HMC_Neural Network

$$p(\tau|\theta, \mathcal{D}) \propto \tau^{\alpha_0+N/2-1} \exp(-\tau(\frac{1}{2}(\mathbf{y}-\mathbf{X}\theta)^T(\mathbf{y}-\mathbf{X}\theta)+\beta_0))$$
$$\alpha - 1 = \alpha_0 + N/2 - 1 \Rightarrow \alpha = \alpha_0 + N/2$$
$$\beta = \beta_0 + \frac{1}{2}(\mathbf{y}-\mathbf{X}\theta)^T(\mathbf{y}-\mathbf{X}\theta)$$

```
# Defining the sampler for tau from a random.gamma distribution
def create_p_tau(X, y, alpha_0, beta_0, N):
alpha = alpha_0 + N / 2

    def p_tau(tau, nn):
        z = y - nn(X)
        beta = (beta_0 + 0.5 * z.T @ z).detach().numpy().item()
        return np.random.gamma(alpha, 1 / beta)

return p_tau
```

$$p(\theta|\tau, \mathcal{D}) \propto \exp(-\frac{1}{2}(\tau(\mathbf{y}-\mathbf{X}\theta)^T(\mathbf{y}-\mathbf{X}\theta)+\frac{1}{\sigma_0^2}\theta^T\theta))$$
$$\log p\left(\theta \mid \sigma^2, D\right) \propto -\frac{1}{2\sigma^2}(y-f_\theta(x))^T\left(y-f_\theta(x)\right)-\frac{1}{2\sigma_0^2}\theta^T\theta$$

where f(x) is a the neural network and is a vector that contains all parameters of the neural network.

```
# Defining the Potential Energy function which is calculated from the eq
uation P(.) = exp(-U(.))
# U ~ log P(.)
# P(.) = exp(z - reg)
# z acts as an auxilary variable
# reg acts as an regularization factor which is calculated from the para
meters of the the neural network
    def U(nn, tau):
        z = -0.5 * tau * (y - nn(X)).T @ (y - nn(X))
        reg = (collect_params(nn) @ collect_params(nn).T) / (2 * sigma_0
** 2)
        return z - reg
```

Gradient calculations: This can be done calling .backward() on the density in step 2

```
    # Helper function for constructing gradient of potential energy function
    def create_grad_U(X, y, sigma_0):
        def grad_U(nn, tau, U):
            PE = U(mlp, tau) # Constructing the potential Energy to with the
    helper function of U
            nn.zero_grad()   # Setting the gradients to zero before starting
    to do backpropagation
            PE.backward()    # Calling the .backward() method to calculate t
    he gradient of the Potential Energy
            grad = collect_grads(nn) # Forming a row vector with all the gra
    dients form the neural network
            return grad

        return grad_U
```

In [61]:

```
# Importing libraries
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
```

In [62]:

```
class Simulator:

    def __init__(self, w, b, sigma, N, design_range=(-10,10)):
        self.w = w
        self.b = b
        self.theta = np.expand_dims(np.concatenate([w, [b]], axis=0), axis=1)
        self.sigma = sigma
        self.N = N
        self.design_range = design_range
        self.X = None
        self.y = None
        self.y_mean = None

    def run(self):
        designs = np.random.uniform(self.design_range[0], self.design_range[1], size
        self.X = np.concatenate([designs, np.ones((self.N, 1))], axis=1)
        self.y_mean = (self.X @ self.theta).squeeze()
        self.y = np.random.multivariate_normal(mean=self.y_mean, cov=np.diag([self.s

    def plot(self):
        x = self.X[:, 0]
        plt.scatter(x, self.y, label="data")
        x_dense = np.linspace(self.design_range[0], self.design_range[1], 100)
        y_dense = x_dense * self.w[0] + self.b
        plt.plot(x_dense, y_dense, label="y mean")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.legend()
        plt.title("Simulated data, N="+str(self.N))
        plt.show()
```

In [63]:

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        #self.layer_1 = nn.Linear(d + 1, 16)
        self.output_layer = nn.Linear(d+1, 1)
    def forward(self, x):
        #x = torch.sigmoid(self.layer_1(x))
        return self.output_layer(x)
```

In [64]:

```
# Helper function for collecting nn gradient into a vector
def collect_grads(model):
    return torch.cat([p.grad.data.view(1, -1) for p in model.parameters()], dim=-1)

# Helper function for computing sizes of all nn parameters
def get_param_sizes(model):
    return [p.reshape(-1).size()[0] for p in model.parameters()]

# Helper function for writing the updated weights
def update_params(new_params, model, param_sizes):
    start_index = 0
    for i, p in enumerate(model.parameters()):
        end_index = start_index + param_sizes[i]
        source_tensor = new_params[:, start_index:end_index].reshape(p.shape)
        p.data = source_tensor
        start_index = end_index

def collect_params(model):
    return torch.cat([p.data.view(1, -1) for p in model.parameters()], dim=-1)
```

In [65]:

```
# True weight(s)
w = np.array([1.5, -1.0, 0.7])

# Input dimensionality
d = w.size

# True intercept
b = 0.5

# True standard deviation
sigma = 0.5

# Number of data points
N = 100

# Defines range of inputs x
design_range = (-1.0, 1.0)

# Simulate
simulator = Simulator(w, b, sigma, N, design_range)
simulator.run()

X = simulator.X
y = simulator.y
```

In [89]:

```
# Helper function for constructing potential energy function

# Defining the Potential Energy function which is calculated from the equation P(.)
# U ~ log P(.)
# P(.) = exp(z - reg)
# z acts as an auxilary variable
# reg acts as an regularization factor which is calculated from the parameters of th
def create_U(X, y, sigma_0):

    def U(nn, tau):
        z = -0.5 * tau * (y - nn(X)).T @ (y - nn(X))
        reg = (collect_params(nn) @ collect_params(nn).T) / (2 * sigma_0 ** 2)
        return z - reg

    return U

# Helper function for constructing gradient of potential energy function
def create_grad_U(X, y, sigma_0):

    def grad_U(nn, tau, U):
        PE = U(mlp, tau) # Constructing the potential Energy to with the helper func
        nn.zero_grad()   # Setting the gradients to zero before starting to do backp
        PE.backward()    # Calling the .backward() method to calculate the gradient
        grad = collect_grads(nn) # Forming a row vector with all the gradients form
        return grad

    return grad_U


# Helper function for constructing conditional distribution for tau
# Defining the sampler for tau from a random.gamma distribution
def create_p_tau(X, y, alpha_0, beta_0, N):
    alpha = alpha_0 + N / 2

    def p_tau(tau, nn):
        z = y - nn(X)
        beta = (beta_0 + 0.5 * z.T @ z).detach().numpy().item()
        return np.random.gamma(alpha, 1 / beta)

    return p_tau
```

In [67]:

```
class ODESolver:

    def __init__(self, grad_U, epsilon, L, m):
        self.grad_U = grad_U
        self.epsilon = epsilon
        self.L = L
        self.m = m

    def dq(self, p):
        return p / self.m

    def euler(self, q, p, tau):
        for i in range(self.L):
            p_new = p - epsilon * self.grad_U(q, tau)
            q = q + self.epsilon * self.dq(p)
            p = p_new
        return q, p

    def modified_euler(self, q, p, tau):
        for i in range(self.L):
            p = p - self.epsilon * self.grad_U(q, tau)
            q = q + self.epsilon * self.dq(p)
        return q, p

    def leapfrog(self, q, p, tau, nn):
        p = p - self.epsilon * self.grad_U(nn, tau, U) / 2
        for i in range(self.L):
            q = q + self.epsilon * self.dq(p)
            if i != self.L - 1:
                p = p - self.epsilon * self.grad_U(nn, tau, U)
        p = p - self.epsilon * self.grad_U(nn, tau, U) / 2
        p = -p
        return q, p
```

In [68]:

```
class HMC:

    def __init__(self, U, grad_U, p_tau, m, num_params, solver):
        self.U = U
        self.grad_U = grad_U
        self.p_tau = p_tau
        self.m = m
        self.num_params = num_params
        self.solver = solver
        self.samples = None

    def step(self, q, tau, model, param_sizes):
        p = np.random.multivariate_normal(np.zeros(self.num_params), np.diag([1] * s
        p = torch.tensor(p, dtype=torch.float32)
        q_proposed, p_proposed = self.solver.leapfrog(q, p, tau, model)
        U = self.U(model, tau)
        K = ((p ** 2) / (2 * self.m)).sum()

        # Write proposed parameters into neural network
        update_params(q_proposed, model, param_sizes)

        # Evaluate potential energy with proposed parameters
        U_proposed = self.U(model, tau)

        K_proposed = ((p_proposed ** 2) / (2 * self.m)).sum()

        log_energy_ratio = (U - U_proposed + K - K_proposed).detach().numpy()

        u = np.random.uniform()
        if u < np.exp(log_energy_ratio):
            return q_proposed, 1
        else:
            return q, 0

    def run(self, model, param_sizes, theta, tau, num_samples=1000, burn_in=100):

        N = num_samples - burn_in
        self.samples = np.zeros((N, self.num_params + 1))
        acceptances = []

        for i in range(num_samples):
            tau = self.p_tau(tau, model)
            theta, acceptance = self.step(theta, tau, model, param_sizes)
            update_params(theta, model, param_sizes)

            if i >= burn_in:
                j = i - burn_in
                self.samples[j, 0] = 1 / np.sqrt(tau)
                self.samples[j, 1:] = theta.detach().numpy()
                acceptances.append(acceptance)

        return self.samples, acceptances

    def plot(self, true_params):

        num_params = self.d + 2
        plt.figure(figsize=(15, 5 * num_params))
        param_names = ["sigma"]
```

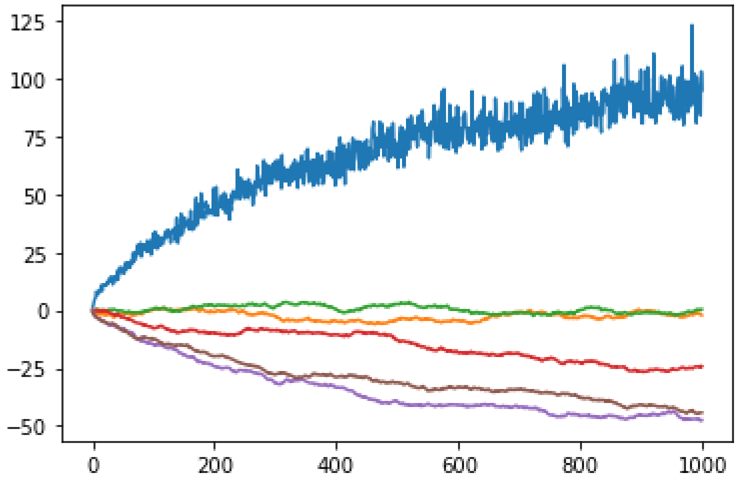
```
        for i in range(num_params - 2):
            param_names.append("weight " + str(i+1))
        param_names.append("intercept")

        for i in range(num_params):
            true_val = true_params[i]
            samples = self.samples[:, i]
            plt.subplot(num_params,2,i*2+1)
            y, _, _ = plt.hist(samples, bins=100, label="samples")
            max_y = int(np.max(y))
            plt.plot([true_val] * max_y, range(max_y), c="r", label="true value")
            plt.title("Histogram for {}".format(param_names[i]))
            plt.legend()
            plt.subplot(num_params,2,i*2+2)
            plt.plot(samples, label="chain")
            plt.plot([0, samples.size], [true_val, true_val], c="r", label="true val
            plt.title("Trace plot for {}".format(param_names[i]))
            plt.legend()

        plt.show()

    def evaluate(self, test_simulator):

        "Computes MSE between test data and the predictions of the model"

        test_covariates = test_simulator.X
        mean_weights = self.samples.mean(axis=0)[1:]
        predictions = test_covariates @ mean_weights
        mse = ((predictions - test_simulator.y) ** 2).mean()
        return mse
```

In [69]:

```
# Define priors
sigma_0 = 1000
alpha_0 = 0.001
beta_0 = 0.001

# Create helper functions
X_tensor = torch.tensor(X, dtype=torch.float32, requires_grad=False)
y_tensor = torch.tensor(y, dtype=torch.float32, requires_grad=False).view(-1, 1)
U = create_U(X_tensor, y_tensor, sigma_0=1000)
grad_U = create_grad_U(X_tensor, y_tensor, sigma_0=1000)
p_tau = create_p_tau(X_tensor, y_tensor, alpha_0, beta_0, N)

mlp = MLP()
param_sizes = get_param_sizes(mlp)
num_params = sum(param_sizes)

# Define HMC parameters

# Step size
epsilon = 0.01

# Number of steps in the proposal
L = 20

# Mass parameter
m = 1.0

solver = ODESolver(grad_U, epsilon, L, m)

chain = HMC(U, grad_U, p_tau, m, num_params, solver)

# Initialize chain
#theta = np.zeros(d+1)
theta = collect_params(mlp)
tau = 1.0

# Run chain
samples, acceptances = chain.run(mlp, param_sizes, theta, tau, num_samples=1000, bur
```

In [87]:

```
plt.plot(samples[:, 0])
plt.plot(samples[:, 1])
plt.plot(samples[:, 2])
plt.plot(samples[:, 3])
plt.plot(samples[:, 4])
plt.plot(samples[:, 5])
plt.show()
```



In [11]:

```
samples.shape
```

Out[11]:

```
(1000, 6)
```

In [ ]:

In [ ]:

In [ ]: