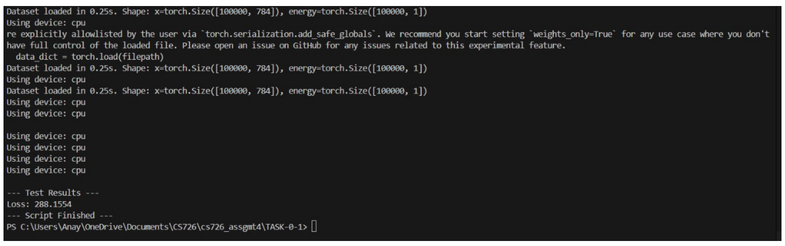
CS726 Assignment 4

Kushal Jain   Nimay Shah   Anay Arora

March 2025

1 Task 0

1.1 Output generated by get_results.py



```
Dataset loaded in 0.25s. Shape: x=torch.Size([100000, 784]), energy=torch.Size([100000, 1])
Using device: cpu
re explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't
have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  data_dict = torch.load(filepath)
Dataset loaded in 0.25s. Shape: x=torch.Size([100000, 784]), energy=torch.Size([100000, 1])
Using device: cpu
Dataset loaded in 0.25s. Shape: x=torch.Size([100000, 784]), energy=torch.Size([100000, 1])
Using device: cpu
Using device: cpu

Using device: cpu
Using device: cpu
Using device: cpu
Using device: cpu

--- Test Results ---
Loss: 288.1554
--- Script Finished ---
PS C:\Users\Anay\OneDrive\Documents\CS726\cs726_assgmt4\TASK-0-1> 
```

Figure 1: Output generated by task 0

2 Task 1: MCMC Sampling Implementation

2.1 Algo-1: MALA (Metropolis-adjusted Langevin Algorithm)

This sampler uses Langevin dynamics combined with a Metropolis-Hastings acceptance step. The proposal step involves computing the gradient of the energy and perturbing the state with Gaussian noise.

```
gt = self.compute_gradient(X)
xi = torch.randn_like(X) * np.sqrt(self.tau)
X_prime = X - (self.tau / 2) * gt + xi
gt_prime = self.compute_gradient(X_prime)
```

Listing 1: MALA Gradient and Proposal Step

Here, a candidate sample X' is generated using Langevin dynamics:

$$X' = X - \frac{\tau}{2}\nabla E(X) + \xi, \quad \xi \sim \mathcal{N}(0, \tau I)$$

The gradient is computed both at the current and proposed positions.

```
mu_X = X - (self.tau / 2) * gt
mu_X_prime = X_prime - (self.tau / 2) * gt_prime

log_q_X_given_Xprime = - (1/(4 * self.tau)) * torch.norm(X -
    mu_X_prime, p=2)**2
log_q_Xprime_given_X = - (1/(4 * self.tau)) * torch.norm(
    X_prime - mu_X, p=2)**2

log_alpha = (energy_X - energy_Xprime) + (
    log_q_X_given_Xprime - log_q_Xprime_given_X)
```

Listing 2: MALA Acceptance Probability Calculation

To satisfy detailed balance, the acceptance probability α is computed using the energy difference and transition probabilities.

The proposal is accepted if:

$$\log u < \log \alpha \quad \text{where } u \sim \mathcal{U}(0, 1)$$

```
if log_alpha >= 0 or np.log(np.random.rand()) < log_alpha:
    X = X_prime.detach()
```

Listing 3: MALA Accept or Reject

The proposed sample is accepted with probability $\min(1, e^{\log \alpha})$. If rejected, the previous sample is retained.

2.2 Algo-2: ULA (Unadjusted Langevin Algorithm)

This algorithm omits the Metropolis correction, thus reducing computational cost at the expense of bias in the stationary distribution.

```
gt = self.compute_gradient(X)
xi = torch.randn_like(X) * np.sqrt(self.tau)
X = X - (self.tau / 2) * gt + xi
```

Listing 4: ULA Update Rule

The update rule is similar to MALA's proposal step, but always accepted:

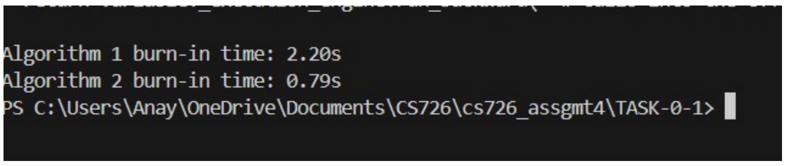
$$X_{t+1} = X_t - \frac{\tau}{2}\nabla E(X_t) + \xi$$

Because there's no acceptance/rejection step, ULA can be more efficient per iteration but less accurate for complex distributions.

Task 1 - Results

Burn-in Time of each algorithm:

- Algorithm 1: 2.2 sec
- Algorithm 2: 0.79 sec



```
Algorithm 1 burn-in time: 2.20s
Algorithm 2 burn-in time: 0.79s
PS C:\Users\Anay\OneDrive\Documents\CS726\cs726_assgmt4\TASK-0-1>
```

Figure 2: Burn-In time

The samples generated by the two algorithms were projected into 2 dimensions using PCA and visualized as follows:

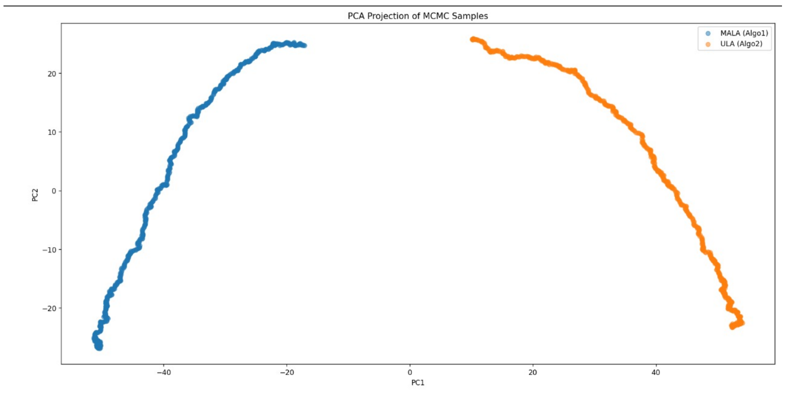


Figure 3: Visualized samples

3 Task 2 - Approximating a Black Box Function using Gaussian Processes

Implementation

```python
def branin_hoo(x):
    x1, x2 = x
    b = 5.1 / (4 * np.pi**2)
    c = 5 / np.pi
    d = 1 / (8 * np.pi)
    return (x2 - b * x1**2 + c * x1 - 6)**2 + 10 * (1 - d) * np.cos(x1) + 10
```

This function defines the Branin-Hoo test function. It unpacks a 2D input vector into `x1` and `x2`. Constants `b`, `c`, and `d` are derived from the original function formula. The final line computes the function value using these constants and the inputs.

```python
def rbf_kernel(x1, x2, length_scale=1.0, sigma_f=1.0):
    sq_dist = np.sum(x1**2,1).reshape(-1,1) + np.sum(x2**2,1) - 2*np.dot(x1,x2.T)
    return (sigma_f**2) * (np.exp(-0.5 * sq_dist / length_scale**2))
```

This defines the Radial Basis Function (RBF) kernel. It calculates the squared Euclidean distance between all pairs of vectors from `x1` and `x2`, and then computes the RBF kernel using the length scale and output scale.

```python
def matern_kernel(x1, x2, length_scale=1.0, sigma_f=1.0, nu=1.5):
    x1_sq = np.sum(x1**2, axis=1).reshape(-1, 1)
    x2_sq = np.sum(x2**2, axis=1).reshape(1, -1)
    sq_dist = x1_sq + x2_sq - 2 * np.dot(x1, x2.T)
    m_dist = np.sqrt(np.maximum(sq_dist, 1e-12))
    return sigma_f**2 * (1 + (np.sqrt(3) * m_dist / length_scale)) * \
           np.exp(-(np.sqrt(3) * m_dist / length_scale))
```

This is the Matérn kernel function with $\nu = 1.5$. It first computes the squared pairwise distances between vectors and ensures numerical stability with a small epsilon. Then it uses the Matérn kernel formula to compute similarity scores.

```python
def rational_quadratic_kernel(x1, x2, length_scale=1.0, sigma_f=1.0, alpha=1.0):
    sq_dist = np.sum(x1**2,1).reshape(-1,1) + np.sum(x2**2,1) - 2*np.dot(x1,x2.T)
    return (sigma_f**2) * ((1 + sq_dist/(2*alpha*length_scale**2))**(-alpha))
```

This function computes the Rational Quadratic kernel. It calculates pairwise squared distances and plugs them into the formula, which behaves like a mixture of RBF kernels with varying length scales, controlled by `alpha`.

```python
def log_marginal_likelihood(x_train, y_train, kernel_func, length_scale, sigma_f, noise=1e-4):
    K = kernel_func(x_train, x_train, length_scale, sigma_f) + (noise**2) * np.eye(len(x_train))
    L = np.linalg.cholesky(K)
    z = np.linalg.solve(L, y_train)
    log_like = -0.5 * np.dot(z, z) - np.sum(np.log(np.diagonal(L))) - 0.5 * len(x_train) * np.log(2*np.pi)
    return log_like
```

This function computes the log marginal likelihood for a Gaussian Process. It starts by creating the kernel matrix `K` and adds Gaussian noise. Then it performs Cholesky decomposition for numerical stability, solves the system, and uses the resulting vector to compute the final log-likelihood value.

```python
def expected_improvement(mu, sigma, y_best, xi=0.01):
    z = (mu - y_best - xi) / sigma
    small_phi = np.exp(-0.5 * z**2) / np.sqrt(2*np.pi)
    big_phi = 1 / (1 + np.exp(-1.702 * z))
    return (mu - y_best - xi) * big_phi + sigma * small_phi
```

This function computes the Expected Improvement (EI) at predicted points. `mu` is the predicted mean, `sigma` is the standard deviation, and `y_best` is the best observed value so far. The function computes a standardized value `z`, calculates approximations of the PDF (`small_phi`) and CDF (`big_phi`), and combines them to produce the EI value, encouraging exploration and exploitation.

```python
def probability_of_improvement(mu, sigma, y_best, xi=0.01):
    z = (mu - y_best - xi) / sigma
    return 1 / (1 + np.exp(-1.702 * z))
```

This function computes the Probability of Improvement (PI). It also computes a standardized `z` value and applies a logistic sigmoid function (approximating the Gaussian CDF) to return the probability that the prediction improves over the best seen value so far, with a margin `xi`.

Performance Comparison of Various Kernels

We see the standard deviation plots for the same number of samples(50) with the same acquistion strategy but for different kernels.

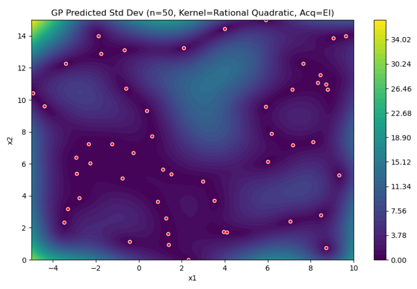


Figure 4: GP standard dev with kernel = rational quadratic

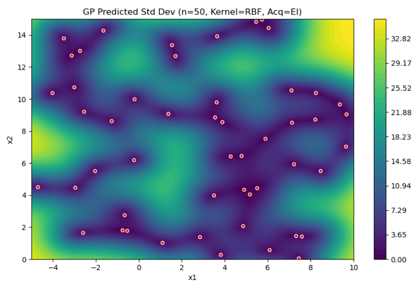


Figure 5: GP standard dev with kernel = rbf

Kernel	**Smoothness Assumption**	**Uncertainty Spread**	**Handles Sharp Features**	**Best Use Case**
Matérn ($\nu = 1.5$)	Moderate	Localized	Yes	Non-smooth functions, moderate noise
RBF	High	Diffuse	No	Smooth functions, low noise
Rational Quadratic	Varying	Balanced	Yes	Mixed smooth/rough regions

Table 1: Comparison of Gaussian Process kernels

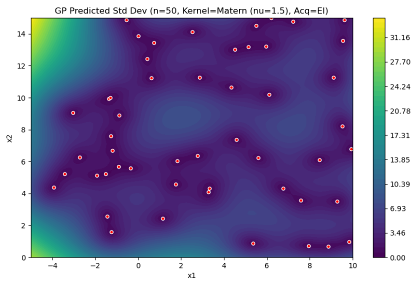


Figure 6: GP standard dev with kernel = matern

Comparison of effect of number of samples

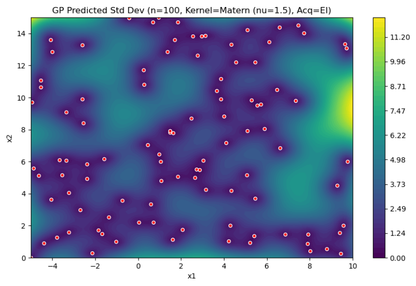


Figure 7: N=100

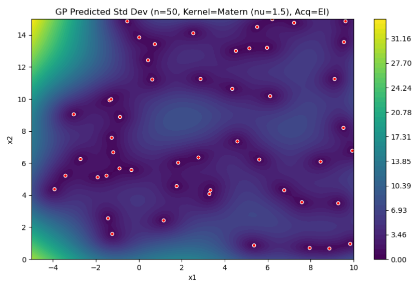


Figure 8: N=50

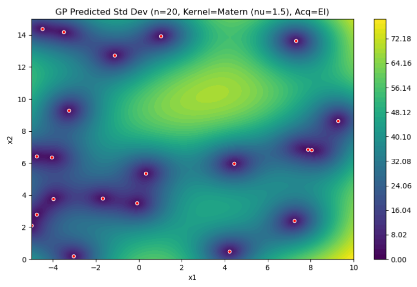


Figure 9: N=20

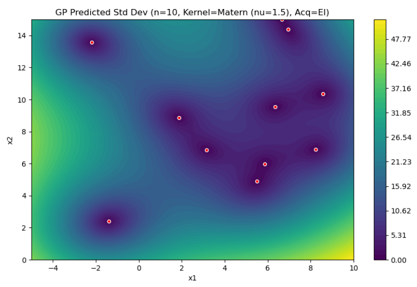


Figure 10: N=10

We select the same kernel, the same acquistion strategy and then see the GP standard deviation for those across the different number of samples. For low n (n=10), Standard deviation is high across the entire input space, with little dip near data points. GP is very uncertain due to sparse training data. As n increases we narrower standard deviation bands around the training data. GP learns the function better. The model is more confident across a larger fraction of the input space. As n=100, the uncertainty is the lowest among all plots. There is clear confidence in prediction across the domain, except for extrapolated boundaries.
All the observations have been drawn from the uncertainty estimate plots.

- **Random Sampling:**
  - Selects points uniformly at random, independent of the GP model.
  - We can see that it has pure exploration; slow and inefficient at improving the GP model.s
- **Expected Improvement (EI):**
  - Balances exploration and exploitation.
  - Targets regions with high expected gain over current best.
  - Improves the GP model effectively near optima.
- **Probability of Improvement (PI):**
  - It is more exploitative than EI.
  - Focuses on areas likely to improve current best value.
  - Can get stuck in local optima; less exploration.

4 Work Contribution

- Anay(22B3939) : Wrote the code for task 1 and 2
- Kushal(22B4514) : Wrote the code for task 2 and report
- Nimay(22B1232) : Wrote