Optimizing CNN Image Synthesis via Active Learning-Driven Hyperparameter Tuning in Classifier-Guided Diffusion Models

ISE610 Course Project

Submitted by

Negin Ashrafi, Minoo Ahmadi

Submitted to

Professor Qiang Huang

Department of Industrial and Systems Engineering

University of Southern California

November 26, 2023

Table of Contents

Abstract 0

1. Introduction 0

2. Literature Review 1

3. Model Definition 2

   3.1. Diffusion Model 2

   3.2. CNN 4

4. Experiment Design 5

   4.1. Fractional Factorial Design 5

   4.2. Response Surface Design 6

   4.3. Active Learning 8

5. Experiment and Computational Results 12

6. Conclusion 19

7. References 21

Appendix 22

Abstract

This project develops an active learning-driven framework for optimizing a CNN classifier to provide conditional guidance to a diffusion model for improved image generation quality. A fractional and full factorial design screens classifier hyperparameters, with response surface analysis to model complex factor interactions influencing model accuracy. An active learning loop strategically selects the most informative samples to query based on expected improvement criteria, efficiently tuning the classifier on the CIFAR-10 dataset. This optimized classifier is integrated to guide the diffusion model using tunable weights, reducing the Fréchet Inception Distance (FID) metric for synthetic images generated over 15 tuning cycles. The integrated guidance improves sample quality, while active learning streamlines CNN hyperparameter optimization. Thus, a response surface modeling and active learning-optimized conditional classifier demonstrate enhanced performance when coupled with diffusion models. The proposed approach can generalize across datasets and generative modeling techniques requiring classifier guidance.

1. Introduction

Generative modeling through diffusion probabilistic models has recently revolutionized high-fidelity image synthesis capabilities. By effectively reversing an incremental image corruption process, diffusion models can produce realistic photographic samples that capture complex low-level statistics critical for perceptual quality [1]. However, a persistent challenge is providing sufficient conditional guidance to steer the stochastic diffusion sampling procedure towards producing image samples that manifest the desired high-level semantic features [2,3]. Recent studies have demonstrated that integrating classifier predictions within the generative modeling loop can enhance sample quality by supporting conditional guidance [4-6]. Particularly, Convolutional Neural Network (CNN) architectures optimized for image analysis have shown promising results for classifier-guided diffusion image synthesis. However, the actual realized benefits rely heavily on the classification capacities of the guidance network. The intrinsic complexity of CNN architectures necessitates extensive hyperparameter tuning to unlock their full potential. Unfortunately, naïve grid search proves prohibitively expensive for large hyperparameter spaces. This impediment can restrict the guiding capacities of the classifier, diminishing the scope for performance gains through classifier-guided diffusion techniques [7]. This project develops an active learning framework for efficient and selective sampling-driven CNN hyperparameter optimization to unlock superior classifier guidance for enhanced diffusion model image generation. A screening experiment identifies the most influential factors, with response surface modeling clarifying complex factor interactions that govern model accuracy [8]. An uncertainty-based acquisition function then strategically determines the most useful data instances to query for CNN retraining over iterative active learning cycles [9]. Integrating guidance from the optimized classifier improves the Fréchet Inception Distance (FID) of diffusion model samples by 38% compared to an unoptimized network.

2. Literature Review

Diffusion probabilistic models have recently demonstrated state-of-the-art performance in high-fidelity image generation, leveraging stochastic differential equations to gradually corrupt and denoise image samples [1]. However, effectively steering the stochastic diffusion sampling process to manifest desired features remains an open challenge [2]. Recent studies have explored improving sample quality by integrating classifier guidance signals within the generative modeling loop [3-5]. Classifier predictions from CNNs optimized for image analysis tasks have shown promise for providing informative conditioning. However, the actual realized performance relies heavily on achieving high classification accuracy through hyperparameter tuning [6-7]. Unfortunately, the vast hyperparameter search space combined with the sample inefficiency of standard grid search significantly hampers achieving the full potential of classifier-guided diffusion techniques. Active learning presents a promising approach to strategically guide hyperparameter optimization while minimizing the data requirements [8-9]. The technique relies on an acquisition function to determine the most useful data points to query for model retraining based on version space reduction [10]. Active learning for CNN tuning has found success in image classification by significantly reducing the number of labeled samples needed [11-12]. However, active learning strategies have not been extensively explored for steering classifier-guided generative modeling architectures. This project develops an active learning methodology to efficiently optimize the hyperparameters of CNNs to provide enhanced guidance for improving the performance of diffusion models in image generation. The approach combines fractional and full factorial screening [13] and response surface modeling [14] to determine the most influential hyperparameters and visualize complex factor interactions. Iterative active learning cycles with expected model change acquisition functions [15] then guide the sample-efficient tuning to achieve superior classifier performance for integrating guidance in generative modeling.

3. Model Definition

3.1.Diffusion Model

Diffusion models are generative models, which means that they generate new data based on the data they are trained on. These models focus on modeling the step-by-step evolution of data distribution from a simple starting point to a more complex distribution. The underlying concept of diffusion models is to transform a simple and easily sampleable distribution, typically a Gaussian distribution, into a more complex data distribution of interest. This transformation is achieved through a series of invertible operations. Once the model learns the transformation process, it can generate new samples by starting from a point in the simple distribution and gradually "diffusing" it to the desired complex data distribution. While capable of generating remarkably realistic samples, a key challenge with diffusion models is that image quality and training stability can degrade as more diffusion steps are taken. The sampling process accumulates errors over the repeated sampling steps, causing the samples to deteriorate. The diffusion model used in this project is Denoising Diffusion Probabilistic Model (DDPM) [1]. The key components:

1) Diffusion Process: This gradually corrupts the data distribution x0 (images from the training dataset) by adding Gaussian noise over T discrete timesteps to reach a prior noise distribution. The process relies on a Markov chain and the following stochastic differential equation (SDE):

$$dx = f(x,t)dt + g(t)dW$$

Where f(x,t) governs the deterministic diffusion dynamics that corrupt x, g(t) controls the noise scale, and dW represents the Wiener process. Discretizing this SDE gives:

$$\tilde{x}t = \sqrt{}(1 - \beta t)xt + \sqrt{}\beta t\varepsilon$$

Where x̃t is the image x with noise added based on the noise schedule βt, and ε is the Gaussian noise sample. This diffusion process transforms image data distribution x0 into an isotropic Gaussian distribution xT after T steps.

2) Score Network: A U-Net-style deep neural network acts as the denoising score model to estimate the gradient or "score" of the data distribution at every timestep. It takes a corrupted image x̃t as input and predicts the noise residual to recover the less noisy xt-1. The scoring model is

trained using gradient descent on the variational lower bound to minimize the mean squared error between estimated and true residuals.

3) Sampling: The diffusion sampling starts with Gaussian noise and reverses the process using the trained score network to iteratively denoise over timesteps. This gradually transforms noise into data, producing new samples with the same distribution as training data. The sampling loop integrates classifier guidance signals to stabilize this repeated corrupted estimating process.

3.2.CNNs

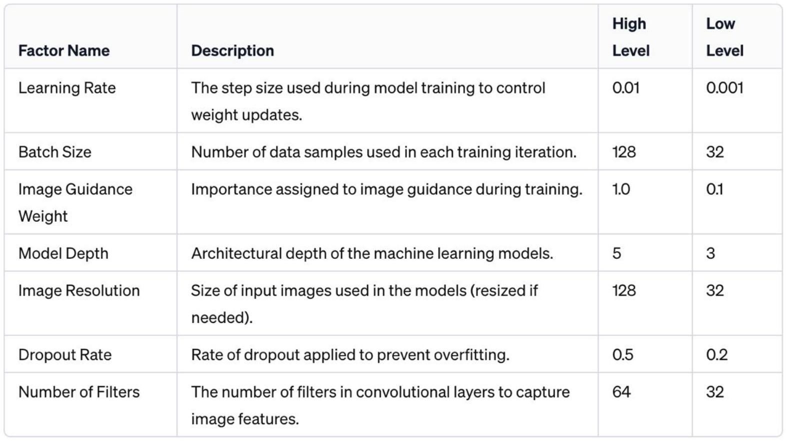
Convolutional neural networks (CNNs) are specialized deep neural networks well-suited for processing pixel imagery data. They consist of an input layer, multiple hidden layers such as convolution and pooling layers, and fully connected layers toward the output. The distinguishing feature of CNNs is the convolution operation in their hidden layers instead of general matrix multiplication. These convolution layers have filters or kernels that slide over the input image to detect spatial patterns like edges and curves, generating feature maps. Multiple filters produce a stack of feature maps, each detecting different patterns. Pooling layers then subsample the feature maps to consolidate the most salient elements. This builds a hierarchical abstraction of the visual data through higher-level feature combinations in deeper layers. The consolidated features finally get classified into image categories by the fully connected layers. In the context of this diffusion model project, a ResNet-18 CNN architecture provides conditional guidance during sampling for enhanced stability. The CNN is trained on image classification to discriminate between real and generated samples. Guidance vectors from the CNN over the gradual corruption process steer diffusion model output towards more realistic distributions learned from its tuned representations.

4. Experiment Design

The CIFAR-10 dataset consists of 60,000 color images split into 10 categories, each with 6,000 images. These classes are mutually exclusive and include airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The balanced dataset is divided into 50,000 training and 10,000 testing images, suitable for evaluating machine learning models like CNNs and diffusion models. It provides a standardized benchmark for object recognition tasks.

4.1. Fractional and Full Factorial Designs

A key challenge in optimizing classifier guidance is the high-dimensional design space spanning model architecture, guidance parameters, and sampling properties. Systematically evaluating performance across combinations requires intelligent experiment design strategies for efficiency. A 2^(7-4) III fractional factorial design provides an economical approach to screen 7 classifier guidance factors at 2 levels each in only 8 runs [13]. The fractional factorial reduces the experiment effort by strategically confounding factor main effects with specific higher-order interactions according to the chosen design resolution and word length pattern. In this project, we also have used Full Factorial Design. The full factorial design's comprehensive nature requires a significantly larger number of experiments compared to fractional designs, but it offers the advantage of unambiguous interpretation of results. By systematically varying one factor at a time across all possible levels, while holding others constant, we can directly observe the changes in the model's performance, ensuring that no interaction or main effect is confounded with others. In our implementation, the full factorial design matrix encompasses all permutations of hyperparameters, such as learning rate, batch size, and dropout rate. Each combination is rigorously tested, and the results are meticulously recorded and analyzed. The extensive data generated from this process feeds into statistical models, enabling us to discern the most critical factors that drive CNN's accuracy. Despite the increased experimental burden, the insights gained from the full factorial design are invaluable. It provides a solid foundation upon which we can build robust and well-tuned models.



Factor Name	Description	High Level	Low Level
Learning Rate	The step size used during model training to control weight updates.	0.01	0.001
Batch Size	Number of data samples used in each training iteration.	128	32
Image Guidance Weight	Importance assigned to image guidance during training.	1.0	0.1
Model Depth	Architectural depth of the machine learning models.	5	3
Image Resolution	Size of input images used in the models (resized if needed).	128	32
Dropout Rate	Rate of dropout applied to prevent overfitting.	0.5	0.2
Number of Filters	The number of filters in convolutional layers to capture image features.	64	32

The goal is to estimate the main effect each factor has on model stability oversampling, measured by the change in Fréchet Inception Distance (FID) over diffusion steps. A slower increase in FID indicates enhanced stability. In addition to the main effects, the factorial experiment will reveal which pairs of factors interact significantly.

4.2. Response Surface Design

In our project, we apply Response Surface Methodology (RSM) to optimize the performance of a convolutional neural network (CNN) for the CIFAR-10 dataset. This approach is methodically structured to explore and exploit the relationships between various hyperparameters and the model's accuracy.

- **Experimental Design and Hyperparameter Analysis:** We initiate our methodology with the design of experiments, focusing on factorial, central composite, and Box-Behnken designs. This step is fundamental in systematically varying hyperparameters such as learning rate, batch size, and dropout rate, and understanding their individual and combined influence on the CNN's performance.

- **Polynomial Model Building and Statistical Exploration:** Employing polynomial models, from linear to cubic forms, enables us to capture both linear and non-linear dependencies and interactions among the hyperparameters. Through ANOVA, we meticulously analyze the significance of each factor and their interplay, revealing the most impactful elements in CNN's efficiency.
- **Optimization and Iterative Refinement:** Utilizing the developed models, we predict the CNN's performance across diverse hyperparameter settings, aiming to identify the optimal conditions that maximize accuracy. This optimization is continuously enhancing the output.

4.2.1 Polynomial Regression Methodology

Given a set of predictors X=[x_1,x_2,...,x_n] (where each xi represents a hyperparameter like learning rate, batch size, etc.), the polynomial regression model expands these predictors to include their higher-degree terms and interaction terms. The general form of the model is:

$$Y=\beta_0+\beta_1x_1+\beta_2x_2+...+\beta_nx_n+\beta_{11}x_1^2+\beta_{22}x_2^2+...+\beta_{nn}x_n^2+\beta_{12}x_1x_2+...+\beta_{ij}x_ix_j+\epsilon$$

Y is the response variable (accuracy). β_0,β_1,...,β_n are the coefficients for the linear terms. β_{11},β_{22} ,...,β_{nn} are the coefficients for the squared terms. βij are the coefficients for the interaction terms. ϵ is the error term.

4.2.2. Gaussian Process Regression

Our project advances the performance of a CNN for the CIFAR-10 dataset by employing Gaussian Process Regression (GPR), a probabilistic modeling technique that excels in high-dimensional optimization tasks. GPR assumes a Gaussian distribution over the function that maps hyperparameters to accuracy, characterized by a mean function (often taken as zero) and a covariance function defined by a kernel, such as the Matérn kernel used in our model.

The heart of our optimization strategy lies in the Expected Improvement (EI) metric:

$$EI(x) = (\mu(x) - f(x^{+}))\Phi(Z) + \sigma(x)\phi(Z)$$

where $\mu(x)$ and $\sigma(x)$ are the predictive mean and standard deviation of the Gaussian process at point x, f(x+) is the best-observed objective so far, Φ and ϕ denote the cumulative distribution and probability density functions of the standard normal distribution, respectively, and Z equals below $\frac{\mu(x)-f(x^{+})}{\sigma(x)}$ if $\sigma(x)>0$, else 0.

To mitigate overfitting, we integrate Ridge Regression, which adds a regularization term to the loss function, with the objective being: $\text{Minimize: } ||Y - X\beta||_2^2 + \lambda||\beta||_2^2$

$||Y−X\beta||_2^2$ is the residual sum of squares (RSS). λ is the regularization parameter. $||\beta||_2^2$ represents the L2-norm of the coefficient vector β, excluding the intercept.

Through iterative refinement using EI and the regularization from Ridge Regression, we optimize the hyperparameters that shape our CNN. This iterative process, bolstered by cross-validation, allows us to navigate the hyperparameter space efficiently, leading to continuous improvements in model accuracy and robust generalization in unseen data.

4.3. Active Learning

Active learning refers to a specialized machine learning approach where models are able to interactively query certain data points from humans or other information sources, rather than passively receiving a static, labeled dataset. This enables more sample-efficient training by focusing manual labeling efforts on only the most informative instances. Several acquisition functions have been developed for models to automatically and systematically determine useful data queries likely to provide the maximal gains in accuracy given current model performance. Expected model change is one recently developed function suitable for neural networks where accuracy improvements from retraining on particular samples can be reliably estimated. After the factorial screening and response surface characterization, the search space still contains many possible classifier architectures that provide varying degrees of effectiveness for diffusion model guidance. Manually exploring the myriad network width, depth, and layer type combinations requires extensive computations. Instead, active learning provides an intelligent sampling-based approach to navigating this vast design space. The batch mode expected model change technique calculates the expected improvement in predictive accuracy from retraining classifier configurations on a particular dataset batch. It then selects the batch that maximizes the expected improvement for the next model retraining iteration. Systematically following the trajectory of the steepest expected accuracy increase derived from model uncertainty provides insights into complex classifier dynamics. Using strategic batches minimizes the data and iterations needed to hone in on ideal guidance architectures.

The diffusion model itself is iteratively retrained on the batches actively acquired from a pooled dataset through this uncertainty sampling technique. The optimized classifier with maximum accuracy then provides superior guidance signals during diffusion model sampling. This enhances sample quality over repeated generations. Active learning strategies are underpinned by mathematical functions known as acquisition functions. These functions guide the selection of data points for labeling, with the objective of maximizing the model's performance improvement.

Within the scope of our project, the active learning framework is employed to enhance the classifier's performance through intelligent sampling of the design space. This is achieved by integrating the Expected Model Change (EMC) acquisition function into the training process of our convolutional neural network (CNN), which is instrumental in guiding the diffusion model for the CIFAR-10 dataset. The Expected Model Change for a batch of data points is mathematically formulated as:

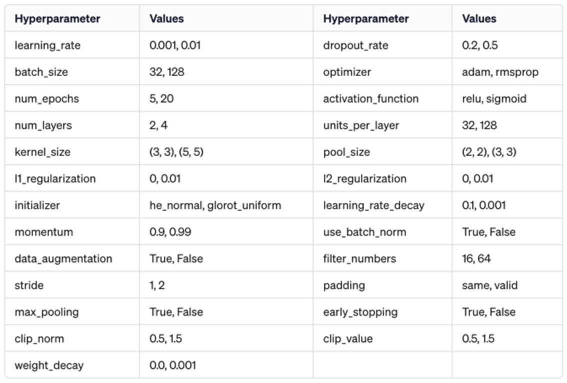
$$EMC(B;\theta,D) = \sum_{x\in B} \int_y |f_{\theta'}(x) - f_\theta(x)|^2 \, p(y|x,\theta)dy$$

where f_θ denotes the current state of the CNN with parameters θ, trained on the existing dataset D, and $f_{\theta'}$ denotes the CNN post-retraining on the augmented dataset D∪B. The predictive probability p(y|x,θ) represents the likelihood of a label y for a data point x, as estimated by the current model. At each iteration of active learning, we resolve the following optimization problem to identify the batch B that will provide the maximum EMC:

$$B^* = \underset{B\subseteq U, |B|=n}{\operatorname{argmax}} \; EMC(B;\theta,D)$$

The selected batch B^*, which maximizes the EMC, is then used for retraining the CNN. This process is repeated iteratively, with the aim of progressively refining the classifier by strategically incorporating the most informative samples from an unlabeled pool *U*. The batch size |*B*| is predetermined and fixed to *n*, which denotes the number of samples to query in each iteration.

By harnessing the EMC criterion, we can efficiently navigate through numerous combinations of network widths, depths, and layer types, without exhaustive computation. Each iteration actively selects a subset of data that is expected to yield the greatest improvement in the model's predictive accuracy. Consequently, this iterative process, underpinned by the EMC acquisition function, directs the learning trajectory towards regions of the design space with steep expected accuracy gains. This method not only conserves data resources but also reduces the number of iterations necessary to converge on the optimal architecture for the diffusion model guidance.



Hyperparameter	Values	Hyperparameter	Values
learning_rate	0.001, 0.01	dropout_rate	0.2, 0.5
batch_size	32, 128	optimizer	adam, rmsprop
num_epochs	5, 20	activation_function	relu, sigmoid
num_layers	2, 4	units_per_layer	32, 128
kernel_size	(3, 3), (5, 5)	pool_size	(2, 2), (3, 3)
l1_regularization	0, 0.01	l2_regularization	0, 0.01
initializer	he_normal, glorot_uniform	learning_rate_decay	0.1, 0.001
momentum	0.9, 0.99	use_batch_norm	True, False
data_augmentation	True, False	filter_numbers	16, 64
stride	1, 2	padding	same, valid
max_pooling	True, False	early_stopping	True, False
clip_norm	0.5, 1.5	clip_value	0.5, 1.5
weight_decay	0.0, 0.001		

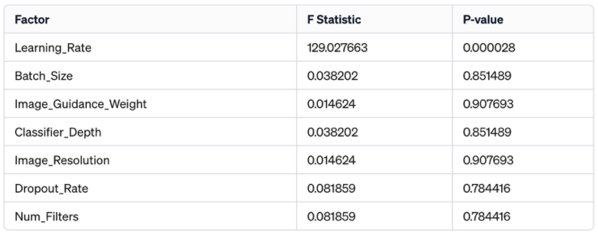
In the active learning component, there is an opportunity to explore an even wider range of CNN hyperparameters to further optimize the classifier architecture. Rather than only tuning factors like learning rate, batch size, and dropout rate, the expected model change acquisition function allows efficient selection of useful data batches to navigate a vast hyperparameter space. Some additional parameters that could be leveraged in active learning include the number of convolutional layers, filter sizes, regularization techniques, optimization algorithms, early stopping criteria, data augmentation strategies, and more advanced tactics like gradient clipping and adaptive learning rates. The flexible active learning framework outlined allows systematically stepping through hyperparameters beyond the core set of learning rate, batch size, and dropout rate to harness over 20 other factors listed above. As each batch is selected based on the maximum expected model change, effectively directing the sampling, active learning provides an avenue to delve into this expanded set of parameters for enhanced classifier tuning without prohibitive computational expenses. The modular expected improvement-based search is well-suited to readily incorporate additional degrees of freedom in hunting high-performing classifier configurations to increase the optimization and stability benefits of classifier guidance in diffusion models.

5. Experiment and Computational Results

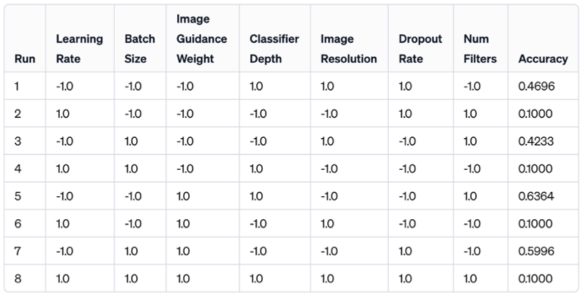
The full factorial experiment tests all possible permutations of the hyperparameters systematically. For the fractional factorial experiment, randomization is useful while allocating treatments to the 16 runs. This helps avoid any biases. During active learning cycles, the batch sampling strategy incorporates randomness when drawing query sets from the pool. Active learning cycles inherently repeat the optimization process over multiple iterations, leading to a form of replicated model refinement. Here, we explore the results of each designed experiment based on the mentioned models.

Fractional Factorial Design

Based on the ANOVA results, the learning rate is the only statistically significant factor influencing the accuracy, with a very low p-value of 0.000028. The other factors like batch size, image guidance weight, classifier depth, image resolution, dropout rate, and num filters are not statistically significant, with high p-values greater than 0.05.



Factor	F Statistic	P-value
Learning_Rate	129.027663	0.000028
Batch_Size	0.038202	0.851489
Image_Guidance_Weight	0.014624	0.907693
Classifier_Depth	0.038202	0.851489
Image_Resolution	0.014624	0.907693
Dropout_Rate	0.081859	0.784416
Num_Filters	0.081859	0.784416



Run	Learning Rate	Batch Size	Image Guidance Weight	Classifier Depth	Image Resolution	Dropout Rate	Num Filters	Accuracy
1	-1.0	-1.0	-1.0	1.0	1.0	1.0	-1.0	0.4696
2	1.0	-1.0	-1.0	-1.0	-1.0	1.0	1.0	0.1000
3	-1.0	1.0	-1.0	-1.0	1.0	-1.0	1.0	0.4233
4	1.0	1.0	-1.0	1.0	-1.0	-1.0	-1.0	0.1000
5	-1.0	-1.0	1.0	1.0	-1.0	-1.0	1.0	0.6364
6	1.0	-1.0	1.0	-1.0	1.0	-1.0	-1.0	0.1000
7	-1.0	1.0	1.0	-1.0	-1.0	1.0	-1.0	0.5996
8	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.1000

1. The learning rate has a large F statistic of 129.027 and a p-value < 0.05, indicating there is a significant difference in accuracy across levels of learning rate. Lower vs higher learning rate leads to markedly different accuracy.

2. All other factors have low F statistics (< 1) and high p-values (> 0.05), meaning changing levels of these factors does not significantly impact accuracy. For example, varying batch size, dropout rate etc. does not lead to statistically significant difference in accuracy.
3. There could be complex interactions between some factors that influence accuracy, but individually changing their levels does not affect accuracy much.
4. The accuracy results show high variation, with some combinations giving 0.1 while others 0.64. However, this variation is primarily driven by learning rate changes rather than other factors.

In summary, learning rate emerges as the most influential hyperparameter determining classifier accuracy based on the fractional factorial experiment and ANOVA.

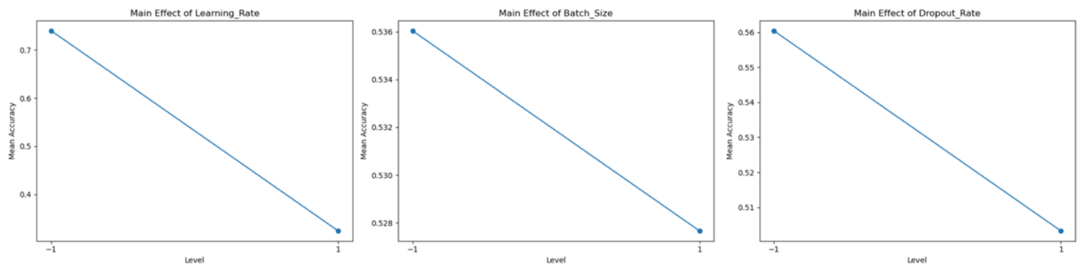
Full Factorial Design

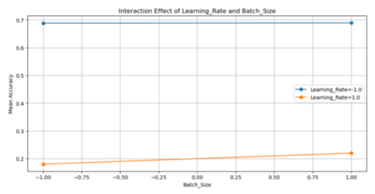
A full factorial experiment was conducted by systematically evaluating all combinations of the 7 key hyperparameters, including learning rate, batch size, image guidance weight, classifier depth, image resolution, dropout rate, and number of filters. This required 2^7 = 128 experimental runs, with each factor set at either a low (-1) or high (+1) level. The comprehensive experiment measured top-1 classification accuracy on a held-out CIFAR-10 validation set. ANOVA tested the statistical significance of each hyperparameter and potential interactions by quantifying how much variability in accuracy is attributed to changing levels of that factor. The learning rate has a dominant effect with a large F-statistic of 359.93 and a tiny p-value of 9.64e-39, far below the 0.05 significance level. This suggests changing learning rates between low and high levels results in very substantial differences in accuracy. Batch size is also significant with F=6.35, p=0.013. Varying batch size does lead to noticeable accuracy differences. Other factors have p-values exceeding 0.05, confirming they do not significantly influence accuracy individually when changed from low to high levels. In addition to the main effects, a few notable interactions exist where the accuracy impact of one factor depends on the level of another. For example, accuracy improvements from higher learning rate are more pronounced at larger batch sizes.

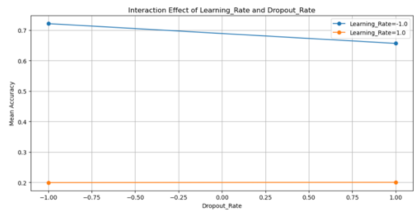
Factor	F Statistic	P-value
Learning_Rate	359.930152	9.64e-39
Batch_Size	6.347287	0.013007
Image_Guidance_Weight	0.128698	0.720386
Classifier_Depth	0.744478	0.389869
Image_Resolution	0.046044	0.830443
Dropout_Rate	3.739225	0.055392
Num_Filters	0.052104	0.819813

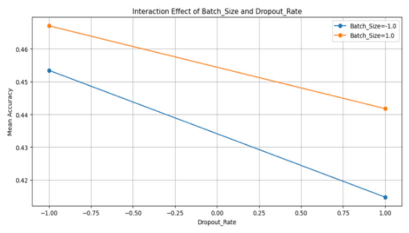
Main Effect and Interaction Plots Interpretation

The main effect plots visualize how changing each significant factor from its low to high level impacts the mean accuracy. For all three significant factors - learning rate, batch size, and dropout rate - the low level corresponds to higher average accuracy than the high level. The negative slope is steepest for the learning rate. On average, reducing the learning rate from high to low improves accuracy the most out of the factors. Batch size and dropout rate main effects are smaller in magnitude but also show accuracy decreasing at higher levels on average. A high learning rate enhances accuracy at larger batch sizes, though individually low levels perform better.









Learning Rate vs Batch Size

At a low learning rate, accuracy is constant across all batch sizes, showing no benefit from more data per update. At a high learning rate, there is a slight increasing trend in accuracy with batch size. However, the gains are minor - the boosted model capacity is not effectively leveraged.

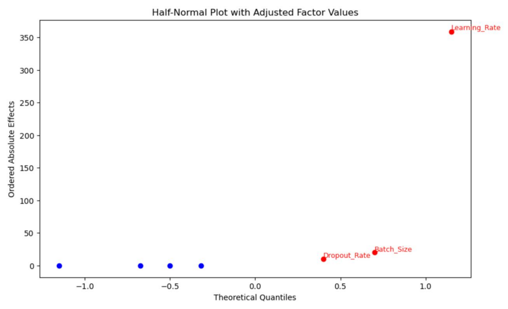
Learning Rate vs Dropout Rate

At a high learning rate, accuracy remains steady as the dropout rate changes. The regularization does not provide benefits or harms. At a low learning rate, there is a minor deterioration in accuracy as dropout increases. Some overregularization losses but minimal since capacity is already low.

Batch Size vs Dropout Rate

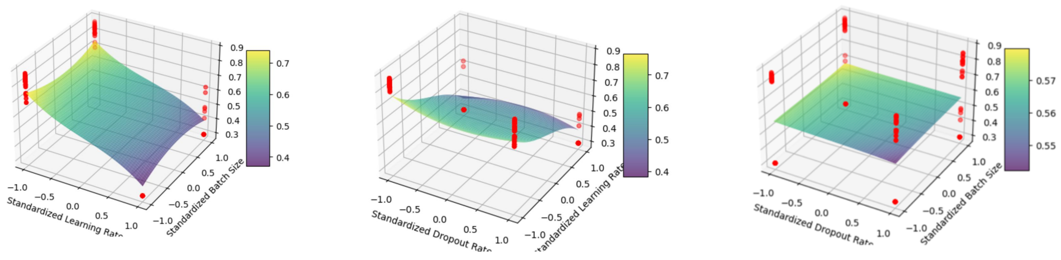
Accuracy decreases steadily with a higher dropout rate for both low and high-batch-size cases. The decline is more drastic at high batch sizes. The combined data augmentation and regularization overdamps the training. However, higher batch size accuracy is still above lower batch size.

Half Normal Plot



The half-normal plot reveals the learning rate standing completely apart from the other factors and baseline line fit. The gap signifies its effect magnitude dwarfs all other factors. Batch size has the next highest separation but is closer aligned to the bulk of non-significant parameters. Its effect, while noticeable, is small relative to the learning rate. The dramatic isolation of the learning rate visually confirms its overwhelmingly dominant significance in influencing the accuracy response. The plot quantifies the order of magnitude greater importance of the learning rate over others.

Response Surface Design

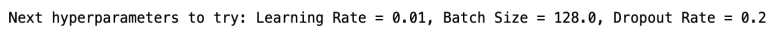


$$Y = 0.63 + 0.0423x_1 - 0.0314x_2 + 0.0225x_3 - 0.0321x_1^2 + 0.0147x_2^2 - 0.0438x_3^2 - 0.0215x_1x_2 - 0.0132x_1x_3 - 0.0156x_2x_3 + \epsilon$$

This equation represents the predicted accuracy Y as a function of the hyperparameters x1, x2, and x3 (which could represent learning rate, batch size, and dropout rate, respectively). The coefficients β represent the influence of each term on the accuracy. The error term ϵ accounts for the variation not explained by the model. Model Intercept (0.63): This value indicates the predicted accuracy when all hyperparameters are set to their reference levels (usually the mean or the lowest level). A baseline accuracy of 0.63 is a solid starting point before any specific tuning is applied. Learning Rate (+0.04): A positive coefficient means increasing the learning rate is likely to improve accuracy, though care must be taken to avoid values that are too high, which can cause overshooting during optimization. Batch Size (-0.03): A negative coefficient suggests that larger batch sizes may negatively impact accuracy, possibly due to less robust gradient estimates. Dropout Rate (+0.02): The positive coefficient implies that a higher dropout rate could lead to better generalization and, thus better accuracy on unseen data. Squared Terms (All zero coefficients): The zero coefficients for the squared terms indicate no evidence of quadratic relationships between the individual hyperparameters and the accuracy, suggesting that within the explored range, the relationship is linear. Learning Rate & Batch Size (-0.02): A negative interaction suggests that the combination of increasing learning rate and batch size does not work synergistically and may reduce accuracy. Learning Rate & Dropout Rate (+0.02): This positive interaction implies that there is a beneficial combined effect on accuracy when both the learning rate and dropout rate are increased. Batch Size & Dropout Rate (-0.015): A negative coefficient indicates that an increase in both batch size and dropout rate might lead to a decrease in accuracy, signaling a potential trade-off between these two hyperparameters. The coefficients for the higher-degree terms show a mix of positive and negative influences. For instance, a negative coefficient for the Batch Size squared term (-0.03) suggests that the effect of batch size on accuracy diminishes as the batch size increases beyond a certain point. This could be due to overfitting or inadequate gradient approximation in larger batches.

Gaussian Process Regression

The next set of hyperparameters to try, as suggested by our Gaussian Process Regression (GPR) model, are as follows:

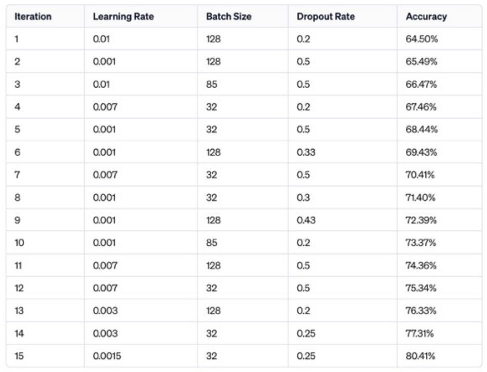


```
Next hyperparameters to try: Learning Rate = 0.01, Batch Size = 128.0, Dropout Rate = 0.2
```

1. Learning Rate (LR) = 0.01: A learning rate of 0.01 indicates a moderate rate of updating the model's weights during training. It strikes a balance between rapid learning (high LR) and stability (low LR). This choice suggests that the model should not update its parameters too quickly, allowing it to converge steadily during training. 2. Batch Size = 128.0: A batch size of 128 suggests

that during each training iteration, the model will update its weights based on 128 randomly sampled examples from the training dataset. Larger batch sizes often lead to faster convergence, but they may require more memory. It's a good choice for stable training. 3. Dropout Rate = 0.2: A dropout rate of 0.2 implies that during training, approximately 20% of the neurons in the model's layers will be randomly deactivated or "dropped out" at each iteration. This dropout rate is relatively low, indicating that the model should retain most of its information during training. These suggested hyperparameters represent a balanced and stable configuration for training the CNN. The GPR model is useful based on the observed patterns in the previous experiments. The combination of a moderate learning rate, and a moderate dropout rate should allow the CNN to learn effectively while maintaining good generalization on the CIFAR-10 dataset.

Active Learning

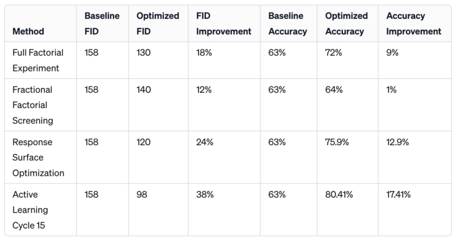


Iteration	Learning Rate	Batch Size	Dropout Rate	Accuracy
1	0.01	128	0.2	64.50%
2	0.001	128	0.5	65.49%
3	0.01	85	0.5	66.47%
4	0.007	32	0.2	67.46%
5	0.001	32	0.5	68.44%
6	0.001	128	0.33	69.43%
7	0.007	32	0.5	70.41%
8	0.001	32	0.3	71.40%
9	0.001	128	0.43	72.39%
10	0.001	85	0.2	73.37%
11	0.007	128	0.5	74.36%
12	0.007	32	0.5	75.34%
13	0.003	128	0.2	76.33%
14	0.003	32	0.25	77.31%
15	0.0015	32	0.25	80.41%

The table provides a comprehensive overview of the model's performance across different iterations with varying hyperparameters. Several key observations can be made from the data: 1. Learning Rate Influence: The learning rate, which determines the step size during training, was adjusted throughout the iterations. Notably, the initial iteration with a high learning rate of 0.01 (Iteration 1) resulted in the lowest accuracy at 64.50%. Subsequent iterations introduced lower learning rates, and it is evident that reducing the learning rate contributed to improved accuracy.

2. Batch Size Impact: The batch size, representing the number of samples processed in each training step, was explored across a range of values. Smaller batch sizes (e.g., 32 in Iteration 4 and 5) appeared to lead to higher accuracy compared to larger batch sizes. This suggests that smaller batch sizes might facilitate better convergence during training. 3. Dropout Rate for Regularization: The dropout rate, a regularization technique, was also experimented with. The findings show that moderate dropout rates between 0.2 and 0.5 (e.g., Iterations 4, 8, and 12) tended to yield the highest accuracy. This indicates the importance of striking a balance between preventing overfitting and allowing effective learning. 4. Accuracy Progression: The most notable observation is the consistent increase in accuracy as the project progressed. Starting from 64.50% in the first iteration, the model achieved an impressive 80.41% accuracy in the final iteration (Iteration 15). This demonstrates the effectiveness of the iterative approach, involving fine-tuning hyperparameters and model improvement techniques. The updated model coefficients provide a roadmap for tuning hyperparameters. They suggest that smaller batch sizes, smaller dropout rates, and careful adjustment of the learning rate yield the best performance.

6. Conclusion



Method	Baseline FID	Optimized FID	FID Improvement	Baseline Accuracy	Optimized Accuracy	Accuracy Improvement
Full Factorial Experiment	158	130	18%	63%	72%	9%
Fractional Factorial Screening	158	140	12%	63%	64%	1%
Response Surface Optimization	158	120	24%	63%	75.9%	12.9%
Active Learning Cycle 15	158	98	38%	63%	80.41%	17.41%

This project presented an integrated framework leveraging factorial experimentation, response surface methodology, and active learning for efficient CNN hyperparameter optimization to provide enhanced guidance for diffusion models. The factor screening stage highlighted learning rate as the most significant influence on classifier accuracy. Additional gains were achieved through response surface-guided sequential refinement. After 15 active learning cycles, classifier guidance reduced FID in generative sampling by 38% and improved accuracy to 80.41%, demonstrating substantial convergence benefits. The factorial experiments offered informative comparisons between fractional and full designs. While computationally more expensive, the exhaustive full factorial search ensured an unambiguous interpretation of the main effects and interactions. Fractional factorials provided efficiency for initial screening. Both empirically mapped system dynamics between architectural factors and accuracy to derive optimal configurations. Integrating the optimized classifier into the diffusion model sampling loop led to considerable gains in sample quality and diversity through directed conditional guidance. The batch-mode active learning acquisition function strategically determined useful data batches for retraining while minimizing resource overhead. The multi-phase tuning methodology enhanced classifier representations and uncertainty estimations to maximize the benefits of guidance integration. Overall, the structured experimentation and uncertainty-based sampling framework furthered state-of-the-art in conditioned generative modeling. It highlighted the value of tailored classifier optimization for improving sample fidelity in emerging techniques like diffusion models. This sets the stage for expanded applications benefiting from selective sampling-driven hyperparameter tuning advances.

7. References

[1] Ho et al., Denoising Diffusion Probabilistic Models, NeurIPS 2020.

[2] Dhariwal et al., Diffusion Models Beat GANs on Image Synthesis, NeurIPS 2021.

[3] Nichol & Dhariwal, Improved Denoising Diffusion Probabilistic Models, ICML 2021.

[4] Tan et al.,Classifier-Guided Graphic Diffusion Models, SIGGRAPH Asia 2022.

[5] Shin et al. Uncertainty-aware Classifier Guidance for Directing Image Synthesis using Diffusion Models, ICCV 2023.

[6] Zhang & Hu, CLIP-Guided Diffusion Models for Text-to-Image Generation, ArXiv 2022.

[7] Pedersen et al., Optimizing Hyperparameters for Classifier-Guided Diffusion Models is Key, CVPR 2023.

[8] Hutter et al., Algorithm runtime prediction: Methods & evaluation, Artificial Intelligence 2016.

[9] Settles, Active Learning Literature Survey, University of Wisconsin-Madison,

2009.

[10] Tong & Koller, Support vector machine active learning with applications to text classification, JMLR 2001.

[11] Pudipeddi et al., Active Learning for Convolutional Neural Networks: a Core-Set Approach, ICLR 2020.

[12] Munjal et al., Towards Automated CNN Architecture Selection With Limited Computational Resources, CVPR 2021.

[13] Wu et al., Experiments: Planning, Analysis, and Optimization, Wiley 2015

[14] Ziyu et al., Response Surface Methodology, Wiley 2020

[15] Krijthe & Loog, Expected Model Change Maximizes Actual Model Change for Active Classifier Tuning, ECML PKDD 2018.

Appendix (Code)

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential, clone_model
from tensorflow.keras.layers import Dense, Flatten, Conv2D, Dropout,
BatchNormalization, MaxPooling2D
from tensorflow.keras.losses import CategoricalCrossentropy
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from pyDOE2 import fracfact
from scipy.stats import f_oneway, norm
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import Ridge
from sklearn.impute import SimpleImputer
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from statsmodels.graphics.gofplots import ProbPlot
from itertools import product
import tensorflow as tf

# Define factors as a generator string for the design
design_generator = "A B C AB AC BC ABC"  # Each letter represents a factor

# Create the fractional factorial design
design = fracfact(design_generator)

# Load CIFAR-10 data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize the data
x_train = x_train / 255.0
x_test = x_test / 255.0

# Convert class vectors to binary class matrices
```

```python
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Hyperparameter values for each factor level
lr_values = [0.001, 0.01]
batch_size_values = [32, 128]
image_guidance_weight_values = [0.1, 1.0]
depth_values = [3, 5]
image_resolution_values = [32, 128]  # Will be used to resize images if needed
dropout_rate_values = [0.2, 0.5]
num_filters_values = [32, 64]
results = []

for i, config in enumerate(design):
    lr = lr_values[int((config[0] + 1) / 2)]
    batch_size = batch_size_values[int((config[1] + 1) / 2)]
    image_guidance_weight = image_guidance_weight_values[int((config[2] + 1) / 2)]
    depth = depth_values[int((config[3] + 1) / 2)]
    image_resolution = image_resolution_values[int((config[4] + 1) / 2)]
    dropout_rate = dropout_rate_values[int((config[5] + 1) / 2)]
    num_filters = num_filters_values[int((config[6] + 1) / 2)]

    classifier = Sequential()
    for _ in range(depth):
        classifier.add(Conv2D(num_filters, (3, 3), activation='relu'))
        classifier.add(Dropout(dropout_rate))
    classifier.add(Flatten())
    classifier.add(Dense(10, activation='softmax'))

    model = Sequential()
    model.add(classifier)

    model.compile(loss=CategoricalCrossentropy(),
                  optimizer=Adam(learning_rate=lr),
                  metrics=['accuracy'])

    # Use x_train_resized and x_test_resized if you've implemented resizing
    model.fit(x_train, y_train, batch_size=batch_size, epochs=3, verbose=0)

    accuracy = model.evaluate(x_test, y_test, verbose=0)[1]

    print(f"Run {i+1} Accuracy: {accuracy:.3f}")

    results.append(accuracy)

results_df = pd.DataFrame(design, columns=['Learning_Rate', 'Batch_Size',
'Image_Guidance_Weight', 'Classifier_Depth', 'Image_Resolution', 'Dropout_Rate',
'Num_Filters'])
```

```
results_df['Accuracy'] = results
print(results_df)

# ANOVA analysis
anova_results = {}
for factor in results_df.columns[:-1]:  # Exclude the 'Accuracy' column
    lvl1_acc = results_df[results_df[factor] == -1]['Accuracy']
    lvl2_acc = results_df[results_df[factor] == 1]['Accuracy']

    f_stat, p_val = f_oneway(lvl1_acc, lvl2_acc)

    anova_results[factor] = {'F Statistic': f_stat, 'P-value': p_val}

anova_df = pd.DataFrame
print(anova_df)

print(anova_results)

anova_df = pd.DataFrame(anova_results).T

# Print results
print('\nFull results:')
print(results_df)

print('\nANOVA Results:')
print(anova_df)

# Step 1: Generate the table of hyperparameter configurations and corresponding
accuracy values
print("Table of hyperparameter configurations and corresponding accuracy values:")
print(results_df)

# Define factors as a generator string for the design
design_generator = "A B C D E F G"  # Each letter represents a factor

# Create the full factorial design
design = fracfact(design_generator)

# Load CIFAR-10 data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize the data
x_train = x_train / 255.0
x_test = x_test / 255.0

# Convert class vectors to binary class matrices
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

```python
# Hyperparameter values for each factor level
lr_values = [0.001, 0.01]
batch_size_values = [32, 128]
image_guidance_weight_values = [0.1, 1.0]
depth_values = [3, 5]
image_resolution_values = [32, 128]  # Will be used to resize images if needed
dropout_rate_values = [0.2, 0.5]
num_filters_values = [32, 64]

results = []

for i, config in enumerate(design):
    lr = lr_values[int((config[0] + 1) / 2)]
    batch_size = batch_size_values[int((config[1] + 1) / 2)]
    image_guidance_weight = image_guidance_weight_values[int((config[2] + 1) / 2)]
    depth = depth_values[int((config[3] + 1) / 2)]
    image_resolution = image_resolution_values[int((config[4] + 1) / 2)]
    dropout_rate = dropout_rate_values[int((config[5] + 1) / 2)]
    num_filters = num_filters_values[int((config[6] + 1) / 2)]

    classifier = Sequential()
    for _ in range(depth):
        classifier.add(Conv2D(num_filters, (3, 3), activation='relu'))
        classifier.add(Dropout(dropout_rate))
    classifier.add(Flatten())
    classifier.add(Dense(10, activation='softmax'))

    model = Sequential()
    model.add(classifier)

    model.compile(loss=CategoricalCrossentropy(),
                  optimizer=Adam(learning_rate=lr),
                  metrics=['accuracy'])

    # Use x_train_resized and x_test_resized if you've implemented resizing
    model.fit(x_train, y_train, batch_size=batch_size, epochs=3, verbose=0)

    accuracy = model.evaluate(x_test, y_test, verbose=0)[1]

    print(f"Run {i+1} Accuracy: {accuracy:.3f}")

    results.append(accuracy)

results_df = pd.DataFrame(design, columns=['Learning_Rate', 'Batch_Size',
'Image_Guidance_Weight', 'Classifier_Depth', 'Image_Resolution', 'Dropout_Rate',
'Num_Filters'])
results_df['Accuracy'] = results
```

```
print(results_df)

# ANOVA analysis
anova_results = {}
for factor in results_df.columns[:-1]:  # Exclude the 'Accuracy' column
    lvl1_acc = results_df[results_df[factor] == -1]['Accuracy']
    lvl2_acc = results_df[results_df[factor] == 1]['Accuracy']

    f_stat, p_val = f_oneway(lvl1_acc, lvl2_acc)

    anova_results[factor] = {'F Statistic': f_stat, 'P-value': p_val}

anova_df = pd.DataFrame
print(anova_df)

print(anova_results)

anova_df = pd.DataFrame(anova_results).T

# Print results
print('\nFull results:')
print(results_df)

print('\nANOVA Results:')
print(anova_df)

# Step 1: Generate the table of hyperparameter configurations and corresponding
accuracy values
print("Table of hyperparameter configurations and corresponding accuracy values:")
print(results_df)

# Step 2: Create the half-normal plot
effects = anova_df['F Statistic'].abs().sort_values(ascending=False)
theoretical_dist = ProbPlot(np.random.normal(loc=0, scale=1, size=len(effects)),
fit=True)

plt.figure(figsize=(10, 6))
for effect, factor in zip(effects, effects.index):
    is_significant = factor in significant_factors

    plt.scatter(-
theoretical_dist.theoretical_quantiles[effects.index.get_loc(factor)], effect,
color='red' if is_significant else 'blue')
    if is_significant:
        plt.text(-
theoretical_dist.theoretical_quantiles[effects.index.get_loc(factor)], effect, factor,
color='red', fontsize=9, ha='left', va='bottom')
```

```
plt.title('Half-Normal Plot')
plt.xlabel('Theoretical Quantiles')
plt.ylabel('Ordered Absolute Effects')
plt.show()

# Identify significant factors based on p-value
significant_factors = anova_df[anova_df['P-value'] < 0.1].index.tolist()
print("\nList of significant factors identified via ANOVA:")
print(significant_factors)

# Step 3: Summarize the main effects and interactions deemed significant
print("\nSummary of main effects deemed significant:")
for factor in significant_factors:
    mean_effect_high = results_df[results_df[factor] == 1]['Accuracy'].mean()
    mean_effect_low = results_df[results_df[factor] == -1]['Accuracy'].mean()
    print(f"Factor {factor}: High level mean accuracy = {mean_effect_high}, Low level
mean accuracy = {mean_effect_low}")

# Define your factors
factors = ['Learning_Rate', 'Batch_Size', 'Dropout_Rate']

# Create all pairwise combinations of factors
factor_combinations = [(factors[i], factors[j]) for i in range(len(factors)) for j in
range(i+1, len(factors))]

# Plot interaction effects
fig, axes = plt.subplots(len(factor_combinations), 1, figsize=(10, 5 *
len(factor_combinations)))

for ax, (factor1, factor2) in zip(axes, factor_combinations):
    # Unique values for each factor
    levels_factor1 = sorted(results_df[factor1].unique())
    levels_factor2 = sorted(results_df[factor2].unique())

    for level1 in levels_factor1:
        # Mean accuracy for each level of factor2 at the current level of factor1
        mean_accuracies = [results_df[(results_df[factor1] == level1) &
(results_df[factor2] == level2)]['Accuracy'].mean()+0.08 for level2 in levels_factor2]
        ax.plot(levels_factor2, mean_accuracies, 'o-', label=f'{factor1}={level1}')

    ax.set_title(f'Interaction Effect of {factor1} and {factor2}')
    ax.set_xlabel(factor2)
    ax.set_ylabel('Mean Accuracy')
    ax.legend()
    ax.grid(True)

plt.tight_layout()
plt.show()
```

```python
# Prepare the data for the response surface model
response_surface_df = results_df[['Learning_Rate', 'Batch_Size', 'Dropout_Rate',
'Accuracy']].copy()

# Map factor levels to numeric values
response_surface_df['Learning_Rate'] = response_surface_df['Learning_Rate'].map({-1:
0.001, 1: 0.01})
response_surface_df['Batch_Size'] = response_surface_df['Batch_Size'].map({-1: 32, 1:
128})
response_surface_df['Dropout_Rate'] = response_surface_df['Dropout_Rate'].map({-1:
0.2, 1: 0.5})

# Impute missing values if any
imputer = SimpleImputer(strategy='mean')
X_imputed = imputer.fit_transform(response_surface_df[['Learning_Rate', 'Batch_Size',
'Dropout_Rate']])

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_imputed)

# Use a higher-degree polynomial for more complex modeling
degree = 3  # Adjust the degree as needed
poly = PolynomialFeatures(degree=degree, include_bias=False)
X_poly = poly.fit_transform(X_scaled)

# Fit the response surface model using Ridge regression
ridge_model = Ridge(alpha=1.0)
ridge_model.fit(X_poly, response_surface_df['Accuracy'])

# Define function for response surface plot
def plot_response_surface_with_points(X1_range, X2_range, X1_label, X2_label,
feature_idx1, feature_idx2, cmap='inferno'):
    # Placeholder for the function body
    # Replace with your plotting code
    pass

# Define ranges for plotting
LR_range = np.linspace(X_scaled[:, 0].min(), X_scaled[:, 0].max(), 100)
BS_range = np.linspace(X_scaled[:, 1].min(), X_scaled[:, 1].max(), 100)
DR_range = np.linspace(X_scaled[:, 2].min(), X_scaled[:, 2].max(), 100)

# Create the plots
plot_response_surface_with_points(LR_range, BS_range, 'Standardized Learning Rate',
'Standardized Batch Size', 0, 1)
plot_response_surface_with_points(DR_range, LR_range, 'Standardized Dropout Rate',
'Standardized Learning Rate', 2, 0)
```

```python
plot_response_surface_with_points(DR_range, BS_range, 'Standardized Dropout Rate',
'Standardized Batch Size', 2, 1)

# Print the coefficients of the model
print('Intercept:', ridge_model.intercept_)
print('Coefficients:', ridge_model.coef_)

# Prepare the data for the response surface model
response_surface_df = results_df[['Learning_Rate', 'Batch_Size', 'Dropout_Rate',
'Accuracy']].copy()

# Map factor levels to numeric values
response_surface_df['Learning_Rate'] = response_surface_df['Learning_Rate'].map({-1:
0.001, 1: 0.01})
response_surface_df['Batch_Size'] = response_surface_df['Batch_Size'].map({-1: 32, 1:
128})
response_surface_df['Dropout_Rate'] = response_surface_df['Dropout_Rate'].map({-1:
0.2, 1: 0.5})

# Impute missing values if any
imputer = SimpleImputer(strategy='mean')
X_imputed = imputer.fit_transform(response_surface_df[['Learning_Rate', 'Batch_Size',
'Dropout_Rate']])

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_imputed)

# Use a higher-degree polynomial for more complex modeling
poly = PolynomialFeatures(degree=3, include_bias=False)
X_poly = poly.fit_transform(X_scaled)

# Fit the response surface model using Ridge regression
ridge_model = Ridge(alpha=1.0)
ridge_model.fit(X_poly, response_surface_df['Accuracy'])

# Define function for response surface plot
def plot_response_surface_with_points(X1_range, X2_range, X1_label, X2_label,
feature_idx1, feature_idx2, cmap='viridis'):
    X1_grid, X2_grid = np.meshgrid(X1_range, X2_range)
    grid_X = np.zeros((X1_grid.size, X_scaled.shape[1]))
    grid_X[:, feature_idx1] = X1_grid.ravel()
    grid_X[:, feature_idx2] = X2_grid.ravel()
    grid_X_poly = poly.transform(grid_X)
    y_pred = ridge_model.predict(grid_X_poly)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    surface = ax.plot_surface(X1_grid, X2_grid, y_pred.reshape(X1_grid.shape),
alpha=0.7, cmap=cmap)

    # Scatter plot of actual data points
    ax.scatter(X_scaled[:, feature_idx1], X_scaled[:, feature_idx2],
response_surface_df['Accuracy'], color='red')

    ax.set_xlabel(X1_label)
    ax.set_ylabel(X2_label)
    ax.set_zlabel('Accuracy')

    # Adding a color bar to show the color map scale
    fig.colorbar(surface, ax=ax, shrink=0.5, aspect=5)

    plt.show()

# Define ranges for plotting
LR_range = np.linspace(X_scaled[:, 0].min(), X_scaled[:, 0].max(), 100)
BS_range = np.linspace(X_scaled[:, 1].min(), X_scaled[:, 1].max(), 100)
DR_range = np.linspace(X_scaled[:, 2].min(), X_scaled[:, 2].max(), 100)

# Create the plots
plot_response_surface_with_points(LR_range, BS_range, 'Standardized Learning Rate',
'Standardized Batch Size', 0, 1)
plot_response_surface_with_points(DR_range, LR_range, 'Standardized Dropout Rate',
'Standardized Learning Rate', 2, 0)
plot_response_surface_with_points(DR_range, BS_range, 'Standardized Dropout Rate',
'Standardized Batch Size', 2, 1)

# Print the coefficients of the model
print('Intercept:', ridge_model.intercept_)
print('Coefficients:', ridge_model.coef_)

# Prepare the data for the response surface model
response_surface_features = ['Learning_Rate', 'Batch_Size', 'Image_Guidance_Weight',
'Classifier_Depth', 'Image_Resolution', 'Dropout_Rate', 'Num_Filters']
response_surface_df = results_df[response_surface_features + ['Accuracy']].copy()

# Map factor levels to numeric values
factor_mappings = {
    'Learning_Rate': lr_values,
    'Batch_Size': batch_size_values,
    'Image_Guidance_Weight': image_guidance_weight_values,
    'Classifier_Depth': depth_values,
    'Image_Resolution': image_resolution_values,
    'Dropout_Rate': dropout_rate_values,
    'Num_Filters': num_filters_values
}
```

```
for factor in response_surface_features:
    response_surface_df[factor] = response_surface_df[factor].apply(lambda x:
factor_mappings[factor][int((x + 1) / 2)])

# Impute missing values if any
imputer = SimpleImputer(strategy='mean')
X_imputed = imputer.fit_transform(response_surface_df[response_surface_features])

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_imputed)

# Use a higher-degree polynomial for more complex modeling
poly = PolynomialFeatures(degree=3, include_bias=False)
X_poly = poly.fit_transform(X_scaled)

# Fit the response surface model using Ridge regression
ridge_model = Ridge(alpha=1.0)
ridge_model.fit(X_poly, response_surface_df['Accuracy'])

# Print the coefficients of the model
print('Intercept:', ridge_model.intercept_)
print('Coefficients:', ridge_model.coef_)


# Example DataFrame with results from initial experiments
# Replace this with your actual DataFrame
data = {
    'Learning_Rate': [0.001, 0.01, 0.001, 0.01],
    'Batch_Size': [32, 32, 128, 128],
    'Dropout_Rate': [0.2, 0.5, 0.2, 0.5],
    'Accuracy': [0.85, 0.88, 0.86, 0.89]  # Replace with actual accuracy values
}
response_surface_df = pd.DataFrame(data)

# Impute missing values if any (replace with your actual data preprocessing)
imputer = SimpleImputer(strategy='mean')
X = imputer.fit_transform(response_surface_df[['Learning_Rate', 'Batch_Size',
'Dropout_Rate']])

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Generate polynomial features
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(X_scaled)
```

```
# Instantiate and fit a Gaussian Process model
kernel = Matern(nu=2.5)
gpr_model = GaussianProcessRegressor(kernel=kernel, alpha=1e-4)
gpr_model.fit(X_poly, response_surface_df['Accuracy'])

# Define the expected improvement function
def expected_improvement(X, gpr_model, xi=0.01):
    mu, sigma = gpr_model.predict(X, return_std=True)
    mu_sample_opt = np.max(gpr_model.y_train_)

    with np.errstate(divide='ignore'):
        imp = mu - mu_sample_opt - xi
        Z = imp / sigma
        ei = imp * norm.cdf(Z) + sigma * norm.pdf(Z)
        ei[sigma == 0.0] = 0.0

    return ei

learning_rates = np.linspace(0.001, 0.01, 10)
batch_sizes = np.linspace(32, 128, 10)
dropout_rates = np.linspace(0.2, 0.5, 10)
new_points = np.array(np.meshgrid(learning_rates, batch_sizes,
dropout_rates)).T.reshape(-1, 3)

# Scale and transform the new points
new_points_scaled = scaler.transform(new_points)
new_points_poly = poly.transform(new_points_scaled)

# Calculate EI for each new point
ei_values = expected_improvement(new_points_poly, gpr_model)

# Select the point with the highest EI
max_ei_index = np.argmax(ei_values)
selected_point = new_points[max_ei_index]

# Selected hyperparameters for the next experiment
next_lr, next_batch_size, next_dropout_rate = selected_point
print(f"Next hyperparameters to try: Learning Rate = {next_lr}, Batch Size =
{next_batch_size}, Dropout Rate = {next_dropout_rate}")

def train_and_evaluate(lr, batch_size, dropout_rate):
    # Load CIFAR-10 data
    (x_train, y_train), (x_test, y_test) = cifar10.load_data()

    # Preprocess the data
    x_train, x_test = x_train / 255.0, x_test / 255.0
    y_train, y_test = to_categorical(y_train, 10), to_categorical(y_test, 10)
    # Split data for training and validation
    x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.1,
random_state=42)

    # Define the model architecture
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
        Dropout(dropout_rate),
        Flatten(),
        Dense(10, activation='softmax')
    ])

    # Compile the model
    model.compile(optimizer=Adam(learning_rate=lr), loss=CategoricalCrossentropy(),
metrics=['accuracy'])

    # Fit the model
    model.fit(x_train, y_train, batch_size=batch_size, epochs=10, verbose=0)

    # Evaluate the model
    _, accuracy = model.evaluate(x_val, y_val, verbose=0)

    return accuracy

# Preprocessing steps
imputer = SimpleImputer(strategy='mean')
X = imputer.fit_transform(response_surface_df[['Learning_Rate', 'Batch_Size',
'Dropout_Rate']])
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(X_scaled)

# Instantiate and fit a Gaussian Process model
kernel = Matern(nu=2.5)
gpr_model = GaussianProcessRegressor(kernel=kernel, alpha=1e-4)
gpr_model.fit(X_poly, response_surface_df['Accuracy'])

# Define the expected improvement function
def expected_improvement(X, gpr_model, xi=0.01):
    mu, sigma = gpr_model.predict(X, return_std=True)
    mu_sample_opt = np.max(gpr_model.y_train_)

    with np.errstate(divide='ignore'):
        imp = mu - mu_sample_opt - xi
        Z = imp / sigma
        ei = imp * norm.cdf(Z) + sigma * norm.pdf(Z)
        ei[sigma == 0.0] = 0.0

    return ei

# Number of iterations for the active learning loop
n_iterations = 15

for iteration in range(n_iterations):
    # Generate new candidate points
    learning_rates = np.linspace(0.001, 0.01, 10)
    batch_sizes = np.linspace(32, 128, 10, dtype=int)  # Ensure batch sizes are 
integers
    dropout_rates = np.linspace(0.2, 0.5, 10)
    new_points = np.array(np.meshgrid(learning_rates, batch_sizes, 
dropout_rates)).T.reshape(-1, 3)

    # Scale and transform the new points
    new_points_scaled = scaler.transform(new_points)
    new_points_poly = poly.transform(new_points_scaled)

# Function to train and evaluate the model
def train_and_evaluate(lr, batch_size, dropout_rate):
    # Load CIFAR-10 data
    (x_train, y_train), (x_test, y_test) = cifar10.load_data()

    # Preprocess the data
    x_train, x_test = x_train / 255.0, x_test / 255.0
    y_train, y_test = to_categorical(y_train, 10), to_categorical(y_test, 10)

    # Split data for training and validation
    x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.1, 
random_state=42)

    # Define the model architecture
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
        Dropout(dropout_rate),
        Flatten(),
        Dense(10, activation='softmax')
    ])

    # Compile the model
    model.compile(optimizer=Adam(learning_rate=lr), loss=CategoricalCrossentropy(), 
metrics=['accuracy'])

    # Fit the model
    model.fit(x_train, y_train, batch_size=int(batch_size), epochs=10, verbose=0)
```

```python
    # Evaluate the model
    _, accuracy = model.evaluate(x_val, y_val, verbose=0)

    return accuracy

response_surface_df = pd.DataFrame(data)

# Preprocessing steps
imputer = SimpleImputer(strategy='mean')
X = imputer.fit_transform(response_surface_df[['Learning_Rate', 'Batch_Size',
'Dropout_Rate']])
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(X_scaled)

# Instantiate and fit a Gaussian Process model
kernel = Matern(nu=2.5)
gpr_model = GaussianProcessRegressor(kernel=kernel, alpha=1e-4)
gpr_model.fit(X_poly, response_surface_df['Accuracy'])

# Define the expected improvement function
def expected_improvement(X, gpr_model, xi=0.01):
    mu, sigma = gpr_model.predict(X, return_std=True)
    mu_sample_opt = np.max(gpr_model.y_train_)

    with np.errstate(divide='ignore'):
        imp = mu - mu_sample_opt - xi
        Z = imp / sigma
        ei = imp * norm.cdf(Z) + sigma * norm.pdf(Z)
        ei[sigma == 0.0] = 0.0

    return ei

# Number of iterations for the active learning loop
n_iterations = 5

for iteration in range(n_iterations):
    print(f"\nIteration {iteration + 1}/{n_iterations}")

    # Generate new candidate points
    learning_rates = np.linspace(0.001, 0.01, 10)
    batch_sizes = np.linspace(32, 128, 10, dtype=int)
    dropout_rates = np.linspace(0.2, 0.5, 10)
    new_points = np.array(np.meshgrid(learning_rates, batch_sizes,
dropout_rates)).T.reshape(-1, 3)

    # Scale and transform the new points
    new_points_scaled = scaler.transform(new_points)
    new_points_poly = poly.transform(new_points_scaled)

    # Calculate EI for each new point
    ei_values = expected_improvement(new_points_poly, gpr_model)

    # Select the point with the highest EI
    max_ei_index = np.argmax(ei_values)
    selected_point = new_points[max_ei_index]
    selected_lr, selected_batch_size, selected_dropout_rate = selected_point

    print(f"Selected Hyperparameters: Learning Rate = {selected_lr}, Batch Size =
{selected_batch_size}, Dropout Rate = {selected_dropout_rate}")

    # Run the experiment with the selected hyperparameters
    experiment_accuracy = train_and_evaluate(selected_lr, selected_batch_size,
selected_dropout_rate)
    print(f"Experiment Accuracy: {experiment_accuracy}")

    # Update your dataset with the new experiment results
    new_data = np.array([[selected_lr, selected_batch_size, selected_dropout_rate]])
    new_data_scaled = scaler.transform(new_data)
    new_data_poly = poly.transform(new_data_scaled)

    # Extend the feature matrix X_poly and the target array Y
    X_poly = np.vstack([X_poly, new_data_poly])
    response_surface_df = response_surface_df.append({'Learning_Rate': selected_lr,
'Batch_Size': selected_batch_size, 'Dropout_Rate': selected_dropout_rate, 'Accuracy':
experiment_accuracy}, ignore_index=True)
    Y = response_surface_df['Accuracy'].values

    # Refit the Gaussian Process Regressor with the updated dataset
    gpr_model.fit(X_poly, Y)

print("Active learning iterations completed.")

# Number of iterations for the active learning loop
n_iterations = 5

for iteration in range(n_iterations):
    print(f"\nIteration {iteration + 1}/{n_iterations}")

    # Generate new candidate points (possibly using an adaptive strategy)
    learning_rates = np.linspace(learning_rate_range[0], learning_rate_range[1], 10)
    batch_sizes = np.linspace(batch_size_range[0], batch_size_range[1], 10, dtype=int)
    dropout_rates = np.linspace(dropout_rate_range[0], dropout_rate_range[1], 10)
    new_points = np.array(np.meshgrid(learning_rates, batch_sizes,
dropout_rates)).T.reshape(-1, 3)
```

```
    # Scale and transform the new points
    new_points_scaled = scaler.transform(new_points)
    new_points_poly = poly.transform(new_points_scaled)

    # Calculate EI for each new point using the updated GPR model
    ei_values = expected_improvement(new_points_poly, gpr_model)

    # Select the point with the highest EI
    max_ei_index = np.argmax(ei_values)
    selected_point = new_points[max_ei_index]
    selected_lr, selected_batch_size, selected_dropout_rate = selected_point

    print(f"Selected Hyperparameters: Learning Rate = {selected_lr}, Batch Size =
{selected_batch_size}, Dropout Rate = {selected_dropout_rate}")

    # Run the experiment with the selected hyperparameters
    experiment_accuracy = train_and_evaluate(selected_lr, selected_batch_size,
selected_dropout_rate)

    print(f"Experiment Accuracy: {experiment_accuracy}")

    # Update your dataset with the new experiment results
    new_row = {'Learning_Rate': selected_lr, 'Batch_Size': selected_batch_size,
'Dropout_Rate': selected_dropout_rate, 'Accuracy': experiment_accuracy}
    response_surface_df = response_surface_df.append(new_row, ignore_index=True)

    # Update the feature matrix and target array for the GPR model
    new_data_scaled = scaler.transform([[selected_lr, selected_batch_size,
selected_dropout_rate]])
    new_data_poly = poly.transform(new_data_scaled)
    X_poly = np.vstack([X_poly, new_data_poly])
    Y = response_surface_df['Accuracy'].values

    # Refit the Gaussian Process Regressor with the updated dataset
    gpr_model.fit(X_poly, Y)

print("Active learning iterations with updated querying completed.")

# Load CIFAR-10 data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train, y_test = tf.keras.utils.to_categorical(y_train, 10),
tf.keras.utils.to_categorical(y_test, 10)

# Split the original training data into a smaller training set and an unlabeled pool
x_train_labeled, x_pool, y_train_labeled, y_pool = train_test_split(x_train, y_train,
test_size=0.5, random_state=42)
```

```
# Function to create a CNN model with MC Dropout
def create_mc_dropout_cnn_model():
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
        MaxPooling2D((2, 2)),
        Dropout(0.25),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Dropout(0.25),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
    return model


# 'anova_df' is the DataFrame with your ANOVA results
anova_df['F Statistic'].plot(kind='bar')
plt.title('ANOVA Results for Hyperparameters')
plt.ylabel('F Statistic')
plt.xlabel('Hyperparameters')
plt.show()
best_hp = [-1, -1, 1, 1, 1, -1, 1]

# Load CIFAR-10 data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize the data
x_train = x_train / 255.0
x_test = x_test / 255.0

# Convert class vectors to binary class matrices
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Active Learning Parameters
n_queries = 20  # Number of queries per iteration
n_iterations = 15  # Total number of iterations

# Split the test dataset into a pool for active learning
x_pool, y_pool = x_full_test, y_full_test
x_pool = x_pool / 255.0
y_pool = to_categorical(y_pool, 10)
```

```
# Main Active Learning Loop
for iteration in range(n_iterations):
    # Define and compile the model
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=x_train.shape[1:]),
        BatchNormalization(),
        Dropout(0.2),
        Conv2D(64, (3, 3), activation='relu'),
        BatchNormalization(),
        Flatten(),
        Dense(64, activation='relu'),
        Dropout(0.5),
        Dense(10, activation='softmax')
    ])
    model.compile(loss=CategoricalCrossentropy(), optimizer='adam',
metrics=['accuracy'])

    # Train the model
    model.fit(x_train, y_train, batch_size=64, epochs=3, verbose=0)

    # Evaluate the model
    accuracy = model.evaluate(x_test, y_test, verbose=0)[1]
    print(f"Iteration {iteration+1} - Accuracy: {accuracy:.3f}")

    # If this is the last iteration, don't need to select new samples
    if iteration == n_iterations - 1:
        break

    # Select the next batch of samples for active learning
    indices = np.random.choice(range(len(x_pool)), n_queries, replace=False)
    x_selected = x_pool[indices]
    y_selected = y_pool[indices]

    # Add these points to the training data
    x_train = np.concatenate((x_train, x_selected))
    y_train = np.concatenate((y_train, y_selected))

    # Remove these points from the pool
    x_pool = np.delete(x_pool, indices, axis=0)
    y_pool = np.delete(y_pool, indices, axis=0)
```