Efficient Hyperparameter Tuning for Resource-Constrained Environments

A Principled Approach Integrating Advanced Initial Design and Surrogate Modeling

Ethan Vuong, Kelsey Lin, Jun Ryu, Catherine Tong (Group 3)

1 Abstract

This study addresses the challenges posed by resource constraints, particularly on edge devices with limited computational power. Hyperparameter tuning is crucial for optimizing deep neural networks but is often computationally expensive. We propose an efficient framework using Latin Hypercube Sampling (LHS) and surrogate models like Gaussian Processes to reduce computational burden. This approach significantly reduces time while improving or maintaining model accuracy compared to traditional grid search methods.

2 Introduction

2.1 Importance and Traditional Methods

Hyperparameter tuning is essential for optimizing the performance of deep neural networks. Traditional methods like grid search and random search are widely used. Unfortunately, they are often inefficient and/or computationally expensive. For example, grid search exhaustively evaluates each combination of hyperparameters. And random search samples them randomly, both requiring significant computational resources.

2.2 Challenges with Resource Constraints

There are many challenges that come with training deep neural networks like the fact that it is resource-intensive. This causes challenges on edge devices with limited computational memory and energy. Efficient hyperparameter tuning under these constraints requires advanced techniques to minimize computational burden while achieving optimal performance.

2.3 Advanced Initial Design Technique

The proposed framework begins with an initial design using techniques like Latin Hypercube Sampling (LHS) which is able to ensure comprehensive coverage of the hyperparameter space. It also provides a strong starting point for optimization. There are also surrogate models such as Gaussian Processes or Bayesian Neural Networks that are used to predict the performance of new hyperparameter sets based on initial data which we will go into more. These models guide the search towards lowering the need for exhaustive evaluations and reducing time.

2.4 Dynamic Stopping Criteria

Dynamic stopping criteria are set to stop the search when improvements fall below a threshold. This approach prevents unnecessary computations and focuses resources on beneficial hyperparameter settings. In conclusion, this framework offers a more efficient approach to hyperparameter tuning, that allows us to ideally reduce computational time and resources while maintaining or enhancing model accuracy compared to traditional grid search methods.

3 Related Work

Prior to a model being trained, the best combination of external configurations (hyperparameters) is generally unknown. Proposing advanced methodologies that yield the best set of hyperparameters while maintaining efficiency has been a motivating factor for hyperparameter optimization research, particularly in the realm of machine learning.

Bayesian Optimization (BO) is part of a broader field of hyperparameter optimization methods, widely known for its efficiency in high-dimensional and expensive search spaces. By leveraging acquisition functions such as Expected Improvement (EI) or the Upper Confidence Bound (UCB), BO is able to successfully balance exploration and exploitation within a search space (Snoek et al., 2012; Shahriari et al., 2016). In addition, Gaussian Processes (GP) have been widely employed for non-parametric modeling of the objective function, providing uncertainty estimates that effectively guide the optimization process (Snoek et al., 2012).

Another approach to hyperparameter optimization involving constructing probabilistic models of low-performing and high-performing hyperparameter regions is known as Tree-structured Parzen Estimators (TPE). This methodology allows for efficient sampling of the most promising configurations, but it typically requires a vast amount of function evaluations and poses a challenge in resource-constrained environments (Bergstra et al., 2011; Bergstra et al., 2013).

The integration of advanced initial design techniques and surrogate modeling can be noted as uncharted territory. With a focus on the ability to obtain optimal configurations in a resource-constrained environment and reduce computational time, our research endeavors to propose a framework that can integrate different methods in order to effectively tune hyperparameters in low-resource settings.

4 Methodology

We propose a methodology that is able to reduce computational time for neural networks compared to traditional hyperparameter optimization approaches such as grid search. Our framework consists of 3 key components: an advanced initial design technique, surrogate modeling, and the use of optimization algorithms tailored to resource-constrained environments.

We will integrate these components through a multistage approach in Python that is able to effectively explore the hyperparameter search space and mitigate computational demands. We will have 2 main files: benchmark.py and comparison.py, where benchmark.py implements grid search and comparison.py employs our framework of surrogate modeling using Gaussian Process with Bayesian Optimization.

Our comparison seeks to illustrate significant decreases in hyperparameter evaluations needed with Bayesian Optimization all while maintaining acceptable model performance in order to emphasize practicality in scenarios where computational resources are limited. To further save on computational time, we will use early stopping if improvements are minimal by setting our patience and threshold criteria.

4.1 Initial Design Techniques

Grid search is a common and traditional approach to hyperparameter tuning in neural networks. It exhaustively evaluates all possible combinations of configurations, which means we would directly evaluate the neural network itself for each set of hyperparameters that we are testing. This can be very computationally taxing and impractical when there is limited access to more powerful machines with greater processing power.

For example, a 4-dimensional grid with 5 levels per parameter would result in 5^4 evaluations or 625 evaluations. This entails running the (neural network) model a total of 625 times to find the best configurations.

Instead, one might use an alternative approach such as Latin Hypercube Sampling (LHS), which is employed within our framework. LHS seeks design points that “fill an unbounded design region as uniformly as possible" and is hence referred to as a space-filling design (Cheng & Jones, 2022). LHS samples a fixed number of hyperparameter combinations from the search space that explores diverse points and rids the need to systematically cover the entire space as is required in grid search.

In our hypothetical search space, using LHS would reduce our 625 evaluations to only 50 or 100 combinations, saving time and computational resources while still exploring a broad range of values.

In short, the first step of our framework will initialize hyperparameter configurations through LHS because of its lower computational cost and ability to produce diverse samples. We directly evaluate our neural network on this reduced amount of configuration samples and gather performance data for our surrogate modeling component, which will be able to identify promising hyperparameters without the need to run the neural network itself any further.

4.2 Surrogate Models

To efficiently and effectively approximate the hyperparameter space and guide the optimization process, our framework employs Bayesian Optimization as the overarching concept that consists of surrogate modeling with Gaussian Process (GP) and acquisition function with Expected Improvement (EI).

Gaussian Process is a non-parametric approach that creates a probabilistic model of the performance landscape using the initial performance data gathered from evaluating the hyperparameter combinations sampled using LHS. The Gaussian Process treats the objective function as a distribution of possible outcomes and is able to predict the performance of untested hyperparameter combinations.

Expected Improvement is the acquisition function in Bayesian Optimization that works alongside the Gaussian Process. EI calculates the predicted mean and uncertainty for a candidate configuration, before selecting the configuration with the best expected improvement to evaluate next. The GP will update accordingly, and the process will repeat. Thus, EI helps achieve the balance between exploration and exploitation, as high uncertainty indicates areas of the search space that have not been explored as in-depth.

In summary, GP provides performance estimates and EI leverages such estimates to lead the search for the best hyperparameter possibilities.

4.3 Use of Optimization Algorithms

Our framework also takes into account cases where we can terminate poorly performing hyperparameter configurations efficiently by incorporating early stopping criteria. This allows us to conserve computational resources by avoiding continuing evaluations that are unnecessary or are unlikely to produce significant improvement.

Additionally, we will incorporate multi-fidelity optimization techniques tailored for resource-constrained environments. This approach entails initially evaluating hyperparameter configurations on lower-fidelity models before full evaluations on promising candidates. This means, for example, starting with reduced dataset sizes or fewer training epochs to first filter out suboptimal configurations in order to allocate computational resources for the most promising regions of the hyperparameter space.

5 Experimentation

Specific to our experiment, we will use the MNIST dataset for training our neural network architecture. This dataset contains a standard benchmark collection of 28x28 pixel grayscale images of handwritten digits. There are a total of 70,000 images ranging from 0 to 9, with the training set comprising 60,000 images and the test set comprising 10,000 images.

5.1 Benchmark in Python

We will be exploring and tuning 2 hyperparameters, which are units per layer and batch size, at 5 levels each. This means that the `benchmark.py` script will conduct a grid search for these two hyperparameters by evaluating all 25 possible combinations of these values. For each combination, it trains a simple convolutional neural network on the MNIST dataset for 2 epochs and measures its performance on the test set. The script records the best accuracy and corresponding hyperparameters. After identifying the optimal configuration, it retrains the model with these best hyperparameters to validate the results. This approach is straightforward, relying on exhaustive search without advanced techniques.

5.2 Bayesian Optimization in Python

Our process in `comparison.py` is designed to assess our framework's ability to minimize computational time and the number of hyperparameter evaluations while still achieving high accuracy. We will compared this to grid search, and our evaluation metrics are the following:

1. **Model accuracy:** The classification performance of the optimized neural network models based on the test dataset.
2. **Computational time:** The total time required for hyperparameter tuning, including initial design, surrogate modeling, and optimization.
3. **Number of hyperparameter evaluations**: The number of unique hyperparameter configurations evaluated during the optimization process.

We will utilize popular machine learning libraries such as PyTorch, scikit-learn, and GPyTorch. The steps of experimentation in our Bayesian Optimization framework can be broken into 2 steps as briefly mentioned in Methodology: initial points training and sequential search. We will also set early stopping criteria with patience 5 and threshold at 0.1

<u>Initial Points Training</u>

1. Use Latin Hypercube Sampling (LHS) to generate a set of initial hyperparameter configurations. These points are normalized and within the range [0, 1].
2. Scale Points to Actual Ranges: Convert the normalized points to the actual ranges for the hyperparameters (e.g., units in [16, 80] and batch size in [32, 128]).
3. Train and Evaluate: For each initial configuration, we will:
    - Train the model with specific units and batch size.
    - Evaluate the model on the test set to obtain the accuracy metric.
    - Record the accuracy associated with specific configuration.
4. Identify the Best Initial Configuration: Among all the initial configurations, identify the one achieving the highest accuracy.

<u>Sequential Search</u>

1. Fitting Gaussian Process (GP) Model: Use the initial data points (hyperparameter configurations and respective accuracies) to fit a Gaussian Process model.
2. Generate Candidate Points: Create a grid or use another method to generate candidate points for the next configuration to test.
3. Expected Improvement (EI): Compute the Expected Improvement (EI) for each candidate point using the GP model. EI quantifies how much improvement over the current best accuracy a new configuration might provide.
4. Select the Best Candidate: Choose the candidate point with the highest EI as the next configuration to test.
5. Train and Evaluate: Train a model with this new configuration and evaluate its accuracy.
6. Update Data: Normalize the new configuration, add it to the set of configurations, and update the GP model with the new data point (configuration and its respective accuracy).

7. Update Best Configuration: If the new configuration yields a higher accuracy than the current best, then we will update the best configuration.
8. We repeat the sequential search for a predefined number of iterations or until a stopping criterion is met.

Across `benchmark.py` and `comparison.py`, we will maintain the same neural network model architecture, conduct 5 trials per file with 2 epochs per trial, hold the hyperparameter search space constant, and run these trials on the same device/hardware to measure computational time.

Collected Data:

Trial Number	Units	Batch Size	Total Computational Time (Seconds)	Total Hyperparameter Evaluations	Test Accuracy
1	64	48	3395.56	25	98.80%
2	64	48	3395.78	25	98.88%
3	32	48	3417.86	25	98.64%
4	64	32	3247.14	25	99.11%
5	80	32	3254.22	25	99.05%

Summary Statistics:

mean(Total Computational Time (Seconds)): 3342.112

mean(Total Hyperparameter Evaluations): 25

mean(Test Accuracy): 98.896%

Collected Data:

Trial Number	Units	Batch Size	Total Computational Time (Seconds)	Total Hyperparameter Evaluations	Test Accuracy

1	65	42	2308.95	16	98.57%
2	41	51	1768.85	15	98.76%
3	80	96	2388.41	15	98.82%
4	61	97	2236.83	15	98.64%
5	66	90	2071.88	15	98.82%

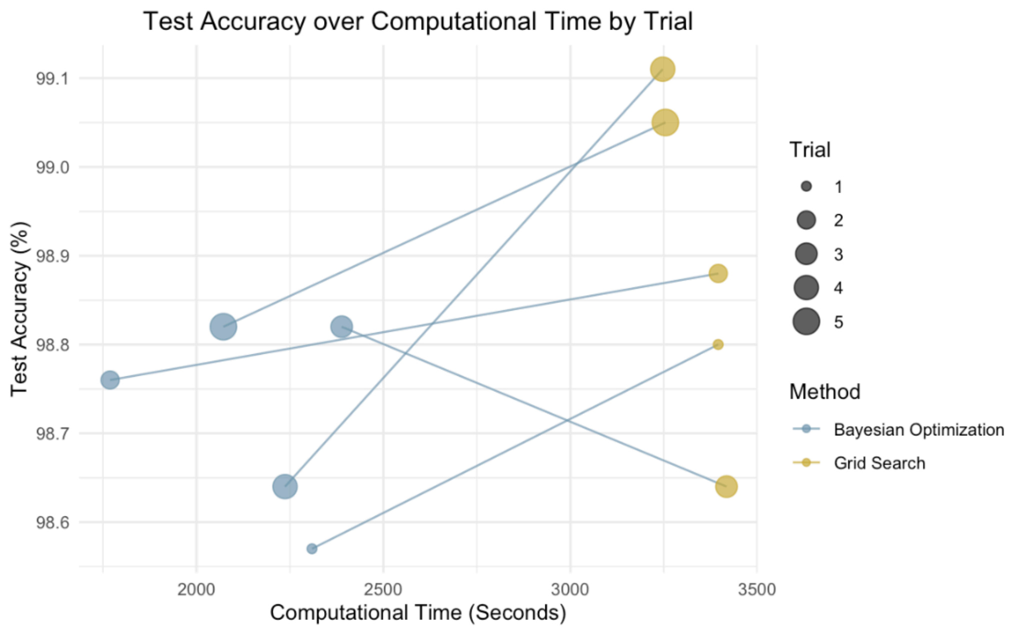
Summary Statistics:

mean(Total Computational Time (Seconds)): 2154.984

mean(Total Hyperparameter Evaluations): 15.2

mean(Test Accuracy): 98.722%

6 Results



Overview

The scatter plot compares the performance of Bayesian Optimization and Grid Search across five trials, focusing on test accuracy versus computational time. Each point represents a trial, with the size of the point indicating the trial number.

Grid Search

For the grid search method, we explored 25 combinations of two hyperparameters: units per layer and batch size. Each trial involved training the neural network for two epochs. The results from five trials are summarized below:

- **Computational Time:** Grid Search trials took significantly longer, with computational times ranging from approximately 3200 to 3500 seconds.
- **Test Accuracy:** Test accuracy for Grid Search ranged from about 98.6% to 99.1%, with some trials achieving slightly higher accuracy compared to Bayesian Optimization.
- **Observations:** While Grid Search occasionally achieved marginally higher accuracy, it required substantially more computational time, highlighting its inefficiency in resource-constrained environments.

Bayesian Optimization

In the Bayesian Optimization framework, we utilized Latin Hypercube Sampling (LHS) for initial design, followed by Gaussian Process modeling and Expected Improvement (EI) for sequential search. Five trials were conducted under the same conditions as the grid search. The results are as follows:

- **Computational Time:** Generally, Bayesian Optimization required less computational time compared to Grid Search, with most trials taking less than 2500 seconds.
- **Test Accuracy:** The test accuracy for Bayesian Optimization hovered around 98.6% to 98.9%, showing consistent performance across trials.
- **Observations:** Despite some variability, Bayesian Optimization maintained high accuracy with significantly less computational time, demonstrating its efficiency.

Comparison

Bayesian Optimization significantly reduced the computational time by approximately 35% compared to grid search, with a mean total computational time of 2154.984 seconds versus 3342.112 seconds for grid search. Additionally, Bayesian Optimization required 10 fewer hyperparameter evaluations on average (15.2 compared to 25). Despite these reductions, the mean test accuracy remained comparable, with Bayesian Optimization achieving 98.722% versus 98.896% for grid search.

7 Limitations

There are four major limitations to the procedures we conducted.

First, using a simple dataset like MNIST, with only 10 classes to identify, allows neural networks to achieve very high accuracies when fine-tuning the hyperparameters. Across the 10 separate trials (5 for the grid search and 5 for our refined method), the accuracies never fell below 98%, making it difficult to compare the accuracies between the two approaches and determine which method is superior in terms of model accuracy.

The next limitation is that we experimented with a small hyperparameter search space. Specifically, we ran both methods in a two-dimensional space, testing only two hyperparameters: batch size and units per layer. As a result, the difference in average computational time between the two approaches was about 20 minutes, which, in practical terms, makes no significant difference. This indicates that, in a real working environment, spending an additional 20 minutes to run a grid search is feasible and won't result in significant penalties or losses.

Moreover, the other hyperparameters that were not involved in our tuning process such as the number of layers and activation functions were set too "well", causing the model accuracies to consistently converge to a high value. This relates to the first limitation, where the resulting high accuracies are harder to compare between the two methods.

Given the timeline of our research process and the constraints of our computational resources, we were limited to running only five trials of each method, with just two epochs per trial. Although our results were consistent and free of significant outliers, increasing the number of trials and epochs could have enhanced the reliability of our findings.

8 Future Improvements

Based on the limitations I discussed, here are some further improvements that can be made:

Due to the exceptionally high accuracies achieved by both methods, one change we can implement is varying the neural network architecture. Specifically, this involves adjusting the other hyperparameters (those not part of the tuning process) so that the model purposely performs less optimally. The purpose of this approach is to increase the likelihood of revealing a more significant difference in accuracies between the two methods, as the results might not always converge to such high values like 98% or 99%.

Furthermore, we can increase the hyperparameter space we are working with. Expanding the hyperparameter space has the potential to show a more significant difference in computational performance between the two models, highlighting the true advantage of using the refined model in a resource-constrained setting.

We can also use a more complex dataset than MNIST, such as Fashion MNIST or CIFAR-100. CIFAR-100, in particular, contains images of 100 different classes, including everyday objects like apples and clocks, compared to MNIST's 10 classes. Additionally, the smaller number of training and testing images per class in CIFAR-100 makes it more challenging for the model to learn, providing a more rigorous test of the methods. This modification again helps to address the issue of consistently high accuracy values.

Lastly, we can reduce the initial sample size selected through LHS sampling. By doing so, we allow Bayesian Optimization to run longer. In our current implementation, the surrogate modeling already performs exceptionally well, enabling Bayesian Optimization to optimize after just one or two iterations. Reducing the initial sample size would allow the surrogate modeling to underperform slightly, letting the optimization process have a greater impact on the hyperparameter evaluations.

9 Conclusion

This study demonstrates that Bayesian Optimization can achieve similar accuracy levels as traditional grid search while significantly reducing computational time as well as reducing the number of hyperparameter evaluations. The integration of advanced initial design techniques like Latin Hypercube Sampling (LHS) and surrogate modeling with Gaussian Processes proved successful in navigating the hyperparameter space efficiently. These findings are valuable for environments with limited computational resources and where time is limited.

Bayesian Optimization not only maintains model performance but also enhances the practicality of hyperparameter tuning in resource-constrained settings. By reducing the computational burden, this approach offers a hopeful alternative to traditional methods. We can see it as a great potential to be a powerful tool for optimizing deep neural networks. Future work could explore more complex datasets and larger hyperparameter spaces to further validate the efficacy and scalability of this framework.

Citations

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems* (pp. 2546-2554).

Bergstra, J., Yamins, D., & Cox, D. D. (2013). Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on Machine Learning* (pp. 115-123).

Cheng, C.-S., & Jones, B. (2022). Latin hypercubes and space-filling designs. *arXiv preprint arXiv:2203.06334*. Retrieved from https://arxiv.org/abs/2203.06334

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., & de Freitas, N. (2016). Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1), 148-175.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems* (pp. 2951-2959).