Heuristic-driven Hyperparameter Optimization Algorithms

Abstract

Deep learning models' performance is heavily influenced by both their parameters and hyperparameters (Hyper_Params). While parameters are learned during the training process, Hyper_Params must be tuned manually or automatically through various hyperparameter optimization (HPO). Traditional HPO methods such as Grid Search, Random Search, and Bayesian Optimization (BO) are widely used but come with limitations. Grid Search is exhaustive and computationally expensive. Random Search is potentially quicker but can miss optimal configurations. BO is more efficient but still computationally demanding for large-scale problems. To address these inefficiencies, I propose a heuristic-based algorithm that combines the systematic nature of Grid Search with heuristics to allow for more exploration of areas of higher importance within a shorter computation time. The goal is to achieve near-optimal performance with significantly less computational effort. The accuracy and time performance of such algorithms were evaluated using image classification applications against Grid Search HPO techniques. The benchmarking was done with one real-world dataset.

Our source code is available at https://github.com/outsidermm/Heuristic-driven-Hyperparameter-Optimization-Algorithms

Keywords: Hyperparameter Optimisation, CIFAR-100, Convolutional Neural Network, Deep learning, Machine learning

1. Introduction

1.1 Hyperparameter Optimization Importance

Machine learning (ML) tasks such as image classification, regression, and clustering involve a set of hyperparameters (Hyper_Params) and parameters. While parameters are learned by optimizing loss functions in inference training, Hyper_Params control the training process. For example, the epoch used to train the process dictates the performance of the model on the given dataset but also the generalisability of the model, as more epochs can induce overfitting of the data while too few epochs can induce model underfitting. Hence, the performance of most modern ML algorithms depends crucially on the Hyper_Params, including their general model architecture, epoch, and regularisation parameters. Empirical studies have shown the significance of hyperparameter optimization (HPO) in transforming a model of average results to one of State-of-the-Art performance [4], [5].

Previous studies have shown how specific parameters contribute to the model performance and/or model behavior. This paper investigates heuristics for the Hyper_Parms of Epoch, Batch Size, and Learning Rate in the context of image classification tasks using Convolutional Neural Networks (CNNs) and a general real-world dataset. In addition, heuristic HPO algorithms are derived for respective Hyper_Parms and compared to an HPO technique called Grid Search in terms of time and accuracy performance. As part of this experiment, I benchmarked the Grid Search technique to perform HPO on a real-world dataset of CIFAR-100 [1]. A total of 3 Hyper_Param were tuned to select the most performant model. For evaluation, the accuracy and time consumption of the HPO process were taken.

The contribution of this work is to provide novice ML communities with heuristic HPO algorithms that can balance their computational requirements and model accuracies according to their preferences.

1.2. Hyper_Param Optimisation – A formal definition

Consider an ML Algorithm A in a model M whose goal is to find a function f that minimises the expected loss $L(x; f)$ over sample x drawn from a dataset D_{train}, as the dataset available for the model M is assumed to be $D = \{D_{train},\ D_{test}\}$. This is usually done by mapping D_{train} in the black box function f. While per iteration, the parameters θ within f are updated from inferential training, the algorithm A itself is parameterized by a set of Hyper_Params $x = \{h_1, h_2, \dots h_n\}$, where each $h_i \in x_i$. Let $\lambda = x_1 \times x_2 \times \dots \times x_n$ be the Hyper_Param space in which the algorithm A chooses its Hyper_Params. This results in a learning algorithm A_λ and expected loss function $L\big(x; A_\lambda(D_{train})\big)$. The goal of HPO is to find the best Hyper_Params which minimize L. It is written as:

$$\lambda^* = \underset{\lambda \in A}{\operatorname{argmin}}\, L\big(x; A_\lambda(D_{train})\big) \tag{1}$$

1.3. CNN

1.3.1. CNNs for Image Classification

In this paper, I focus on HPO in using CNNs to classify real-world images with one label. A CNN is a powerful class of neural network that has proven itself to be very capable of classifying images as it learns to associate the extracted features with the correct labels through a process of backpropagation and optimization [2], [3]. A common CNN architecture for image classification can be seen in Figure 1.

Once the CNNs have been trained with image datasets, they can predict unseen, new images by selecting the label with the highest predicted probability.

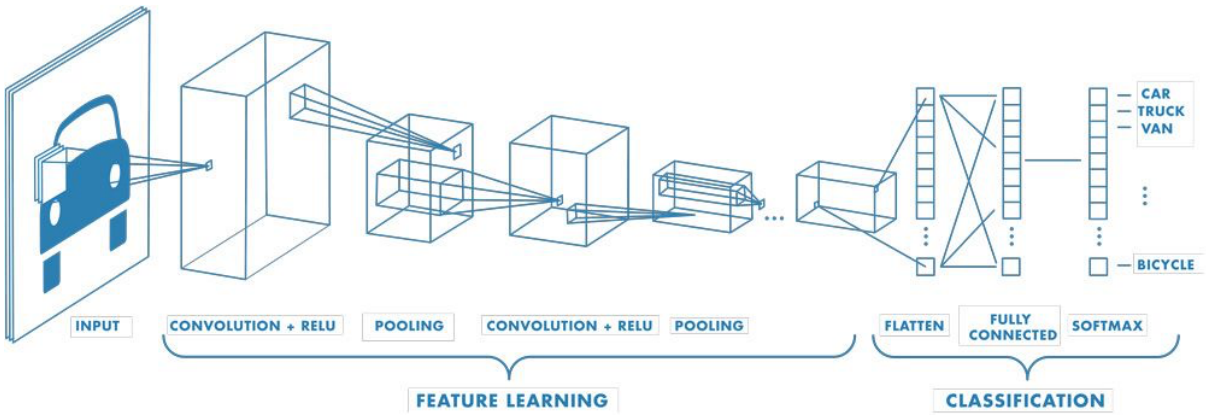

Figure 1: *Generic Convolutional Neural Network Model Architecture for Image Classification adapted from* [6]

1.3.2. Convolution Operation (Conv_Op)

A CNN's key success is based on the use of convolution operations. The Conv_Op is a mathematical operation that computes a signal that represents the influence of one signal on the other, weighted by the shape of the other signal. This is used to extract features in images in CNNs.

The mathematical definition of the Conv_Op [7] is:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]\, g[n-m] \quad (2)$$

The Conv_Op is a fundamental operation for CNNs that involves sliding a kernel or a filter over an input image and computing the dot product between the filter and the corresponding local region of the input image to produce an output feature map. An example of the Conv_Op can be seen in Figure 2.

The Conv_Op is applied to each channel of the input image separately, and the resulting feature maps are combined to form the output. Hence, the output would have the same depth as the color channel depths. In Conv_Op, various parameters are unique Hyper_Params of a CNN, including the size of the filter and the stride, which determine the distance between adjacent filter positions.

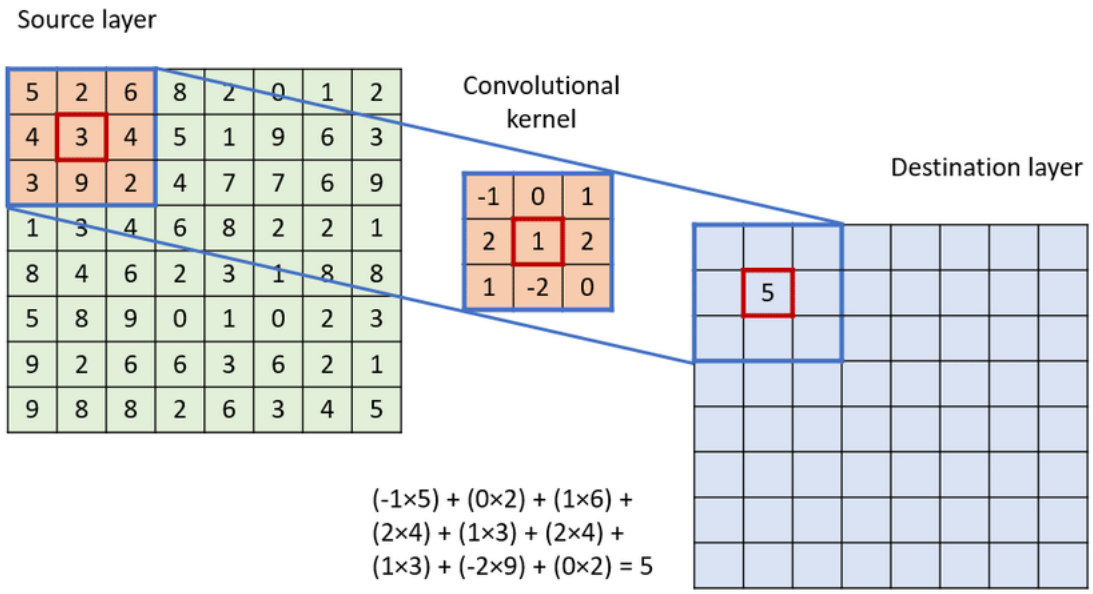

Figure 2: *Illustration of the Conv_Op as the convolutional kernel slides over the source layer, adapted from [8]*

1.3.3. Feature Maps

In CNNs, a feature map (Feat_Map) represents the output of a convolution layer, reflecting the degree to which a region of the image matches the kernels applied to it. Each element of the Feat_Map corresponds to the activation of a neuron in the layer and captures specific features of the input image according to the Conv_Op.

A Feat_Map is computed in each neuron by convolving a set of learnable matrices or kernels with the image. The resulting values from the Conv_Op are then summed and passed through an activation function to obtain the final values of the feature map [7].

As the Conv_Op is repeated for each filter in the layer, a set of Feat_Maps is computed, capturing different extracted features of the input image when the Feat_Maps from different colour channels are stacked together.

1.3.4. Activation function (Activ_Func)

Activ_Funcs are applied to every Feat_Map to introduce non-linearity to CNNs, as real-world phenomena exhibit complex, non-linear behaviors that a linear model can't accurately represent.

CNNs use various Activ_Funcs, including the rectified linear unit (ReLU), the sigmoid function, and the hyperbolic tangent function (tanh).

A. ReLU Activ_Func

Mathematically, the ReLU Activ_Func is defined as:

$$f(x) = \max(0, x) \tag{3}$$

where x is the input to the neuron [9]. This non-linear function suppresses the negative values, and therefore, its simplistic design allows it to eliminate the vanishing gradient problem in the backpropagation of deep neural networks. Due to its superior accuracy and time performance against other Activ_Funcs [10], the ReLU function is often used to generate a Feat_Map from the convolution layers' Conv_Ops.

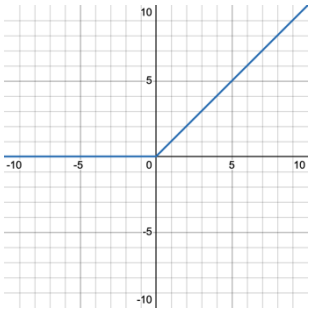

Figure 3: *Graph of the ReLU Activ_Func*

B. Sigmoid Activ_Func

The sigmoid function, on the other hand, is defined as

$$f(x) = \frac{1}{1+e^{-x}} \tag{4}$$

where x is the input to the neutron. The sigmoid function has shown itself as a good classification function as it has a characteristic S-shape seen in Fig. 4 and maps any real number to a value between 0 and 1. However, it suffers from the vanishing gradient problem when used in deep neural networks due to its saturating nature, slowing down the training time.

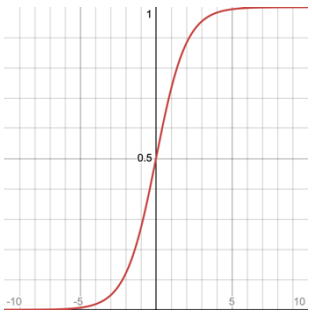

Figure 4: *Graph of the Sigmoid Activ_Func*

C. Hyperbolic Tangent (tanh) Activ_Func

The tanh function is similar to the sigmoid function as both experiences the vanishing gradient issue, seen in Fig. 5. It is mathematically written as:

$$f(x) = \tanh(x) \tag{5}$$

tanh maps any real number to a real value between -1 and 1. It is also a non-linear function that helps introduce complexity into the model.

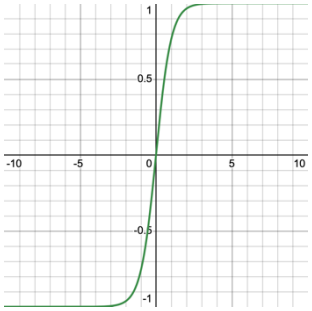

Figure 5: *Graph of the tanh Activ_Func*

D. Softmax Activ_Func

For a probabilistic multi-class classification problem, the CNN's output layer (Out_Lay) maps the output of the last layer to a probability distribution over the classes using a Softmax function. It is mathematically written as:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{6}$$

where σ is softmax. $\vec{z}$ is input vector. K is number of classes in the multi-class classifier. e^{z_i} is standard exponential function for input vector. e^{z_j} is standard exponential function for output vector.

1.3.5. Pooling layer

Pooling layers (Pool_Lays) are often used preceding the convolution layers to reduce the spatial dimension of Feat_Maps produced by taking the maximum or average value of distinct regions [7]. For the purpose of this paper, the max pooling operation was used. In maximum pooling, the layer applies a kernel that returns the maximum value observed within the distinct regions to the feature map. As seen in Figure 3, the maximum pooling layer only takes the maximum value of distinct regions, reducing the spatial dimension of the final outputted feature map.

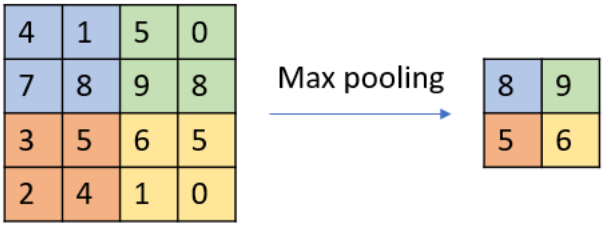

Figure 6: *Illustration of Max Pooling Layer Operation, Adapted from [8]*

It is also observed that this layer would have related model architectural Hyper_Params such as the filter size and the stride - the steps the filter will take to move through the image data.

The important image features are extracted using the convolution layers, Activ_Funcs, and Pool_Lays, which are further flattened into a one-dimensional vector. This is input to the section of the

CNN where classification occurs with fully connected layers, empowering CNNs to achieve image classification functionalities [11].

2. Related Works

2.1. Existing HPO Techniques

Traditionally, HPO has been done through personal experience and intuition, often resulting in sub-optimal model performances. Recent studies indicate that more sophisticated and automated approaches can be used to find better Hyper_Params, resulting in a better model [12], [13], [14].

2.1.1. Grid Search

In grid search, a systemic set of ranges is chosen to tune the Hyper_Param. A grid is constructed within this Hyper_Param space, testing all possible combinations of Hyper_Param values to find the best model performance with a set of Hyper_Params. While grid searches can be efficiently run when given a Hyper_Param space of low dimensionality, such as in 1-dimension or 2-dimensions, it becomes computationally expensive and unviable as time complexity grows exponentially in a Hyper_Param space of increasing dimensionality [15]. Using a sufficiently granular grid to perform the grid search, the solution can be sub-optimal when many areas of low importance are covered, and areas of high importance are overlooked since while a grid of points gives even coverage to the n-dimensional Hyper_Param space, individual Hyper_Param subspaces are insufficiently covered.

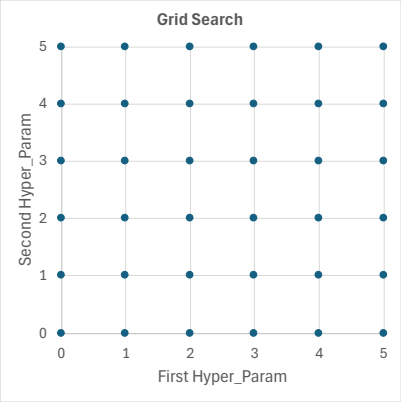

Figure 7: *Illustration of a 2-dimensional Grid Search Space*

2.1.2. Random Search

Random Search is another non-adaptive way to optimize Hyper_Param. Each Hyper_Param of interest is chosen randomly from a predefined distribution, forming random configurations of Hyper_Params to test for the best model performance. This is largely different from the exhaustive enumeration of grids used in a Grid Search.

While Grid Search suffers from the lack of exploration in individual subspaces of high dimensional Hyper_Param spaces, Random Search trades a better exploration of individual subspaces for a slightly less exploration of the overall Hyper_Param space. Studies have proven that Random Search is better than Grid Search in exploring a high dimensional search space with low intrinsic dimensionality, as a

less evenly distributed Hyper_Param space can allow for a better exploration of subspaces that influence the model performance the most [16], [17].

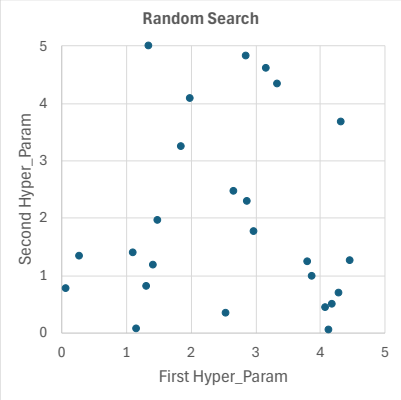

Figure 8: *Illustration of a 2-dimensional Random Search*

2.1.3. Bayesian Optimisation and More

Various other more complicated algorithms exist, including Bayesian Optimisation (BO) and population-based training algorithms. As seen in Fig. 7, BO methods and their variants balance exploitation and exploration using a probabilistic surrogate model for the object function that measures uncertainty [18]. While BO is efficient because it selects the next Hyper_Param in a guided manner, it is hard to benefit from the ease of parallelism used in non-adaptive HPO algorithms. Furthermore, BO has a greater consumption of computational resources compared with Grid Search and Random Search, which is of profound concern regarding complex optimization problems seen in the DL architectures [19]. Nevertheless, BO has been used to reach state-of-the-art model performance and is widely utilized for HPO to achieve excellent final model performances [14], [18], [20].

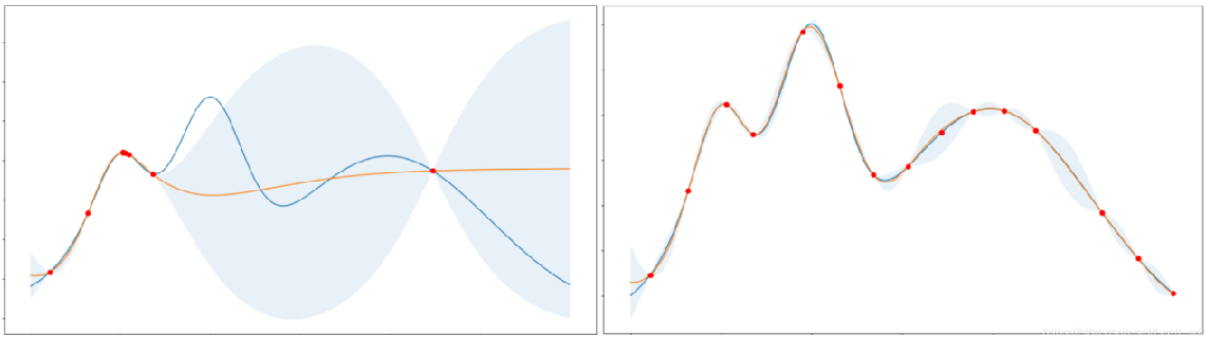

Figure 9: *Exploration-oriented (left) and exploitation-oriented Bayesian optimization (right); the shade indicates uncertainty. Source: [19]*

Various other algorithms, including ant colony optimization and genetic algorithms, have optimized different ML networks with varying performance. Regardless of their potential in tuning models of higher performance compared to simple adaptive algorithms like Grid Search and Random Search, they all require a deep understanding of mathematical foundations in order to comprehend and utilize them effectively.

2.2 Hyper_Param Analysis

2.2.1. Epoch

The number of epochs is a crucial Hyper_Param for tuning ML models. It is an integer value greater than 1 (inclusive). Epoch, the number of times the entire training dataset is passed through, affects the number of weight adjustments through backpropagation of error. Since epochs affect the total frequency of weight updates, they significantly affect model accuracy, as overfitting and underfitting of the data may occur.

While the epoch Hyper_Param does not have a generalizable solution due to the variation in different datasets [21], its impact on the model can have an analyzable trend, allowing for a heuristic trend to be used for optimization. Literature suggests that balancing the data's bias and variance is key to having the most performant model, as, at that point, the total error would be at a global minimum [22].

If the number of epochs is too low, the model might not have enough backpropagation to update the weights to understand the patterns in the data, leading to high bias (underfitting). On the other hand, increasing the number of epochs will increase the number of backpropagation, reducing the bias. However, training for too many epochs can lead to overfitting as memorization of the data occurs within the model, decreasing its generalization ability and increasing the variance [23]. Hence, according to Fig. 8, an optimal number of epochs can be found when bias and variance are minimized to the best of the model's ability, as indicated at the modeled minimum turn point of the total error curve.

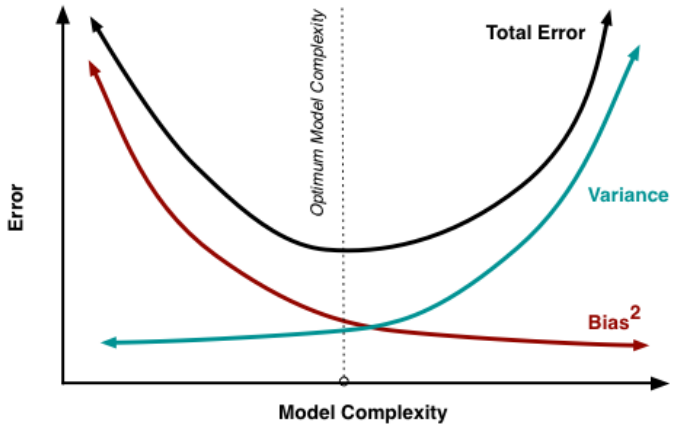

Figure 10: *Bias and Variance contributing to total error, adapted from [22]*

2.2.2. Batch Size

Batch size is the number of samples (positive integer value) fed into the model at each iteration of the training process. Hence, with a constant-sized dataset, a larger batch size will mean fewer updates with more samples per update. Hence, a higher batch size generically is attractive in reducing the time taken to train a model as the number of total iterations reduces. Nevertheless, it is essential to note that more data samples used in each iteration will increase memory usage.

2.2.3. Learning Rate

Learning rate is a Hyper_Param that would affect the speed of gradient descent, affecting the time the model takes to reach convergence. Its implication in determining how much of the gradient gets accounted for in the gradient descent can be seen in the Stochastic Gradient Descent [SGD] formula:

$$\omega_{t+1} = \omega_t - \gamma \nabla_{w_i} J(x^t, y^t; \omega_t) \tag{7}$$

here, y is the target variable, x denotes a single observation, ω_t is the current parameter value at t time step, ω_{t+1} is the updated parameter value, γ is the learning rate and J is the cost function.

While the learning rate Hyper_Param is intrinsic to converging to the global minimum of error, SGD is not always straightforward in complex optimization functions as parameters become trapped at local minimums with flat gradients. A typical sampling of the learning rate would be on a log scale, as the learning rate has multiplicative effects on the training dynamics. Hence, considering a range of learning rates multiplied or divided by some value is much more natural [24].

2.2.4. Momentum

The momentum Hyper_Param was introduced to combat the issue of pure SGD. SGD with momentum provides the opportunity for the parameter values to escape local minima as the momentum accumulation term is added:

$$v_t = \beta v_{t-1} - \gamma \nabla_{w_i} J(x^t, y^t; \omega_t) \tag{8}$$

$$\omega_{t+1} = \omega_t + v_t \tag{9}$$

here, y represents the target variable, while x denotes a single observation. The parameter value at the current time step t is represented as ω_t, and the updated parameter value at the next time step is ω_{t+1}. The momentum Hyper_Param is denoted by β, the learning rate is denoted by γ, and the cost function is represented as J.

Therefore, momentum can improve SGD performance in global optimization problems, enabling greater model generalization performance. Yet, some studies have shown a lesser effect of momentum in providing generalization performance and training acceleration when the learning rate is low in SGD with momentum [25] [26]. Comparatively, the learning rate Hyper_Param is more crucial than momentum due to its wide applicability in affecting all parameter updates. In contrast, momentum's major usage provides a higher generalization ability of complex models with large learning rates.

3. Methodology

3.1. Experimental Procedure

This paper attempts to solve the insufficient coverage of Hyper_Param subspaces in normal grid search by proposing heuristic HPO algorithms. To investigate the heuristics behind Hyper_Params in correlation to computation time and accuracy, I initially established the baseline model performance using Grid Searches. Traversing through the optimized Hyper_Param search space seen in Table 1 one by one, the training time taken to Hyper_Param search one entire set of 1-dimensional Hyper_Param and maximum accuracy yielded from the Grid Search were recorded to establish the baseline of comparison.

During the baseline establishment, we also analyzed how increasing the values of each Hyper_Param affected the training time and model accuracy. Based on the insights gained from the trend analysis and the meaning behind each Hyper_Param, we developed heuristic-driven HPO algorithms that aim to navigate the Hyper_Param subspaces with adaptive abilities efficiently.

The heuristic HPO algorithms were then compared against the Grid Search in their respective Hyper_Param subspaces. Comparison metrics include total time cost, optimal Hyper_Param found, accuracies and model training time associated with the optimal Hyper_Param found, and total number of search iterations.

Table 1: *Overview of fixed and optimized Hyper_Params during the experiments.*

	Hyper_Params	**Values**
Fixed	Activation Function	Rectified Linear Unit
	Pooling Operation	Max Pooling
	Max Pool Size	(2, 2)
	Convolution Kernal Size	(3, 3)
	Optimizer	Stochastic Gradient Descent
	Momentum	0.9
Optimized	Batch Size	$\{2^{4+2x} \mid x \in \mathbb{N}_0, 0 \leq x \leq 5\}$
	Learning Rate	$\{10^{-x} \mid x \in \mathbb{N}, 1 \leq x \leq 7\}$
	Epoch	$\{10 \cdot x \mid x \in \mathbb{N}, 1 \leq x \leq 15\}$

3.2 Data Preparation

In this paper, I used a publicly available dataset, CIFAR-100. CIFAR-100 consists of 100 classes of labels of real-world objects. Images were zero-centred to a range of [0,1] and augmented to decrease model overfitting. The details of the dataset and the dataset splits are described in Table 2, and the data augmentation performed is described in Table 3.

Table 2: *Information about datasets used in the experiments*

Dataset	**Image Resolution**	**Number of Classes**	**Data Split (train/validation/test)**
CIFAR-100	32 x 32	100	60,000/20,000/10,000

Table 3: *Parameters for Data Augmentation used in this paper*

Augmentation Type	***Parameter***	***Description***	***Value***
Rotation	Mode	Direction of flip	HORIZONTAL
	Seed	Seed for random number generator	42
Flip	Mode	Maximum rotation angle as a factor of τ	0.2
	Seed	Seed for random number generator	42
Translation	Height Factor	Maximum height shift as a fraction of image height	0.1
	Width Factor	Maximum width shift as a fraction of image width	0.1
	Seed	Seed for random number generator	42

3.3. Architecture Overview

The experiments were run on a fixed model architecture based on the well-known Visual Geometry Group of the University of Oxford 16 (VGG16) architecture using TensorFlow [27] and Keras [28]. VGG16 is a CNN architecture developed by Karen Simonyan and Andrew Zisserman in 2014 [29] and has demonstrated state-of-the-art performance on the ImageNet dataset [30], [31], [32], [33]. The VGG16 architecture uses multiple layers of small 3×3 convolution filters, followed by max Pool_Lays. To further address the problem of overfitting, the ReLU Activ_Func, batch normalization (batch_norm), and dropout regularizations were used preceding each convolutional layer, which are extra modifications to the generic VGG16 architecture. Furthermore, the number of neurons in the fully connected classification layers was reduced to reduce the time needed to train this network. In total, 15,885,320 parameters are in this modified VGG16 network.

Table 4: *Modified VGG16 network. The convolutional layer parameters are denoted as "conv(receptive field size)-(number of channels)." The ReLU Activ_Func and batch_norm layer preceding each convolutional layer is not shown for brevity.*

Modified VGGNet-16 Configuration
input (224×224 RGB Image)
conv3-64 dropout conv3-64
maxpool
conv3-128 dropout conv3-128
maxpool
conv3-256 dropout conv3-256 dropout conv3-256
maxpool
conv3-512 dropout conv3-512 dropout conv3-512
maxpool
conv3-512 dropout conv3-512 dropout conv3-512
maxpool
dropout
flatten
FC-512
ReLU
batch_norm

dropout
Out_Lay
soft-max

3.4. Device Overview

All experiments in this paper were done on a server with a dual 12-core Intel Xeon E5-2650 v5 processor setup, yielding 24 physical cores. The AI training was done in parallelism with 4 NVIDIA GV100GL (Tesla V100 PCIe 16GB) with 125.76 GiB of Randomly Access Memory. Each GPU's DL performance is 112 teraFLOPS with 640 Tensor Cores. During training, a mirrored strategy was used to create a replica of the model on each GPU, and the variables were mirrored across all replicas. During training, each GPU trained on allocated subsets of the data, and the learned parameters were merged using "all-reduce." Therefore, it is essential to note that any mention of batch size will mean the local batch size allocated per GPU.

3.5. Model Overhead Metrics

One unique metric used in this paper is model overhead. This metric combines model training time and model accuracy with their respective weights, expressed mathematically as:

$$Model\ Overhead = W_{time} \times t + W_{error} \times (1 - acc) \quad (10)$$

where W_{time} is a manually adjustable weight that determines the contribution of training time to the overhead, t is the normalized value of training time, W_{error} is a manually configurable weight that determines the contribution of model error to the overhead, and acc is the normalized value of model accuracy.

In this paper, both the error rate and the training time were given equal weight when calculating the model overhead. However, the trade-off between training time and model accuracy is a subjective area of concern. Hence, the weights are configurable, allowing users to adjust according to their preferences.

3.6. Heuristic HPO Algorithms

3.6.1. Epoch Heuristic HPO Algorithm

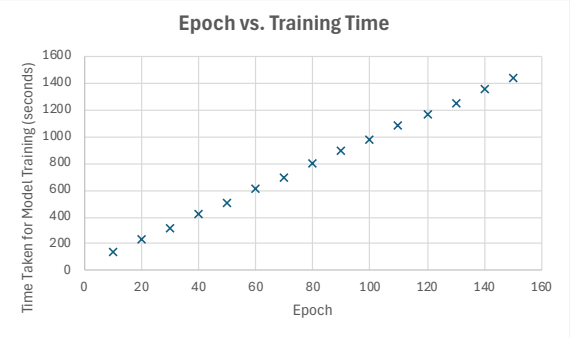

Figure 11: *Relationship Between Epoch and Training Time*

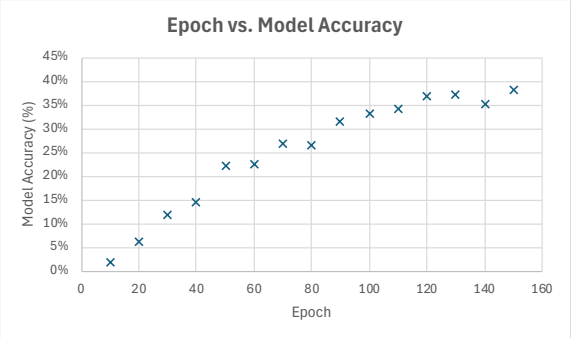

Figure 12: *Relationship Between Epoch and Model Accuracy*

To find the heuristics relevant to the heuristic epoch HPO algorithm, a grid search was performed by varying the epoch as described in Table 1 while keeping batch size at a constant 128, the learning rate at 0.01, and the momentum at a constant 0.9.

As seen in Fig. 11, increasing the epoch linearly increases the model training time. The trend between epochs and training time can be modeled with the least square regression line of $y = 9.3418x + 42.446$. This linear trend has a high Pearson's correlation coefficient of 0.9998. On the other hand, illustrated in Fig. 12, as the epoch Hyper_Param increases, the rate of increase decreases, describing the reaching of the global minima of total error depicted in Fig. 10.

Evidently, increasing epoch linearly will lead to a linear increase in model training time but a logarithmic increase in accuracy. Hence, assuming that training time and model accuracy are normalized to the same scale, the model overhead would decrease, reach a global minimum, and then increase, allowing a binary-search-inspired algorithm to be developed to find the optimal epoch where model overhead is at a minimum.

Algorithm 1 Heuristic Epoch HPO Algorithm

Input:
  $\mathcal{D}$: The dataset to be used.
  L: Left bound for the batch size search.
  R: Right bound for the batch size search.
  ϵ: Minimum range between the left and right bounds of the search space at which the algorithm will conclude the search.

Output:
  best_epoch: Best epoch.
  best_acc: Accuracy at the best epoch.
  best_time: Training time at the best epoch.

Initialization:
  $left \leftarrow L$
  $right \leftarrow R$
  $best\_epoch \leftarrow L$
  $best\_overhead \leftarrow \infty$
  Load $\mathcal{D}$ into *train_ds*, *val_ds*, and *test_ds*

Algorithm:
Train models with epochs *left* and *right*
Calculate overheads: *left_overhead*, *right_overhead*
while $left \leq right$ **do**
  if $right - left \leq E$ **then**
    if $mid\_overhead < best\_overhead$ **or**
      $right\_overhead < best\_overhead$ **or**
      $left\_overhead < best\_overhead$ **then**
      Update *best_epoch* and *best_overhead*
    end if
    break
  end if
  $mid \leftarrow \left\lfloor \frac{left+right}{2} \right\rfloor$
  Train model with epoch $mid, left, right$
  Calculate *mid_overhead*, *left_overhead*, *right_overhead*
  if $left\_overhead < mid\_overhead$ **then**
    $right \leftarrow mid$
  else if $right\_overhead < mid\_overhead$ **then**
    $left \leftarrow mid$
  else
    $right \leftarrow mid$
  end if
end while
return (*best_epoch*, *best_acc*, *best_time*)

Figure 13: *The pseudocode of the heuristic epoch HPO algorithm*

This algorithm is adaptive in providing necessary exploration according to its area of importance. To calculate the model overhead on lines 21 and 33, the accuracy and time taken recorded from lines 20 and 32 have to be Min-Max normalized between a range of [0, 1], mathematically written as:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{11}$$

where x' is the normalized value, x is the original value (either from the list of accuracies or from the list of training times).

Min-max normalization is necessary to ensure that the accuracy and training time data are both on a consistent scale, where their numerical values will not affect the fairness of the model overhead calculation.

In scenarios where the model overheads found on both the left and right boundaries are larger than the model overhead found at the middle marker, the algorithm heuristically explores the set of epochs originally situated between $[left, mid]$ by setting $right \leftarrow mid$, as seen on lines 39 and 40. This approach is motivated by the results shown in Fig. 11 and Fig. 12, where the minimum model overhead is more likely to be associated with an epoch of smaller value.

Recognizing the flexibility that ML practitioners should have while using this heuristic epoch HPO algorithm, the algorithm has an optional argument of $Exploration\ Factor$, denoted by ε. The $Exploration\ Factor$ is the minimum range between the left and right bounds of the search space at which the algorithm will conclude the search. Hence, the parameter ε controls the search granularity of the algorithm. A smaller ε leads to a finer search granularity, providing more precise results, while a larger ε results in a coarser search that concludes more quickly with less precision, derived from a greater range of data.

3.6.2. Batch Size Heuristic HPO Algorithm

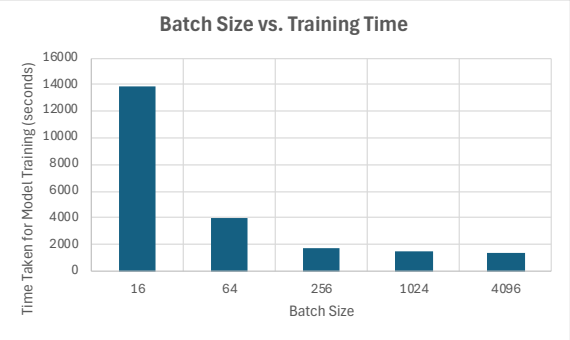

Figure 14: *Relationship Between Batch Size and Training Time*

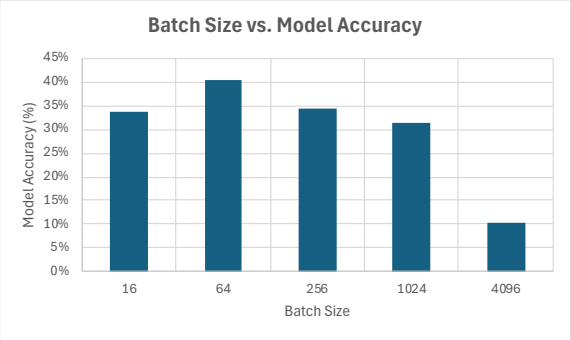

Figure 15: *Relationship Between Batch Size and Model Accuracy*

To identify the heuristics relevant to the heuristic epoch HPO algorithm, a grid search was performed by varying the batch size as described in Table 1 while keeping the epoch at a constant 250, the learning rate at 0.01, and the momentum constant at 0.9.

As seen in Fig. 14, as the batch size increases exponentially, more samples are used per backpropagation, resulting in fewer total backpropagations and decreased the model training time.

In Fig. 14 and Fig. 15, it is observed that prior to the irregular decrease in accuracy at a batch size of 2^{12}, the training time decreased significantly without a drastic decrease in model accuracy, with accuracy ranging between 31% and 40%. After the sharp decrease in accuracy to 10%, further increases in batch size led to Memory Allocation Errors (MAE) as the memory required for the batches exceeded the memory allocated for the training pipeline.

Motivated toward tuning sustainable models with reduced model training time, a heuristic batch size HPO algorithm was created, aiming to find the maximum batch size that can be used without encountering the significant performance drop described above.

Algorithm 2 Heuristic Batch Size HPO Algorithm

```
1:  Input:
2:      𝒟: The dataset to be used.
3:      L: Left bound for the batch size search.
4:      R: Right bound for the batch size search.
5:      ϵ: Acceptable range for accuracy deviation.
6:
7:  Output:
8:      best_batch: Best batch size.
9:      best_acc: Accuracy at the best batch size.
10:     best_time: Training time at the best batch size.
11:
12: Initialization:
13:     left ← L
14:     right ← R
15:     acceptable_range ← ϵ
16:     Load 𝒟 into train_ds, val_ds, and test_ds
17:     acc_bound ← accuracy with batch size 2^L
18:
19: Algorithm:
20: while left ≤ right do
21:     mid ← left + (right − left)//2
22:     try
23:         (time, acc) ← train model with batch size 2^mid
24:         Update batch_list, time_list, accuracy_list
25:         if |acc − acc_bound| < acceptable_range then
26:             acc_bound ← (acc + acc_bound)/2
27:             best_batch ← mid
28:             left ← mid + 1
29:         else
30:             right ← mid − 1
31:         end if
32:     catch (Memory Allocation Error)
33:         right ← mid − 1
34:     end try
35: end while
36: return (best_batch, best_acc, best_time)
```

Figure 16: *The pseudocode of the heuristic batch size HPO algorithm*

The heuristic batch size tuning algorithm aims to move the middle marker as right as possible before encountering expected performance drops and MAEs. The algorithm requires a parameter input of $accuracy\ tolerance$, denoted by ε. The acc_bound is initialized with the accuracy of the left-most bound, which, although taking a lot of time to train, serves as an average accuracy measured before any performance drop.

If the difference between the average accuracy measured and the accuracy measured at the middle point exceeds ε, it is considered an irregular decrease. In such cases, as seen on line 30, the algorithm

adjusts the search space by moving the right bound to the midpoint, effectively exploring the interval between the original left bound and the midpoint.

Furthermore, if an MAE is caught from the model training progress, the same adjustment to the search space will occur, as evident on line 33.

Conversely, if the difference between the average accuracy and the measured accuracy at the midpoint does not exceed ε, the search space is adjusted by moving the left marker to the midpoint, allowing the algorithm to explore intervals that entail larger batch sizes for time optimization. The average accuracy is also updated with the newly measured accuracy at the midpoint, as shown in lines 25-28.

The manual input of the value ε provides flexibility for ML practitioners using the algorithm to find their model's most optimal batch size. Users can adjust ε to control their tolerance for accuracy decreases as batch size increases and training time decreases.

When ε is set to a larger value, the algorithm becomes more tolerant of accuracy drops, allowing it to consider larger batch sizes as optimal. Oppositely, when ε is set to a smaller value, the algorithm becomes more sensitive to accuracy drops, considering smaller batch sizes to be significant for maintaining model performance.

The question of how much time decrease is worth the accuracy drop is subjective. Hence, this algorithm allows the user to define what constitutes an acceptable accuracy level for their needs, balancing it against the reduction in training time.

3.6.3. Learning Rate Heuristic HPO Algorithm

Figure 17: *Relationship Between Learning Rate and Training Time*

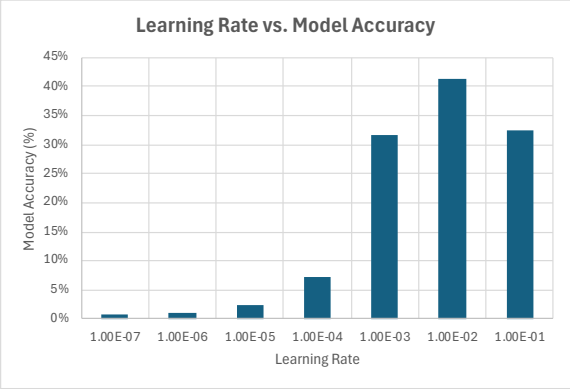

Figure 18: *Relationship Between Learning Rate and Model Accuracy*

As discussed in sections 3.2.3 and 3.2.4, the Hyper_Params of momentum and learning rate play important roles in the SGD used to update the neurons' weights. However, this paper only investigates the learning rate Hyper_Param because it has a higher impact on model training than momentum. Investigating only one variable used in SGD also mitigates the interplay between momentum and learning rate.

As seen in Fig. 17, differing learning rates impact training time minimally. Observed in Fig. 18, as the learning rate increases, the measured model accuracy portrays a bell curve trend with no additional local maxima. This motivates the implementation of a binary-search framework, aiming to find the maxima of the accuracy curve.

Algorithm 3 Heuristic Learning Rate HPO Algorithm

Input:
  Dataset $\mathcal{D}$: The dataset to be used.
  L: Left bound for the learning rate search.
  R: Right bound for the learning rate search.
  ϵ: Local extrema allowance.

Output:
  best_lr: Best learning rate.
  best_acc: Accuracy at the best learning rate.
  best_time: Training time at the best learning rate.

Initialization:
  $left \leftarrow L$
  $right \leftarrow R$
  Initialize lists for evaluated learning rates, accuracies, and times
  Load $\mathcal{D}$ into *train_ds*, *val_ds*, and *test_ds*

Algorithm:
while $left \leq right$ **do**
  $mid \leftarrow \left\lfloor \frac{left+right}{2} \right\rfloor$
  if *mid* not evaluated **then**
    Train model with learning rate 10^{-mid}
    Record *mid_acc* and *mid_time*
  end if
  if *left* not evaluated **then**
    Train model with learning rate 10^{-left}
    Record *left_acc* and *left_time*
  end if
  if *right* not evaluated **then**
    Train model with learning rate 10^{-right}
    Record *right_acc* and *right_time*
  end if
  if $(mid\_acc > left\_acc + \epsilon)$ and $(mid\_acc > right\_acc + \epsilon)$ **then**
    return (mid, mid_acc, mid_time)
  end if
  if $mid\_acc > left\_acc$ **then**
    $left \leftarrow mid + 1$
  else
    $right \leftarrow mid - 1$
  end if
end while
return no optimal learning rate found

Figure 19: *The pseudocode of the heuristic learning rate HPO algorithm*

By comparing the accuracy at the midpoint with those at the boundary points, the trend among these three points can be analyzed to identify areas with high potential for containing the peak. This approach allows the algorithm to explore regions more likely to contain the global maximum while avoiding areas where the trend does not suggest the presence of a global maximum.

While the CIFAR-100 accuracy results in Fig. 16 did not display a local maximum other than the global maximum, my algorithm incorporates a manual ε parameter. When the difference between accuracy measured in the midpoint and those measured at the left and right markers exceed ε, the midpoint is considered a peak. This mechanism ensures that minor local maxima do not obstruct the identification of the global maximum in multimodal accuracy trends.

4. Result

4.1. Epoch Model Performance

Table 5: *The Final epoch solution performance of Grid Search and Heuristic Tuning HPO Algorithm*

HPO Algorithm	**Optimal Epoch**	**Model Overhead**	**Model Accuracy (%)**	**Training Time (seconds)**
Grid Search	70	0.3693	**26.84**	691
Heuristic Tuning	40	**0.3608**	18.16	**419**

According to Table 5, the heuristic HPO algorithm has a marginally lower model overhead than Grid Search. This is due to the trade-off between a 39% relative decrease in training time and a 32% relative decrease in accuracy. This interplay between the model accuracy and training time further reinforces the need to evaluate the use-case suitability of the heuristic tuning algorithm. By adjusting the weights for training time and error rate in the model overhead calculation based on personal preference, users can personalize the algorithm to meet their specific needs better.

4.2 Batch Size Model Performance

Table 6: *The Final batch size solution performance of Grid Search and Heuristic Tuning HPO Algorithm*

HPO Algorithm	**Optimal Batch Size**	**Model Accuracy (%)**	**Training Time (seconds)**
Grid Search	1024	**31.0%**	1437
Heuristic Tuning	2048	20.8%	**1416**

As seen in Table 6, while the heuristic batch size performance did find a batch size with a lower but negligible training time, it faced a significant 17.98% accuracy drop relative to the accuracy found by the Grid Search. This can be explained by the generous value of the accuracy tolerance parameter. Therefore, while effectively reducing training time, the heuristic batch size algorithm may require adjustments of the accuracy tolerance parameter based on personal preferences.

4.3 Learning Rate Model Performance

Table 7: The *Final learning rate solution of Grid Search and Heuristic Tuning HPO Algorithm*

HPO Algorithm	**Optimal Learning Rate**
Grid Search	0.01
Heuristic Tuning	0.01

Commenting on the final performance data seen in Table 7, both HPO algorithms found the same optimal learning rate, indicating the proposed algorithm's effectiveness compared to the existing HPO technique of Grid Search.

4.4 Comparison of Time Cost and Iteration Count in HPO Algorithms

Table 8: *Comparison of Time Cost and Iteration Count in HPO Algorithms*

Hyper_Param	**Metric**	**Grid Search**	**Heuristic Tuning**
Epoch	Number of Iteration	15	**8**
	Total Time Cost (seconds)	11847	**4349**
Batch Size	Number of Iteration	**5**	**5**
	Total Time Cost (seconds)	22483	**18439**

Learning Rate	Number of Iteration	7	**5**
	Total Time Cost (seconds)	16782	**11790**

Per Table 8, all heuristic HPO algorithms proposed for all three parameters consistently showed a time complexity lower or equal to $O(n)$ where n is the number of training cycles a systemic Grid Search has to run. The three HPO algorithms that I designed based on a binary search framework, all have a time complexity of approximately $O(logn)$. This implies that the algorithms' efficiencies will increase as the search space increases. This paper did not investigate the increasing efficiency due to the lack of computation powers. The time cost to run the heuristic tuning algorithms was consistently lower than running a systemic Grid Search by 63%, 18%, and 30%, respectively, for the Hyper_Param of the epoch, batch size, and learning rate. The comparatively lower time cost of the heuristic HPO algorithm showcases its potential as a more sustainable alternative to the systemic Grid Search HPO technique.

4.5 Comparison of Distribution of Tested Values in Different Algorithms

I evaluated the adaptiveness of the algorithm in exploring areas of Hyper_Param subspaces of varying significance by plotting the values of Hyper_Params tested, as shown in Fig. 20, Fig. 21 and Fig. 22.

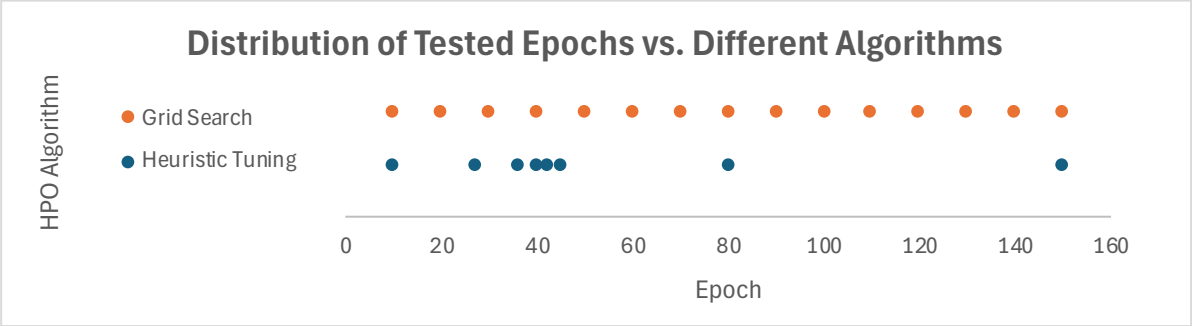

Figure 20: *Distribution of tested epochs compared to different algorithms*

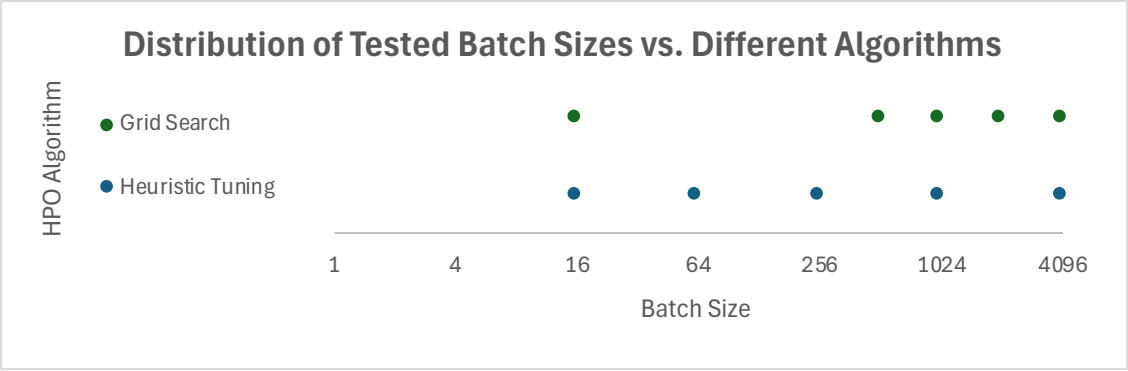

Figure 21: *Distribution of tested batch sizes compared to different algorithms*

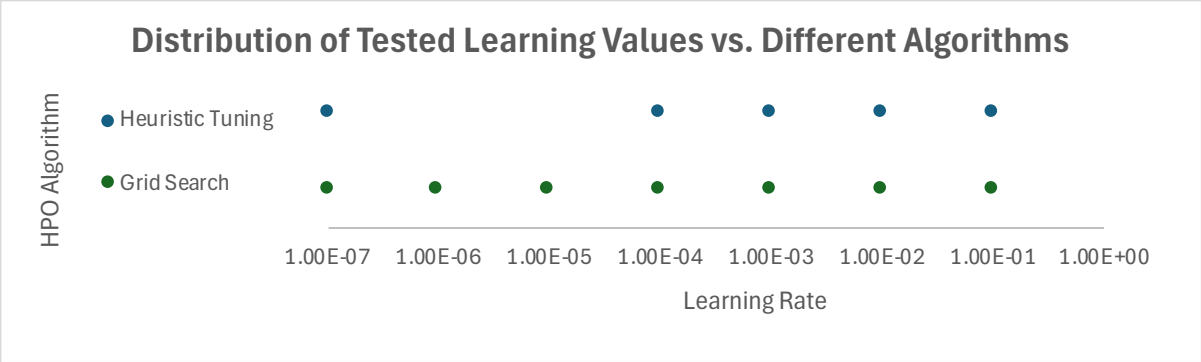

Figure 22: *Distribution of tested learning rates compared to different algorithms*

As observed in Fig. 20, Fig. 21, and Fig. 22, while Grid Search always had a uniform distribution of tested values based on its systemic nature, the heuristic HPO algorithms displayed the expected adaptiveness and non-uniformity distribution of tested points. The non-uniform distribution of the tested Hyper_Params highlights the heuristic algorithm's ability to concentrate on areas with higher potential for optimal solutions, avoiding unnecessary exploration of less significant areas. The adaptive strategies within the heuristic HPO algorithms allowed HPO to be done faster where computational resources are limited.

On the other hand, the uniform distribution in all three figures affirms the exhaustive and systemic nature of Grid Search. While it provides a thorough even search of the Hyper_Param subspaces, it is computationally expensive. It explores areas of less promising results with equal uniformity as more promising search spaces.

5. Future Works

This paper provides a promising evaluation for exploring heuristic-driven HPO algorithms, especially for tuning general Hyper_Params of importance. It also provides interesting insight into how binary search-driven adaptive algorithms can decrease the overall time consumption of the HPO process while providing more exploration to Hyper_Param subspaces of high importance and exploring less in areas of lesser importance. Given the well-founded heuristics demonstrated by the experiments and the strong mathematical basis behind the optimized Hyper_Params, these heuristic HPO algorithms present a more promising HPO solution than Grid Search. They enable novice ML learners to perform HPO that balances their computational requirements and model accuracies according to their preferences.

While the experiments were done with a single dataset and a constant model architecture, their robustness can be further evaluated with variations in conditions that include:

- Different sized datasets
- Different nature of datasets
- Different model architecture

Due to the limited computational resources, the heuristic HPO algorithm was not run on multi-dimensional Hyper_Param spaces and compared against HPO techniques such as Grid Search, Random Search, and BO.

Future works could explore how ML optimization techniques such as pruning, quantization, and parallelism methods affect the reliability and accuracy of these heuristic-driven tuning algorithms.

6. Conclusion

This work presents three respective heuristic-driven algorithms for HPO of Epoch, Batch Size, and Learning rate. I have compared these algorithms' performance against the existing Grid Search technique on CIFAR-100. A binary-search-inspired algorithm framework significantly impacted adaptively exploring Hyper_Param subspaces according to their estimated importance. It is also more efficient and less time-consuming. While these optimization algorithms undeniably yield worse results than industrial solutions like BO, they can be an entry-level optimization tool for ML communities with low technical knowledge, empowering them to perform HPO that balances their computational requirements and model accuracies according to personal preferences. Future studies should explore the robustness of these heuristic HPO algorithms under various conditions.

7. Acknowledgement

I would like to thank Professor Seda Memik from the University of Northwestern for providing mentoring and feedback for this paper. I would also like to thank Professor Manyi Wang and his research team from the Nanjing University of Information Science and Technology for providing technical support and feedback for this paper.

Reference

[1] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” pp. 32–33, Apr. 2009.

[2] L. Shen, Z. Lin, and Q. Huang, “Relay Backpropagation for Effective Learning of Deep Convolutional Neural Networks,” Apr. 03, 2016, *arXiv*: arXiv:1512.05830. doi: 10.48550/arXiv.1512.05830.

[3] E. Dogo, O. Afolabi, N. Nwulu, B. Twala, and C. Aigbavboa, *A Comparative Analysis of Gradient Descent-Based Optimization Algorithms on Convolutional Neural Networks*. 2018. doi: 10.1109/CTEMS.2018.8769211.

[4] G. Habib and S. Qureshi, “Optimization and acceleration of convolutional neural networks: A survey,” *J. King Saud Univ. - Comput. Inf. Sci.*, vol. 34, no. 7, pp. 4244–4268, Jul. 2022, doi: 10.1016/j.jksuci.2020.10.004.

[5] Y. Ren and X. Cheng, *Review of convolutional neural network optimization and training in image processing*. 2019. doi: 10.1117/12.2512087.

[6] “What Is a Convolutional Neural Network? | 3 things you need to know.” Accessed: Jul. 03, 2024. [Online]. Available: https://www.mathworks.com/discovery/convolutional-neural-network.html

[7] M. Krichen, “Convolutional Neural Networks: A Survey,” *Computers*, vol. 12, no. 8, Art. no. 8, Aug. 2023, doi: 10.3390/computers12080151.

[8] D. Podareanu, “Best Practice Guide - Deep Learning,” *Deep Learn.*.

[9] K. Fukushima, “Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements,” *IEEE Trans. Syst. Sci. Cybern.*, vol. 5, no. 4, pp. 322–333, Oct. 1969, doi: 10.1109/TSSC.1969.300225.

[10] T. Szandała, “Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks,” in *Bio-inspired Neurocomputing*, vol. 903, A. K. Bhoi, P. K. Mallick, C.-M. Liu, and V. E. Balas, Eds., in Studies in Computational Intelligence, vol. 903. , Singapore: Springer Singapore, 2021, pp. 203–224. doi: 10.1007/978-981-15-5495-7_11.

[11] N. Sharma, V. Jain, and A. Mishra, “An Analysis Of Convolutional Neural Networks For Image Classification,” *Procedia Comput. Sci.*, vol. 132, pp. 377–384, Jan. 2018, doi: 10.1016/j.procs.2018.05.198.

[12] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for hyper-parameter optimization,” in *Proceedings of the 24th International Conference on Neural Information Processing Systems*, in NIPS’11. Red Hook, NY, USA: Curran Associates Inc., 2011, pp. 2546–2554.

[13] N. Pinto, D. Doukhan, J. J. DiCarlo, and D. D. Cox, “A High-Throughput Screening Approach to Discovering Good Forms of Biologically Inspired Visual Representation,” *PLOS Comput. Biol.*, vol. 5, no. 11, p. e1000579, Nov. 2009, doi: 10.1371/journal.pcbi.1000579.

[14] J. Snoek, H. Larochelle, and R. P. Adams, “Practical Bayesian Optimization of Machine Learning Algorithms,” in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2012. Accessed: Jul. 03, 2024. [Online]. Available: https://papers.nips.cc/paper_files/paper/2012/hash/05311655a15b75fab86956663e1819cd-Abstract.html

[15] M. Claesen, J. Simm, D. Popovic, Y. Moreau, and B. De Moor, “Easy Hyperparameter Search Using Optunity,” Dec. 02, 2014, *arXiv*: arXiv:1412.1114. doi: https://doi.org/10.48550/arXiv.1412.1114.

[16] L. Yang and A. Shami, “On Hyperparameter Optimization of Machine Learning Algorithms: Theory and Practice,” *Neurocomputing*, vol. 415, pp. 295–316, Nov. 2020, doi: 10.1016/j.neucom.2020.07.061.

[17] J. Bergstra and Y. Bengio, “Random Search for Hyper-Parameter Optimization”.

[18] R. Turner *et al.*, “Bayesian Optimization is Superior to Random Search for Machine Learning Hyperparameter Tuning: Analysis of the Black-Box Optimization Challenge 2020,” in *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, PMLR, Aug. 2021, pp. 3–26. Accessed: Jul. 17, 2024. [Online]. Available: https://proceedings.mlr.press/v133/turner21a.html

[19] T. Yu and H. Zhu, “Hyper-Parameter Optimization: A Review of Algorithms and Applications,” Mar. 12, 2020, *arXiv*: arXiv:2003.05689. Accessed: Jul. 17, 2024. [Online]. Available: http://arxiv.org/abs/2003.05689

[20] D. Cireşan, U. Meier, and J. Schmidhuber, “Multi-column Deep Neural Networks for Image Classification,” Feb. 13, 2012, *arXiv*: arXiv:1202.2745. Accessed: Jul. 17, 2024. [Online]. Available: http://arxiv.org/abs/1202.2745
[21] Afaq and Rao, “Significance of Epochs On Training A Neural Network,” *Int. J. Sci. Technol. Res.*, vol. 9, no. 6, pp. 485–488, Jun. 2020.
[22] Fortmann-Roe, “Understanding the Bias-Variance Tradeoff,” Understanding the Bias-Variance Tradeoff. Accessed: Jul. 17, 2024. [Online]. Available: https://scott.fortmann-roe.com/docs/BiasVariance.html
[23] C. Xu, P. Coen-Pirani, and X. Jiang, “Empirical Study of Overfitting in Deep Learning for Predicting Breast Cancer Metastasis,” *Cancers*, vol. 15, no. 7, p. 1969, Mar. 2023, doi: 10.3390/cancers15071969.
[24] “CS231n Convolutional Neural Networks for Visual Recognition.” Accessed: Jul. 19, 2024. [Online]. Available: https://cs231n.github.io/neural-networks-3/
[25] R. Wang, S. Malladi, T. Wang, K. Lyu, and Z. Li, “The Marginal Value of Momentum for Small Learning Rate SGD,” Apr. 15, 2024, *arXiv*: arXiv:2307.15196. Accessed: Jul. 17, 2024. [Online]. Available: http://arxiv.org/abs/2307.15196
[26] J. Fu, B. Wang, H. Zhang, Z. Zhang, W. Chen, and N. Zheng, “When and Why Momentum Accelerates SGD:An Empirical Study,” Jun. 15, 2023, *arXiv*: arXiv:2306.09000. Accessed: Jul. 20, 2024. [Online]. Available: http://arxiv.org/abs/2306.09000
[27] M. Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”.
[28] *keras-team/keras*. (Jul. 07, 2024). Python. Keras. Accessed: Jul. 07, 2024. [Online]. Available: https://github.com/keras-team/keras
[29] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” Apr. 10, 2015, *arXiv*: arXiv:1409.1556. Accessed: Jul. 07, 2024. [Online]. Available: http://arxiv.org/abs/1409.1556
[30] Md. S. Mahmud Khan, M. Ahmed, R. Z. Rasel, and M. Monirujjaman Khan, “Cataract Detection Using Convolutional Neural Network with VGG-19 Model,” in *2021 IEEE World AI IoT Congress (AIIoT)*, May 2021, pp. 0209–0212. doi: 10.1109/AIIoT52608.2021.9454244.
[31] B. Vigneshwaran *et al.*, “Recognition of pollution layer location in 11 kV polymer insulators used in smart power grid using dual-input VGG Convolutional Neural Network,” *Energy Rep.*, vol. 7, pp. 7878–7889, Nov. 2021, doi: 10.1016/j.egyr.2020.12.044.
[32] V. Sudha and D. Ganeshbabu, “A Convolutional Neural Network Classifier VGG-19 Architecture for Lesion Detection and Grading in Diabetic Retinopathy Based on Deep Learning,” *Comput. Mater. Contin.*, vol. 66, pp. 827–842, Jan. 2020, doi: 10.32604/cmc.2020.012008.
[33] A. S. Paymode and V. B. Malode, “Transfer Learning for Multi-Crop Leaf Disease Image Classification using Convolutional Neural Network VGG,” *Artif. Intell. Agric.*, vol. 6, pp. 23–33, Jan. 2022, doi: 10.1016/j.aiia.2021.12.002.