Bayesian Optimization: A Gentle Introduction with Applications

Hubert Witkos, Soujan Niroula, Yuqian Wang, Yongliang Yu

May 13, 2025

Abstract

Bayesian Optimization is a useful technique for optimizing functions which are expensive and/or impossible to evaluate and do not have computable derivatives. This expensive to compute function is modeled by a proxy function which may contain the users prior knowledge of the problem. Then an acquisition function, which balances exploration with exploitation, is utilized to give points in the sample space with the most potential to deliver promising results, and the proxy model is then updated with each new recommended point. This type of optimization is not only limited to optimizing functions, but can be utilized in experimental settings as well, such as in materials science or mining. This method of optimization has found success in situations where each function evaluation or trial is expensive, such as tuning hyperparameters in machine learning and locating optimal points for drilling in geostatistics.

1 Introduction and Literature Review

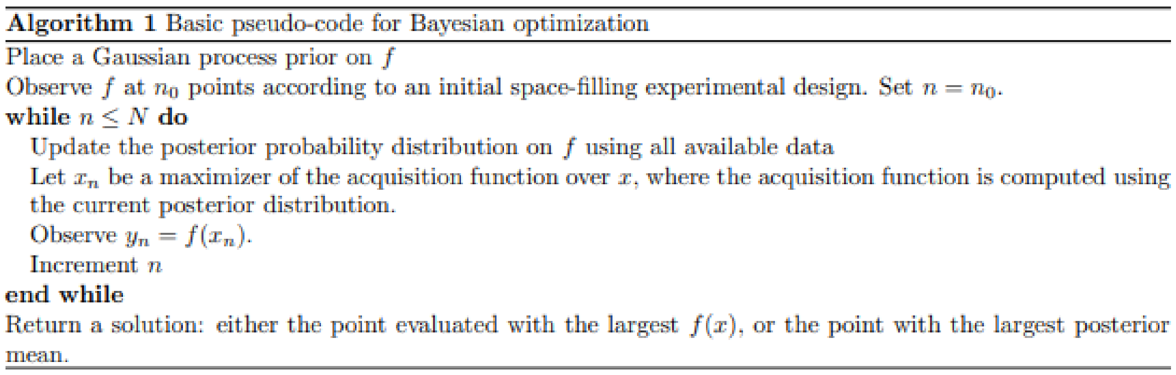
Whether it is training a machine learning model, running a scientific experiment, or adjusting algorithmic hyperparameters, many real-world optimization issues involve costly to evaluate functions where each iteration incurs considerable computational or resource costs [2]. Even if they are effective, traditional gradient-based techniques are frequently impractical in the situations, since they may not exist or be inaccessible for most of people. Bayesian optimization (BO), a systematic paradigm for globally optimizing black-box functions, was developed in response to this challenge. BO has become a key technique for applications ranging from robotics to automatic machine learning (AutoML) by carefully balancing exploration and exploitation using probabilistic surrogate models and acquisition functions according to scholars and researchers [5, 6].

Bayesian Optimization follows an iterative loop of guessing, evaluating, and updating. The objective function is first modeled using a probabilistic surrogate model such as a Gaussian process, which encodes uncertainty and past beliefs. The best option is then chosen using an acquisition function like predicted improvement, which helps gain useful information while avoiding unnecessary trials [2]. Lastly, fresh data is added to the surrogate model, improving forecasts for the future. Uninformed techniques that exhaustively or randomly sample the parameter space, such as grid or random search, often underperform compared to BO. For example, random search is not adaptive and frequently wastes evaluations on suboptimal locations, whereas grid search scales poorly with complexity [1]. However, even in higher dimensional environments, BO's data-driven sequential decision making allows for relatively quick convergence to almost optimal solutions [5].

In machine learning hyperparameter tuning, where evaluating a single configuration can take hours or even days, BO showed reliability in practice. Models like deep neural networks and gradient-boosted trees are highly sensitive to hyperparameters, while their tuning landscapes are often non-convex, noisy, and expensive to navigate. BO addresses these challenges by exploring the search space, prioritizing configurations that balance uncertainty with expected performance. Its consistent outperformance of manual and heuristic methods has made it a foundation in AutoML frameworks such as Google's Vizier and Auto-WEKA [3].

Methodological developments in surrogate modeling and acquisition functions are essential to BO's success. In contrast to conventional squared-exponential kernels, Snoek et al. argue that Gaussian processes (GPs) with Matern kernels offer a resilient prior for representing non-smooth, real-world aims [6]. Furthermore, integrated acquisition functions that marginalize hyperparameter uncertainty, such as Monte Carlo expected improvement, improve robustness against model misspecification [2]. Meanwhile, Shahriari et al. systematized the exploration-exploitation trade-off through a variety of acquisition strategies, including Thompson sampling and entropy search, which adaptively refine the search based on information-theoretic criteria [5].

In our paper, we aim to explore the foundations, mechanisms, and practical applications of Bayesian optimization. We started off by introducing the concept of BO and emphasizing its advantages over conventional methods. Next, we analyze how surrogate models and acquisition functions contribute to the process of running BO. Finally, we apply BO to tune hyperparameters in neural networks using the MNIST dataset, demonstrating its effectiveness in improving model accuracy. The relevant code and data visualization are provided in the Appendix for reference.



```
Algorithm 1 Basic pseudo-code for Bayesian optimization
Place a Gaussian process prior on $f$
Observe $f$ at $n_0$ points according to an initial space-filling experimental design. Set $n = n_0$.
while $n \leq N$ do
  Update the posterior probability distribution on $f$ using all available data
  Let $x_n$ be a maximizer of the acquisition function over $x$, where the acquisition function is computed using
  the current posterior distribution.
  Observe $y_n = f(x_n)$.
  Increment $n$
end while
Return a solution: either the point evaluated with the largest $f(x)$, or the point with the largest posterior
mean.
```

Figure 1: Neat illustration of how the Bayesian Optimization iteration process works. This figure was obtained from [2].

2 Theoretical Approach and Mathematical Foundations

2.1 Gaussian Process Regression

As was previously mentioned, Bayesian Optimization can be very useful in situations where the function we want to optimize is unknown, expensive/difficult to evaluate, and/or we do not have knowledge of its gradient. As in most optimization problems, the problem can be stated rather simply as

$$\hat{x} = \arg\max_{x \in A} F(x)$$

Since the objective function $F(x)$ is either unknown or expensive to evaluate, it is approximated with a surrogate function, which in most cases is a Gaussian Process. The Gaussian Process provides a Bayesian posterior probability distribution which describes the potential values for $F(x)$ at a candidate point x [2]. Each time we evaluate our surrogate function f at a new point this posterior distribution is updated.

A Gaussian Process is a stochastic process where any point x in some domain of interest A is assigned a random variable $f(x)$ and the joint distribution of a finite number of these $f(x)$'s is itself normally distributed

$$\rho(f(x_1), \ldots, f(x_n)) = \mathcal{N}(\mu_0(x_1, \ldots, x_k), \Sigma_0(x_{1:k}, x_{1:k}))$$

We can then write a Gaussian Process in its general form and as it's often found in literature as

$$f(x) \sim \mathcal{GP}(m(x), K(x_i, x_j)) \quad [4]$$

The Gaussian Process is thus completely specified by its mean $m(x)$ and covariance matrix/kernel $K(x_i, x_j)$. The mean $m(x)$ is a vector which is constructed by evaluating a mean function μ_0 at each x_i. The covariance matrix is a positive semi-definite matrix which is constructed by evaluating a kernel function Σ_0 at each pair of points x_i and x_j. Also, the kernel is chosen in such a way so that the points x_i and x_j which are closer in the input space have a larger positive correlation and this encodes the belief that points which are closer to each other in the input space should have more similar function values than points which are located further apart [2].

Suppose we have already observed some values $f(x_1), f(x_2), \ldots, f(x_n) = f(x_{1:n})$ and we wish to infer the value of f at some new point x. Then we can compute the conditional distribution of $f(x)$ given these observations which will also follow a normal distribution

$$f(x)|f(x_{1:n}) \sim \mathcal{N}(\mu_n(x), \sigma_n^2(x))$$

This conditional distribution is the posterior probability distribution with posterior mean $\mu_n(x)$, which is a weighted average between the prior $\mu_0(x)$ and an estimate based on the data $f(x_{1:n})$ with a weight which depends on the type of kernel used [2]. The posterior function is then updated with each new observation that is made. The posterior mean can be written as

$$\mu_n(x) = \Sigma_0(x, x_{1:n})\Sigma_0(x_{1:n}, x_{1:n})^{-1}(f(x_{1:n}) - \mu_0(x_{1:n})) + \mu_0(x)$$

The posterior variance $\sigma_n^2(x)$ is equal to the prior covariance $\Sigma_0(x, x)$ minus a term which corresponds to the variance removed by observing $f(x_{1:n})$ [2]. This can be written as

$$\sigma_n^2(x) = \Sigma_0(x, x) - \Sigma_0(x, x_{1:n})\Sigma_0(x_{1:n}, x_{1:n})^{-1}\Sigma_0(x_{1:n}, x)$$

Although the user has the choice of which kernel they want to specify, it must be a positive semi-definite function and it should have the property that points closer in the input space are more strongly correlated [2]. In mathematical terms this can be represented as if $||x - x'|| < ||x - x''||$ for some norm $|| \cdot ||$ such as the L^2 norm, then $\Sigma_0(x, x') > \Sigma_0(x, x'')$. Two of the most commonly used simple kernels are the power exponential (Gaussian) and Matern kernel.

The power exponential kernel can be written as

$$\Sigma_0(x, x') = \alpha_0 e^{-||x-x'||^2}$$

where $||x - x'||^2 = \sum_i^k \alpha_i(x_i - x_i')^2$ and $\alpha_{0:k}$ are adjustable parameters [2]. Varying this parameter α indicates different beliefs about how quickly the function $f(x)$ changes with x. The other commonly used kernel is the Matern kernel, which can be written as

$$\Sigma_0(x, x') = \alpha_0 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu}||x - x'||^2\right)^\nu K_\nu \left(\sqrt{2\nu}||x - x'||\right)$$

where K_ν is the modified Bessel function and ν is another parameter in addition to $\alpha_{1:k}$ [2]. A clear disadvantage to utilizing this kernel instead of the power exponential kernel is that now we would have one more parameter to tune.

In addition to choosing the kernel function we are also able to choose the mean function. The most common choice for the mean function is a constant value $\mu_0(x) = c$ [2]. If the function f is believed to have a trend or an application-specific parametric structure then the mean function can be specified as

$$\mu_0(x) = \mu + \sum_i^k \beta_i \psi_i(x)$$

where ψ_i is a parametric function with parameters β_i and is often a low order polynomial in x [2].

In the Bayesian nomenclature the mean and the kernel functions are the priors, and the parameters of these functions are the hyperparameters. The hyperparameters can be represented as a vector such as $\gamma = \langle \alpha_{0:k}, \nu, \mu \rangle$. To choose these hyperparameters there are several approaches such as finding the maximum likelihood estimate. In the maximum likelihood approach, using the observations we have already made $f(x_{1:n})$, we calculate the likelihood of these observations under the prior i.e. $P(f(x_{1:n})|\gamma)$ which is a multivariate normal density [2]. Then in maximum likelihood estimation we select the γ which satisfies

$$\hat{\gamma} = \arg\max_\gamma P(f(x_{1:n})|\gamma)$$

Another way to choose the hyperparameters is by supposing that the hyperparameters γ were themselves chosen from a prior $M(\gamma)$ [2]. Then we estimate γ by the maximum a posteriori (MAP) estimate, which is the γ that maximizes the posterior, i.e.

$$\gamma = \arg\max_\gamma P(\gamma|f(x_{1:n})) \propto \arg\max_\gamma P(f(x_{1:n})|\gamma) \cdot P(\gamma)$$

We obtain this by using Bayes' rule and by dropping the normalizing constant since it doesn't depend on the hyperparameter γ.

A third approach which can be used to choose the hyperparameters γ is the fully Bayesian approach. In this approach we compute the posterior distribution on $f(x)$ marginalizing over all possible values of the hyperparameters, i.e

$$P(f(x) = y|f(x_{1:n})) = \int P(f(x) = y|f(x_{1:n}), \gamma) P(\gamma|f(x_{1:n})) d\gamma$$

Usually, this integral cannot be solved analytically but it can be approximated through sampling

$$P(f(x) = y|f(x_{1:n})) \approx \frac{1}{J} \sum_{j=1}^J P(f(x) = y|f(x_{1:n}), \gamma = \hat{\gamma}_j)$$

where $\hat{\gamma}_j : j = 1, \ldots, J$ are sampled from $P(\gamma|f(x_{1:n}))$ by an MCMC method [2].

2.2 Acquisition Functions

Now that we have modeled our objective function with a Gaussian Process we utilize acquisition functions to give us guidance as to where to sample at the next iteration and search for the optimum point which maximizes (or minimizes) our function. Acquisition functions take the mean and variance at each point and compute a value which indicates how desirable it is to sample at this point [4]. We then sample at the point which has the highest output from the chosen acquisition function which should correspond to a high value in the objective function. Acquisition functions will have the highest values where the mean and/or variance is the largest. The main ideas associated with acquisition functions are: they are heuristics for evaluating the utility at a point, they are a function of the surrogate posterior, they combine exploration and exploitation, and they are inexpensive to evaluate [4]. There are numerous acquisition functions with the most popular ones in practice and literature appearing to be the expected improvement, probability of improvement, and upper confidence bound acquisition functions. For the sake of time, I will only elaborate on the expected improvement acquisition function.

Suppose we are trying to maximize the objective function $F(x)$ and we have already made n observations of the surrogate model $f(x)$. Our current best value then is $f_n^* = \max_{m \leq n} f(x_m)$. If we can make an additional observation $f(x)$ anywhere on our domain A then our current best value will either be $f(x)$ if $f(x) > f_n^*$ or it will be f_n^* if $f(x) < f_n^*$. The improvement in the value of the best observed point is then $\max(f(x) - f_n^*, 0)$. Since $f(x)$ is unknown until after it has been evaluated and we would like to maximize the absolute difference of $f(x) - f_n^*$ we can choose the x which maximizes the expected value of $\max(f(x) - f_n^*, 0)$ [2]. Thus the expected improvement can be written as

$$\text{EI}_n(x) = E_n[\max(f(x) - f_n^*, 0)]$$

The expected value here indicates the expectation taken under the posterior distribution given evaluations of f at $x_1, x_2, \ldots, x_n$. The expected improvement can be evaluated in closed form using integration by parts and can be expressed as

$$\text{EI}_n(x) = \max(\delta_n(x), 0) + \sigma_n(x)\phi\left(\frac{\delta_n(x)}{\sigma_n(x)}\right) - \max(\delta_n(x), 0)\Phi\left(\frac{\delta_n(x)}{\sigma_n(x)}\right)$$

where $\delta_n(x) = \mu_n(x) - f_n^*$ is the expected difference in value between the proposed point x and currently observed best [2]. Recall that μ_n is the mean of the posterior $f(x)|f(x_{1:n}) \sim \mathcal{N}(\mu_n(x), \sigma_n^2(x))$. The point which is evaluated at the next iteration then is the one which satisfies

$$x_{n+1} = \arg\max_x \text{EI}_n(x)$$

Unlike the objective function $F(x)$, the $\text{EI}_n(x)$ is inexpensive to evaluate and has rather easy to compute first and second order derivatives which allow simpler optimization techniques to be used to solve $\arg\max_x \text{EI}_n(x)$ [2].

Evaluating at points with high posterior expected value (where the posterior mean is large) relative to the previous best point is valuable as the global optimum is likely to be located at such points. Sampling at points where the posterior expected value is high is known as exploitation. On the other hand, evaluating points with high uncertainty is valuable as it allows us to gain insight into the objective function at points where we have little knowledge about it. Sampling at points with high posterior variance is known as exploration. Expected improvement, along with other acquisition functions, will recommend sampling at points with the highest posterior variance and expected value and this balance is known as the exploration vs exploitation trade-off.

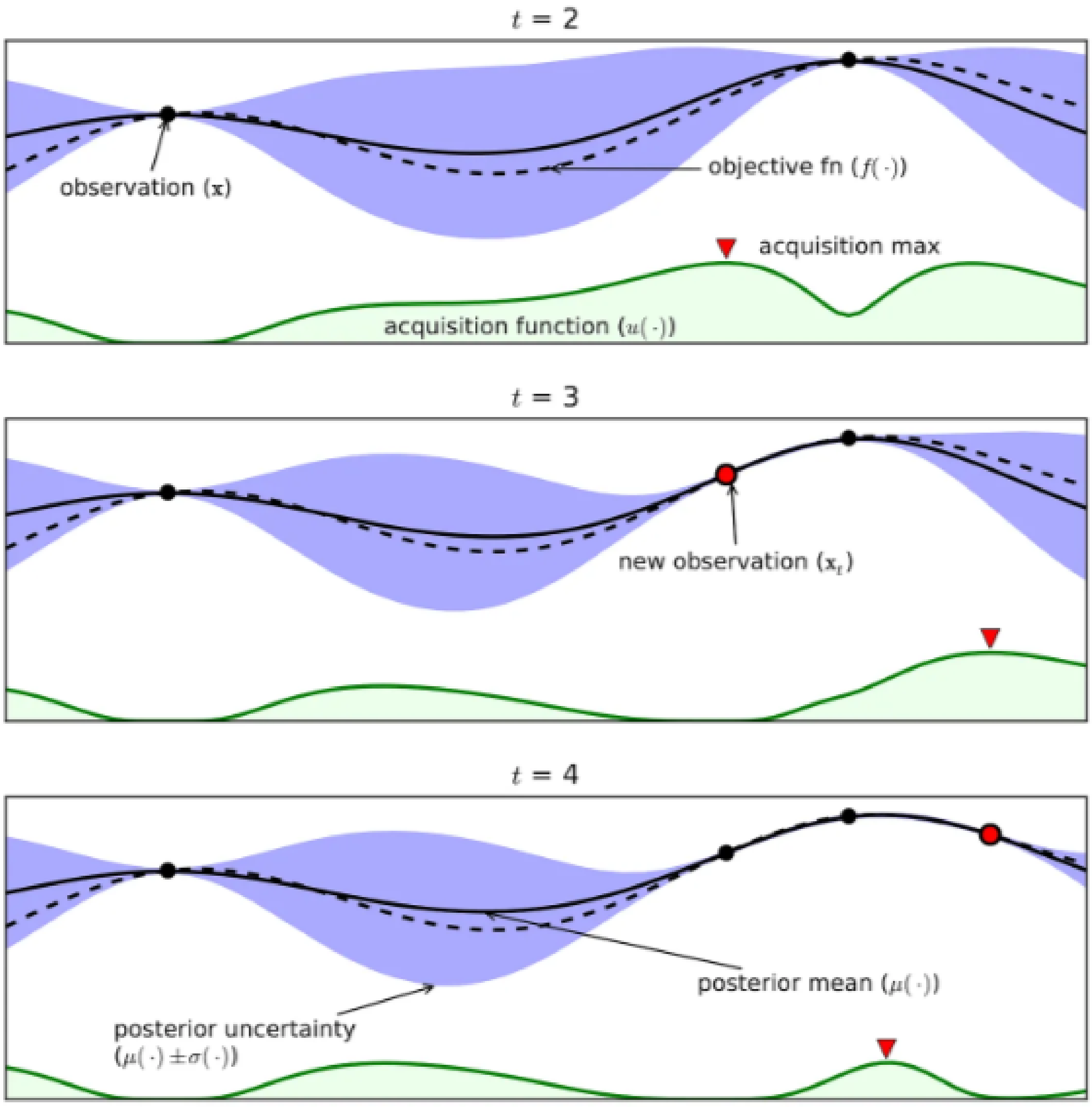


Figure 2: Illustration of Bayesian Optimization over three iterations. This is a helpful visual which shows how the acquisition function changes with each iteration and how it prefers points with high posterior mean and variance. Note that the acquisition function is zero or very close to zero at the points where the function has been evaluated because there is no uncertainty at these points. (This figure was obtained from Medium.com)

3 Bayesian Optimization on Neural Networks

Neural Networks are machine learning models resembling the architecture of the human brain. The network consists of at least three layers: input layer, hidden layer, and output layer. Each layer contains nodes which process input signals with an activation function and pass the output to the subsequent layer. The connections between nodes are associated weights, which determine the influence in outputs.

Neural Networks require hyperparameters such as learning rate, number of hidden layers, batch size, and weight. Training neural networks is usually computationally intensive and time consuming, and difficult to interpret due to the black box like nature, so it is crucial to adopt efficient strategies for hyperparameter optimization. Bayesian Optimization offers a systematic approach to tune hyperparameters for neural networks.

3.1 Implementation

We have implemented a simple demonstration for a feedforward neural network. The neural network has been trained on the MNIST dataset which consists of handwritten images of digits from 0-9. The training set has 20,000 observations and the test set has 10,000 observations.

For this purpose, we used the Scikit-Learn `MLPClassifier` and `BayesianOptimization` package. Accuracy has been used as the primary metric for optimization.

Phase 1: Initially, we begin with optimizing learning rate and alpha value.

- Learning rate: Controls the step size during weight updates.
- Alpha: Ridge penalty coefficient to prevent overfitting.

Table 1: Phase 1: Optimized Parameters

Parameter	Best Values
Alpha	0.008
Learning Rate	0.077
Accuracy	0.946

Phase 2: Next, we included the hidden layer size as an additional parameter for tuning.

Table 2: Phase 2: Optimized Parameters

Parameter	Best Values
Alpha	0.075
Hidden Layer Size	116
Learning Rate	0.005
Accuracy	0.957

From both of these experiments, we can notice tuning of parameters highly influences the accuracy of the model (See Appendix for Plots). Choosing an appropriate combination of parameters is important for modeling. By also including the size of hidden layers in phase 2, we were able to increase the accuracy by about 1%. By considering other parameters that could be included for optimization, we could improve our model with the best possible set of parameter values. However, we need to keep in mind that Bayesian Optimization might not work well with high dimensionality. It could still be computationally expensive to use BO for tuning hyperparameters in deep neural networks.

4 Conclusion

In essence, Bayesian Optimization is a powerful iterative technique which is utilized in situations where the objective function or experiment is difficult, expensive, or impossible to evaluate. We have explored its theoretical foundations such as the acquisition functions and Gaussian Processes which are used to construct it, and we have demonstrated how it can be used to tune hyperparameters in machine learning models such as neural networks. Bayesian optimization does have its disadvantages with one obvious one being that it adds additional hyperparameters, specifically from the kernel functions, which must be specified and that may be hard to do so without knowledge/expertise of the given problem. Bayesian Optimization has many other interesting applications such as Kriging in spatial analysis and maximizing reward/utility in multi-armed bandit problems which would be fascinating topics to talk about in future research.

References

[1] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

[2] P. I. Frazier. A tutorial on bayesian optimization. *arXiv*, 2018. https://arxiv.org/abs/1807.02811.

[3] J. Grosse, C. Zhang, and P. Hennig. Optimistic optimization of gaussian process samples. *arXiv*, 2022. https://doi.org/10.48550/arXiv.2209.00895.

[4] A. Nandy, C. Kumar, D. Mewada, and S. Sharma. Bayesian optimization—multi-armed bandit problem. *arXiv*, 2020. https://doi.org/10.48550/arXiv.2012.07885.

[5] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104:148–175, 2016.

[6] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. 25:2951–2959, 2012.

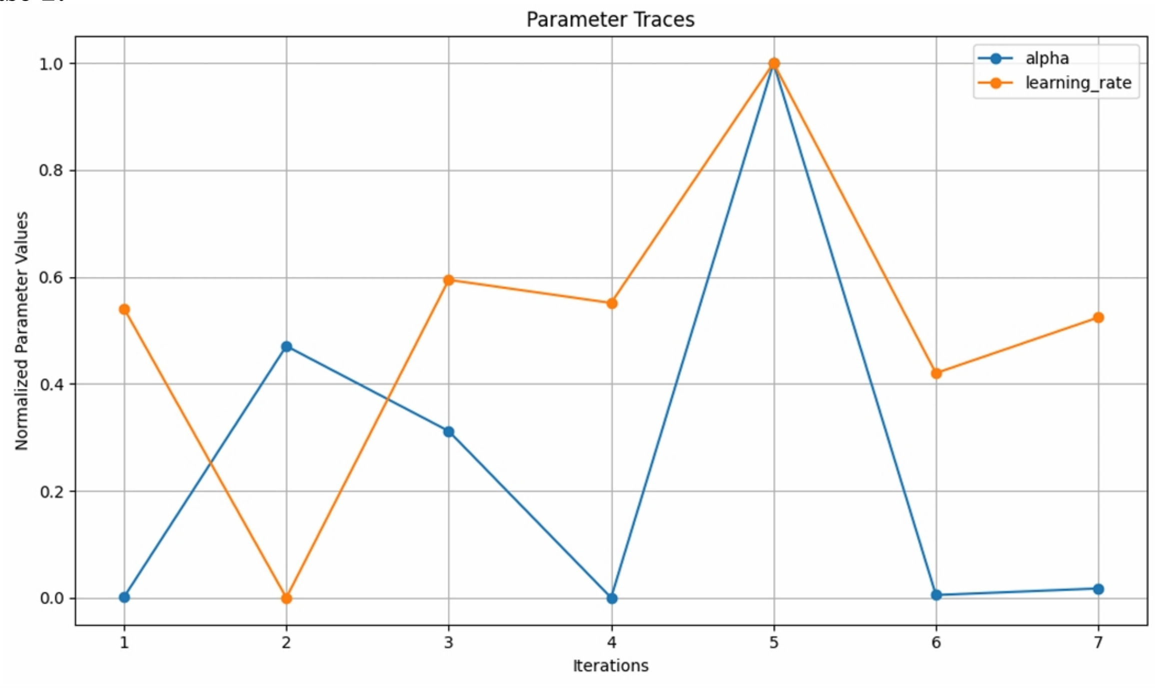
AI Usage and Team Member Contribution

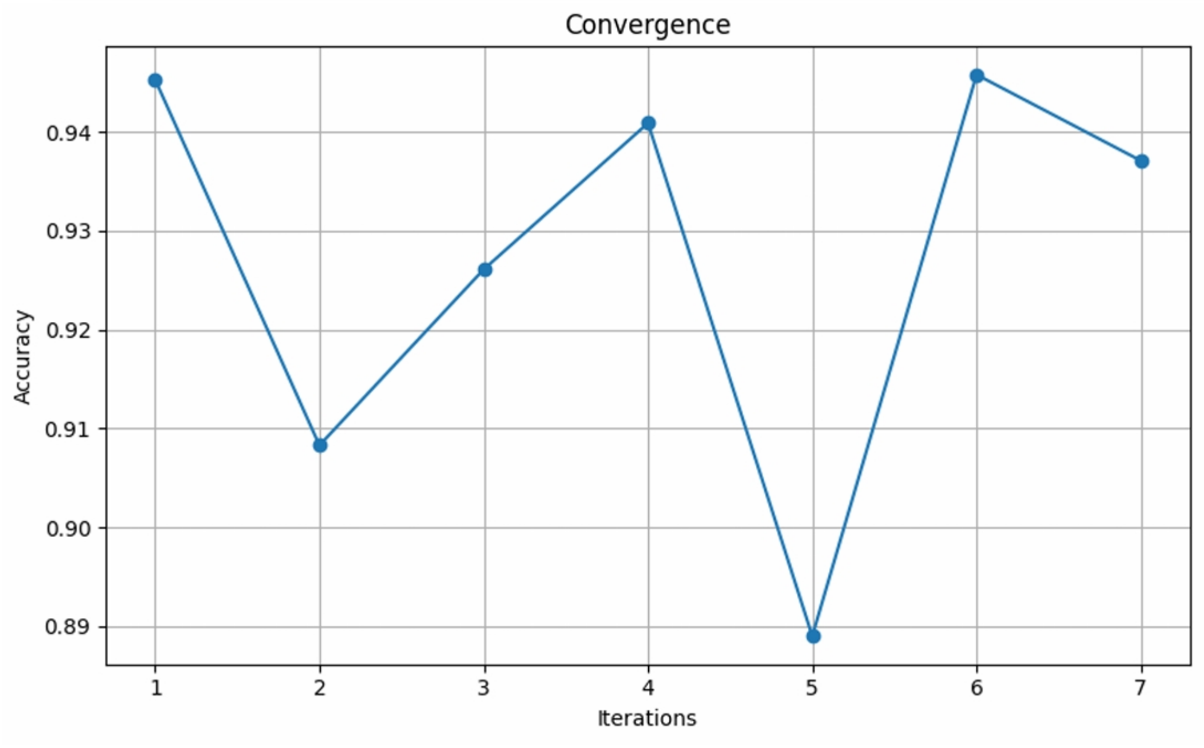
AI was utilized for formatting our document into LaTeX and for helping us write and refine the Python code in section 3. Also, AI was used for doing some basic research and getting some facts about Bayesian Optimization.

Yuqian completed section 1, introducing the topic and performing some literature review. Hubert Witkos completed section 2 on the theory and math behind Bayesian Optimization. Lastly, Soujan, with some help from Yongliang, completed section 3 on hyperparameter tuning in Python with the bayes opt library.

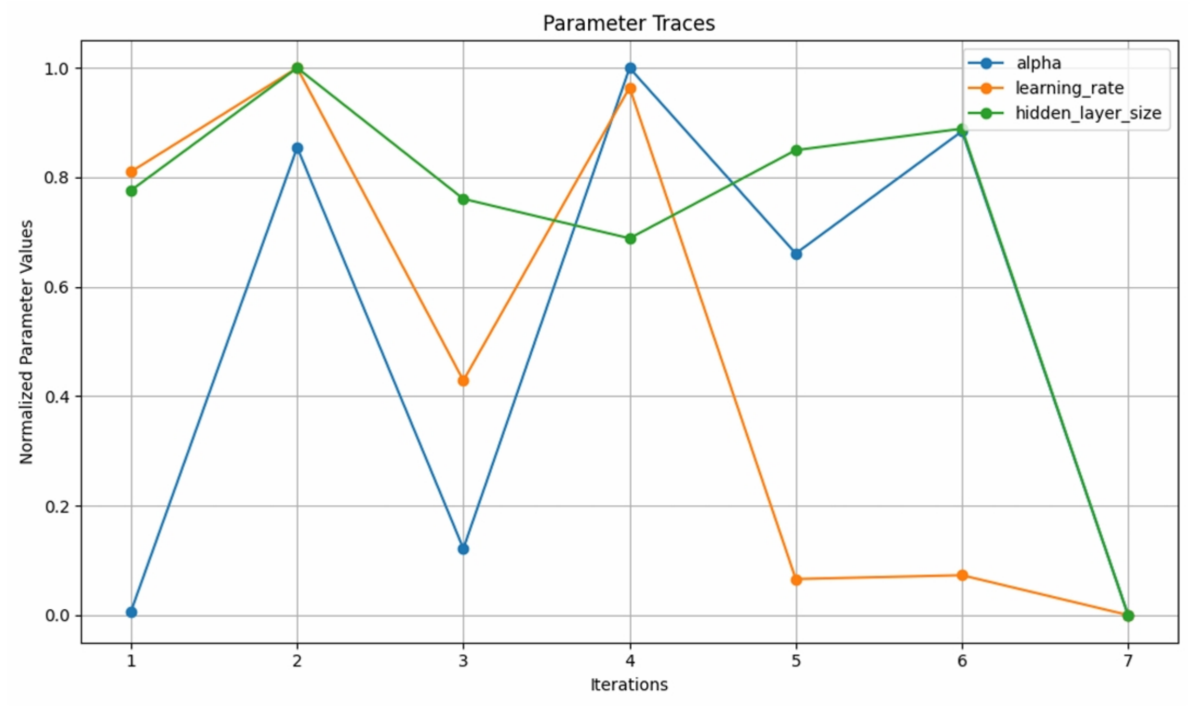
Appendix

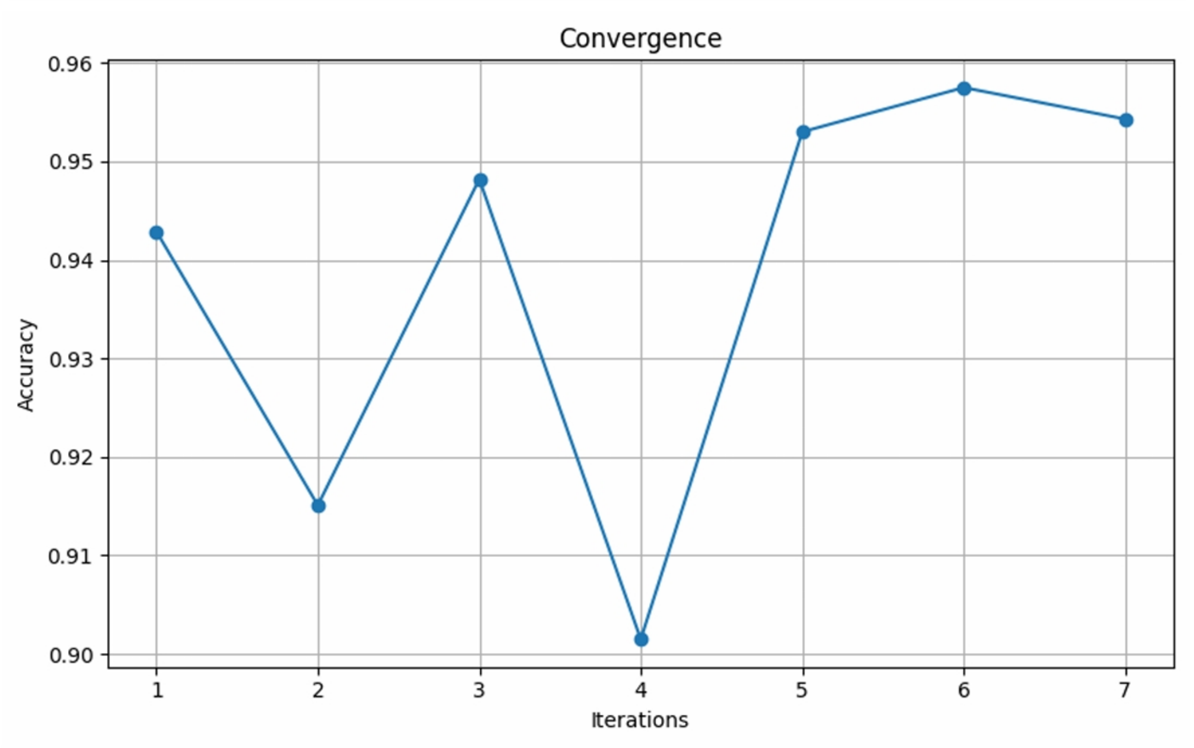
Phase 1:





Phase 2:


Parameter Traces
alpha
learning_rate
hidden_layer_size
Normalized Parameter Values
Iterations
1.0
0.8
0.6
0.4
0.2
0.0
1
2
3
4
5
6
7


Convergence
Accuracy
Iterations
0.96
0.95
0.94
0.93
0.92
0.91
0.90
1
2
3
4
5
6
7

Python Script

```
!pip install bayesian-optimization
```

Phase 1:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from bayes_opt import BayesianOptimization

#Loading MNIST Dataset
mnist_test = pd.read_csv('sample_data/mnist_test.csv')
Y_test = mnist_test.iloc[:, 0].values
X_test = mnist_test.iloc[:, 1:].values

mnist_train = pd.read_csv('sample_data/mnist_train_small.csv')
Y_train = mnist_train.iloc[:, 0].values
X_train = mnist_train.iloc[:, 1:].values

#Normalizing pixel values
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

#Objective Function for BO
def obj_func(alpha, learning_rate):
    model = MLPClassifier(
        hidden_layer_sizes=(64,),  #Fixed Hidden Layer
        activation="relu",
        alpha=alpha,
        learning_rate_init=learning_rate,
        max_iter=30,
        random_state=7,
    )
    model.fit(X_train, Y_train)
    return model.score(X_test, Y_test)

#Parameter Bounds
bounds = {
    "alpha": (1e-5, 0.1),
    "learning_rate": (1e-4, 0.1),
}

#Running Optimizer
optimizer = BayesianOptimization(f=obj_func, pbounds=bounds, random_state=7)
optimizer.maximize(init_points=2, n_iter=5)

#Extracting parameter trace
params_df = pd.DataFrame([res["params"] for res in optimizer.res])
params_df["Iteration"] = range(1, len(params_df) + 1)

#Normalizing parameter values
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
normalized_params = scaler.fit_transform(params_df[["alpha", "learning_rate"]])
```

```
normalized_df = pd.DataFrame(normalized_params, columns=["alpha", "learning_rate"])
normalized_df["Iteration"] = params_df["Iteration"]

#Plotting parameter traces
plt.figure(figsize=(10, 6))
for param in ["alpha", "learning_rate"]:
    plt.plot(normalized_df["Iteration"], normalized_df[param], marker='o', label=param)

plt.xlabel("Iterations")
plt.ylabel("Normalized Parameter Values")
plt.title("Parameter Traces")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

#Plotting convergence
iterations = list(range(1, len(optimizer.space.target) + 1))
targets = optimizer.space.target

plt.figure(figsize=(8, 5))
plt.plot(iterations, targets, marker='o')
plt.xlabel("Iterations")
plt.ylabel("Accuracy")
plt.title("Convergence")
plt.grid(True)
plt.tight_layout()
plt.show()

#Print results
print("Best Parameters: ", optimizer.max["params"])
print("Best Accuracy: ", optimizer.max["target"])
```

Phase 2:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from bayes_opt import BayesianOptimization

#Loading MNIST Dataset
mnist_test = pd.read_csv('sample_data/mnist_test.csv')
Y_test = mnist_test.iloc[:,0].values
X_test = mnist_test.iloc[:,1:].values

mnist_train = pd.read_csv('sample_data/mnist_train_small.csv')
Y_train = mnist_train.iloc[:,0].values
X_train = mnist_train.iloc[:,1:].values

#Normalizing pixel values
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

#Objective Function for BO
def obj_func(alpha, learning_rate, hidden_layer_size):
    model = MLPClassifier(
        hidden_layer_sizes=(int(hidden_layer_size),)
        hidden_layer_sizes=(int(hidden_layer_size),),
        activation="relu",
        alpha=alpha,
        learning_rate_init=learning_rate,
        max_iter=30,
        random_state=7,
    )
    model.fit(X_train, Y_train)
    return model.score(X_test, Y_test)

#Parameter Bounds
bounds = {
    "alpha": (1e-5, 0.1),
    "learning_rate": (1e-4, 0.1),
    "hidden_layer_size": (32, 128),
}

#Running Optimizer
optimizer = BayesianOptimization(f=obj_func, pbounds=bounds, random_state=7)
optimizer.maximize(init_points=2, n_iter=5)

#Extracting parameter trace
params_df = pd.DataFrame([res["params"] for res in optimizer.res])
params_df["Iteration"] = range(1, len(params_df) + 1)

#Normalizing parameter values

scaler = MinMaxScaler()
normalized_params = scaler.fit_transform(params_df[["alpha", "learning_rate", "hidden_layer_size"]])
normalized_df = pd.DataFrame(normalized_params, columns=["alpha", "learning_rate", "hidden_layer_size"])
normalized_df["Iteration"] = params_df["Iteration"]

#Plotting all traces in one plot
plt.figure(figsize=(10, 6))
for param in ["alpha", "learning_rate", "hidden_layer_size"]:
    plt.plot(normalized_df["Iteration"], normalized_df[param], marker='o', label=param)

plt.xlabel("Iterations")
plt.ylabel("Normalized Parameter Values")
plt.title("Parameter Traces")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

#Plotting Convergence
iterations = list(range(1, len(optimizer.space.target) + 1))
targets = optimizer.space.target

plt.figure(figsize=(8, 5))
plt.plot(iterations, targets, marker='o')
plt.xlabel("Iterations")
plt.ylabel("Accuracy")
plt.title("Convergence")
plt.grid(True)
plt.tight_layout()
plt.show()

#Print Results
print("Best Parameters: ", optimizer.max["params"])
print("Best Accuracy: ", optimizer.max["target"])
```