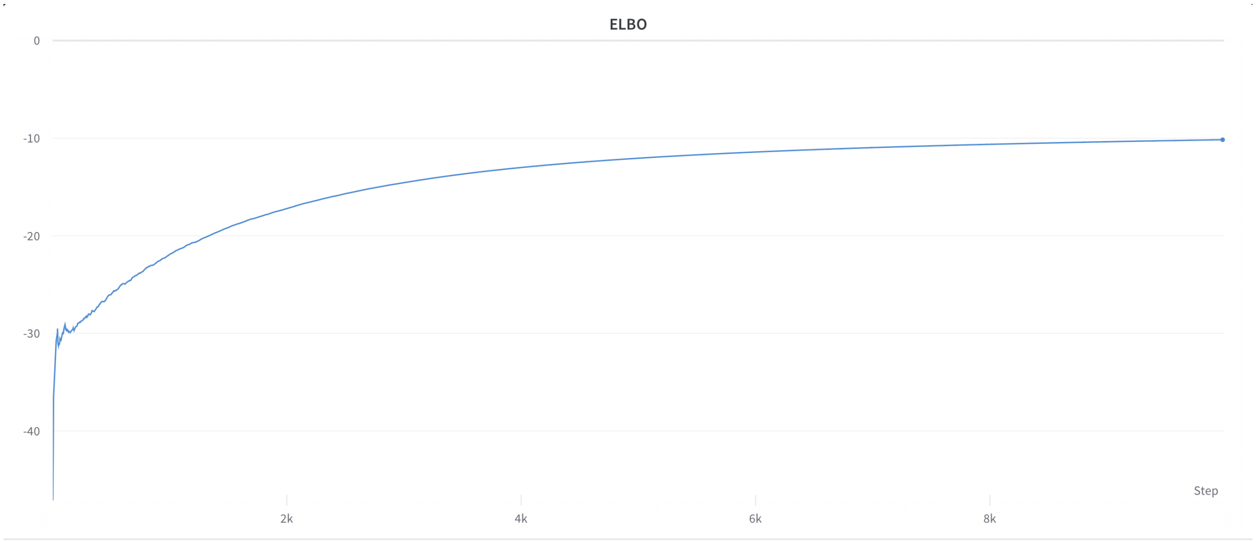

Figure 2: ELBO in task 1

2.2 Task 2

$T = 5 * 10^3$ and $L = 50$ for this task.
Time of drawing samples: **510.36 seconds**
Posterior mean of slope is: **2.1169**
Posterior mean of bias is: **-0.4039**

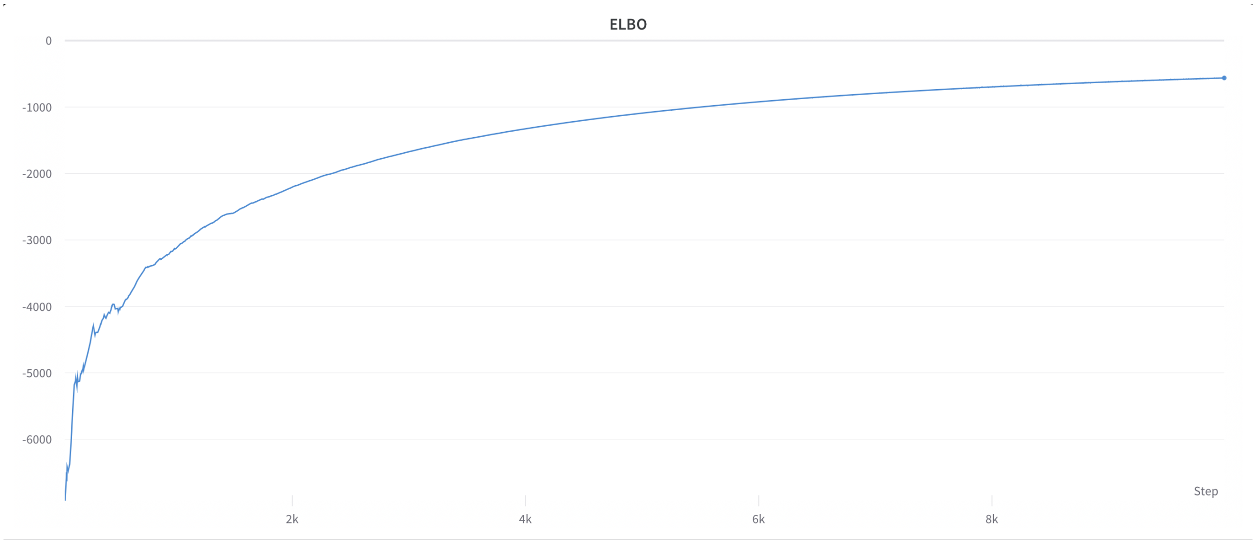

Figure 3: ELBO in task 2

2.3 Task 3

$T = 2 * 10^3$ and $L = 50$ for this task.
Time of drawing samples: **581.53 seconds**
Posterior mean of probability that the first and second datapoint are in the same cluster is: **0.7743**
Should talk about the mode-seeking behaviour of VI

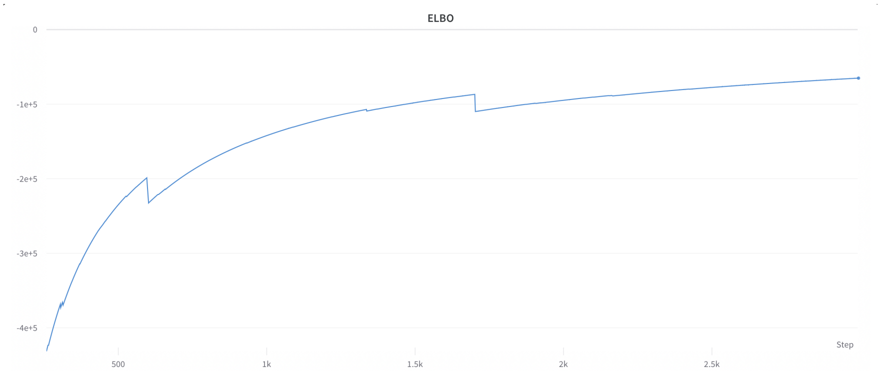

Figure 4: ELBO in task 3

2.4 Task 4

Time of drawing samples: **595.87 seconds**

Figure 5: Posterior distribution of b_0

Figure 6: Posterior distribution of w_0

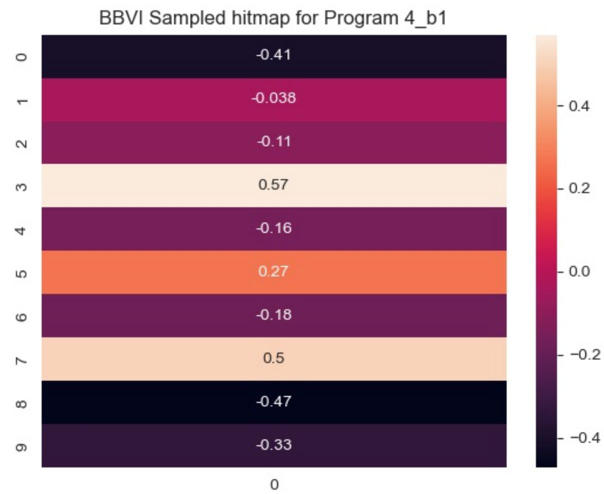

Figure 7: Posterior distribution of b_1

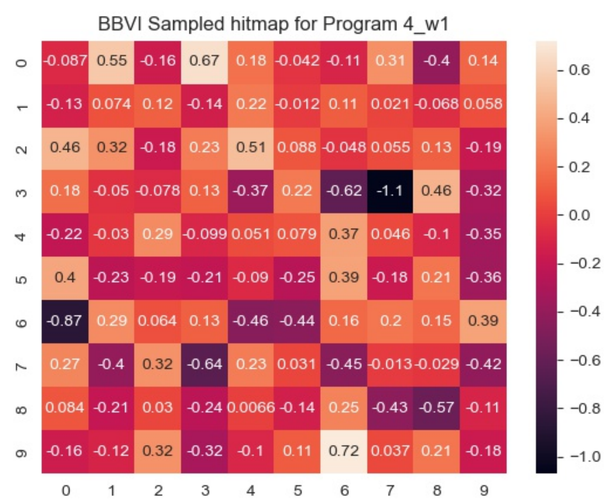

Figure 8: Posterior distribution of w_1

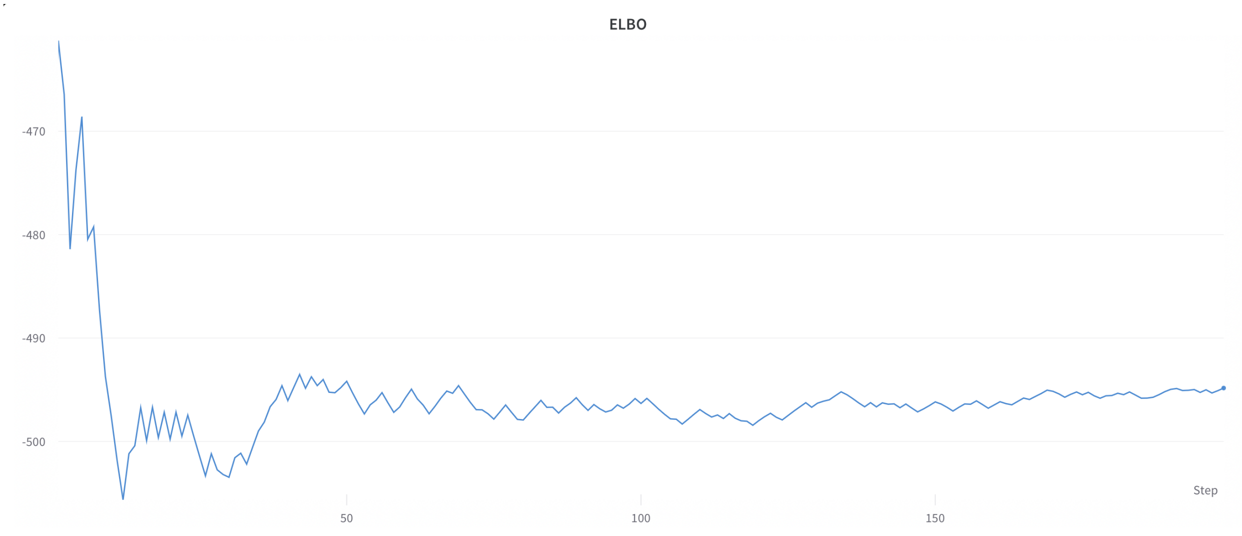

Figure 9: ELBO in task 4

should compare and contrast mean-field BBVI to parameter estimation via gradient descent.

2.5 Task 5

$T = 2 * 10^3$ and $L = 25$ for this task.
Time of drawing samples: **45.19 seconds**
learned variational distribution for s: **Uniform(0.9024, 1.7703)**

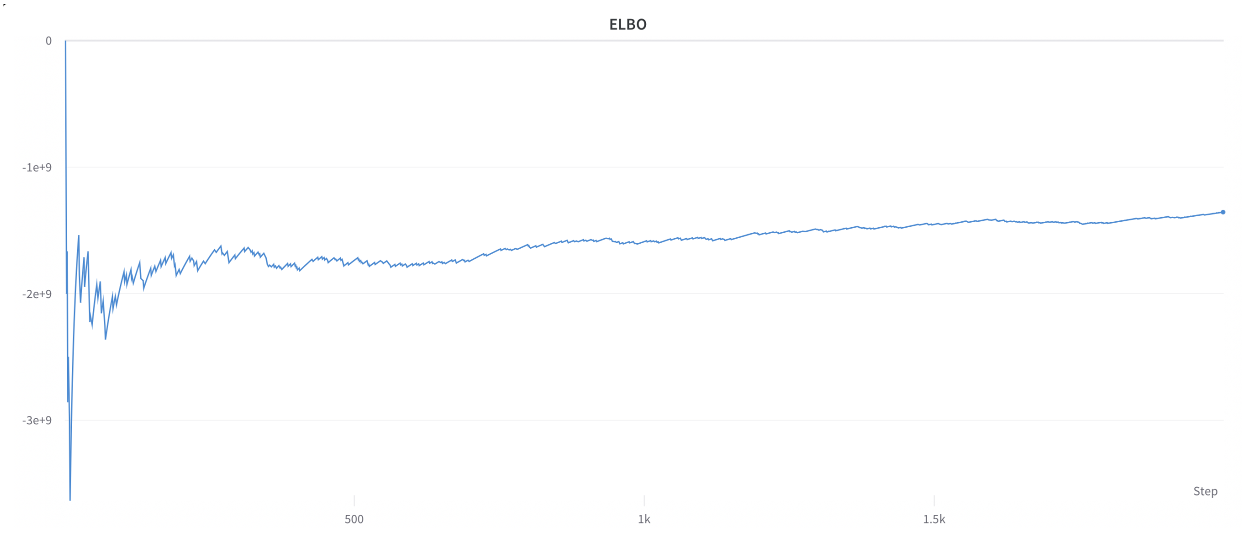

Figure 10: ELBO in task 5