Case study of Hyperparameter optimization framework Optuna on a Multi-column Convolutional Neural Network

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Jenia Afrin Jeba

©Jenia Afrin Jeba, October/2021.
Unless otherwise noted, copyright of the material in this thesis belongs to the author. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Or

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

To observe the condition of the flower growth during the blooming period and estimate the harvest forecast of the Canola crops, the 'Flower Counter' application has been developed by the researchers of P^2IRC at the University of Saskatchewan. The model has been developed using a Deep Learning based Multi-column Convolutional Neural Network (MCNN) algorithm and the TensorFlow framework, in order to count the Canola flowers from the images based on the learning from a given set of training images. To ensure better accuracy score with respect to flower prediction, proper training of the model is essential involving appropriate values of hyperparameters. Among numerous possible values of these hyperparameters, selecting the suitable ones is certainly a time-consuming and tedious task for humans. Ongoing research for developing Automated Hyperparameter Optimization (HPO) frameworks has attracted researchers and practitioners to develop and utilize such frameworks to give directions towards finding better hyperparameters according to their applications.

The primary goal of this research work is to apply the Automated HPO Optuna on the Flower Counter application with the purpose of directing the researchers towards among the best observed hyperparameter configurations for good overall performance in terms of prediction accuracy and resource utilization. This work would help the researchers and plant scientists gain knowledge about the practicality of Optuna while treating it as a black-box and apply it for this application as well as other similar applications.

In order to achieve this goal, three essential hyperparameters, batch size, learning rate and number of epochs, have been chosen for assessing their individual and combined impacts. Since the training of the model depends on the datasets collected during diverse weather conditions, there could be factors that could impact Optuna's functionality and performance. The analysis of the results of the current work and comparison of the accuracy scores with the previous work have yielded almost equal scores while testing the model's performance on different test populations. Moreover, for the tuned version of the model, the current work has shown the potential for achieving that result with substantially lower resource utilization. The findings have provided useful concepts about making the better usage of Optuna; the search space can be restricted or more complicated objective functions can be implemented to ensure better stability of the models obtained when chosen parameters are used in training.

Acknowledgements

I would like to acknowledge my profound indebtedness and convey my heartfelt gratitude to my respected supervisors Dr. Dwight Makaroff and Dr. Derek Eager for their constant guidance, advice, motivation and extraordinary patience during my thesis works. Being an international student, they have given me the opportunity to come to this country and pursue this degree. They have been extraordinarily supportive and helpful to me throughout my program. This thesis would not have made at all possible unless they showed me the right direction and advised me in tough situations that were impossible for me to overcome. Besides my supervisors, I would like to thank my other committee members: Dr. Kevin A. Schneider, Dr. Michael Horsch, Dr. Mark Keil and Dr. Daniel Teng for their valuable suggestions and feedback.

Sincere thanks goes to Mohammed Rashid Chowdhury for developing the designated application of this work, which is the "Flower Counter" application, and providing continuous guidance, answers to the in-depth questions about the working mechanism of the application and overall support. Also, special thanks goes to Arjot Roy, who has rendered excellent efforts in setting up the necessary environments for the Optuna framework and in developing necessary scripts for executing the experiments. Besides, thanks goes to Noah Orensa for helping in Spark/Hadoop cluster setups. Last but not the least, thanks goes to all of the members of the DISCUS Lab for their overall support.

I am thankful to the Department of Computer Science of University of Saskatchewan for their generous financial support through stipends and scholarships that helped me extremely as an International student. I would like to convey my warmest thanks to all the staffs of the Department for their constant support. I would also specially like to thank all my friends and seniors who have been helpful and encouraging throughout the period of my thesis work.

I am extremely grateful to my lovely parents, sisters and relatives for their inspiration, love and truest prayers at every stage of my life. Without their endless sacrifice, I would not have come this far. Finally, I would like to acknowledge the immense contribution of my fiancé for his continuous support, motivation, help and encouragement, which has always helped me to move forward and be able finish this degree smoothly. Lastly, I would like to thank almighty Allah (God) for his kindest blessings on me.

This thesis is dedicated to my beloved and respected mother Amina begum and father Md. Jamal Uddin Ahmed, I've come this far today only because of their sacrifices, love, continuous support and motivation; my dearest adorable sisters Jarin Tasnim Ava and Mridula Mehjabin Biva and my beloved fiancé Monzurul Islam. I am grateful to you all forever and can never repay what you have done for me. Thank you very much for your love and support in every stage of my life.

- Jenia.

Contents

Permission to Use i

Abstract ii

Acknowledgements iii

Contents v

List of Tables vii

List of Figures viii

List of Abbreviations ix

1 Introduction 1
- 1.1 Thesis motivation . . . 2
  - 1.1.1 Research questions . . . 3
- 1.2 Limitations . . . 3
- 1.3 Thesis statement . . . 4
- 1.4 Thesis organization . . . 5

2 Background and Related Work 6
- 2.1 Convolutional Neural Networks (CNN) . . . 6
  - 2.1.1 Terminologies related to CNN . . . 6
- 2.2 Overview of TensorFlow . . . 11
  - 2.2.1 Core concepts and execution model . . . 11
- 2.3 Hyperparameter Optimization (HPO) . . . 12
  - 2.3.1 Hyperparameters . . . 12
  - 2.3.2 Popular search methods and algorithms used for HPO . . . 14
- 2.4 Features of Automated Hyperparameter Optimization (Automated HPO) frameworks . . . 19
  - 2.4.1 Distributed and/or parallel computation and scalability . . . 19
  - 2.4.2 Static vs. dynamic search space . . . 22
  - 2.4.3 Early-stopping and pruning . . . 23
  - 2.4.4 Lightweight and ease-of-use . . . 24
- 2.5 Optuna . . . 24
- 2.6 Review of the HPO frameworks . . . 27
- 2.7 Summary . . . 28

3 Experimental Design and Configuration 29
- 3.1 Architecture of the Flower Counter application . . . 29
  - 3.1.1 Overview of MultiColumn CNN-based (MCNN) Flower Counter model's architecture . 29
  - 3.1.2 Datasets . . . 29
  - 3.1.3 Generation of ground truths and density maps . . . 30
- 3.2 Hardware and software configurations . . . 31
- 3.3 Dataset characterization . . . 31
- 3.4 Experimental design and methodology . . . 39
  - 3.4.1 Accuracy metrics used in Flower Counter . . . 39
  - 3.4.2 Measurements/visualization . . . 40
  - 3.4.3 Batch Size . . . 40
  - 3.4.4 Learning Rate . . . 42

3.4.5 Number of Epochs . . . 43
3.4.6 Combined impact of hyperparameters . . . 43
3.4.7 Analysis of Optuna's performance on test datasets . . . 43
3.5 Summary . . . 44

4 Results and analysis **45**
4.1 Experimental results and analysis of batch size . . . 45
4.2 Experimental results and analysis of learning rate . . . 47
4.3 Experimental results and analysis of numbers of epochs . . . 48
4.4 Simultaneous parameter search space . . . 51
4.4.1 Analysis for lower number of epochs . . . 51
4.4.2 Analysis for larger number of epochs . . . 53
4.5 Testing accuracy results . . . 56
4.5.1 Same population experiments . . . 56
4.5.2 Different population experiments . . . 61
4.6 Summary . . . 64

5 Conclusion and Future Work **65**
5.1 Summary . . . 65
5.2 Thesis contribution . . . 66
5.3 Potential future scope of the work . . . 67

References **69**

List of Tables

3.1 Configuration details of the servers used in the experiments 31
3.2 Binning strategy being used for the 2016-all dataset and the 2018-split dataset 35
3.3 Total number of images (before and after binning) used for training for datasets 36
3.4 Training and test datasets for different population experiments 36

4.1 Total number of trials in study per batch size used in batch sizes experiment 47
4.2 Total number of trials in study per epoch used in epochs experiment 49
4.3 Parameters for testing suggested by Optuna experiments 57

List of Figures

2.1 A general architecture of Neural Network (NN) (based on Hyndman *et al.* [43]) 7
2.2 Implementation of 'filter' operation and 'convolution' operation on the input in a CNN (based on Millstein *et al.* [69]) 8
2.3 TensorFlow's dataflow graph (based on Wongsuphasawat *et al.* [94]) 12
2.4 Hyperparameter optimization ('define-and-run' vs. 'define-by-run') (based on Tokui *et al.* [88]) 22
2.5 Architectural overview of Optuna (based on Akiba *et al.* [2]) 27

3.1 Multi-column Convolutional Neural Network architecture of Flower Counter application [19] . 30
3.2 Sample images from different camera-days of 2016-all dataset and corresponding density maps 32
3.3 Sample images from different camera-days of 2018 dataset and corresponding density maps . 33
3.4 Density histograms of training datasets 37
3.5 Density histograms of testing datasets 38
3.6 Boxplot interpretations 41

4.1 Batch size vs. validation loss (single parameter experiment) 46
4.2 Learning rate vs. validation loss (single parameter experiment) 48
4.3 Epoch vs. validation loss (single parameter experiment) 50
4.4 Batch size, learning rate and epochs vs. validation loss (fewer epochs) 52
4.5 Batch size, learning rate and epochs vs. validation loss (more epochs) 55
4.6 Same population experiments (Optuna vs. Chowdhury [19]) 59
4.7 Samples for Actual vs. Predicted counts (same population experiments) 60
4.8 Different population experiments (Optuna vs. Chowdhury [19]) 62

List of Abbreviations

ASHA	Asynchronous Successive Halving Algorithm
AutoML	Automated Machine Learning
BO	Bayesian Optimization
CNN	Convolutional Neural Network
DL	Deep Learning
DNN	Deep Neural Network
EI	Expected Improvement
GD	Gradient Descent
GP	Gaussian Process
HPO	Hyperparameter Optimization
MAE	Mean Absolute Error
MRE	Mean Relative Error
MCNN	Multi-column Convolutional Neural Network
ML	Machine Learning
MLP	Multilayer Perceptron
MVN	Multivariate Normal
MSE	Mean Square Error
NN	Neural Network
PBT	Population-based Training
RF	Random Forest
RMSE	Root Mean Square Error
SGD	Stochastic Gradient Descent
SHA	Successive Halving Algorithm
SMAC	Sequential Model-based Algorithm Configuration
SMBO	Sequential Model-based Bayesian Optimization
TPE	Tree-structured Parzen Estimator
WEKA	Waikato Environment for Knowledge Analysis

1 Introduction

In recent years, Deep Learning (DL) has solved several challenging problems because of its powerful computational technique of applying Neural Networks (NN) for dealing with huge and complex datasets based on powerful Machine Learning (ML) algorithms [21]. ML applications, particularly Deep Neural Networks (DNNs), depend critically on several model specific parameters termed as 'hyperparameters' [10], which if set appropriately, can enhance the performance (accuracy and precision, execution time, usage of resources, etc.) of the model to a great extent. As this is a cumbersome task for humans, Automated Machine Learning (AutoML) frameworks take this responsibility of automatically selecting the better configurations of hyperparameters among hundreds of thousands of possible combinations in order to optimize the performance of the model [42].

Automated Hyperparameter Optimization (Automated HPO) can have a significant and helpful impact on a Neural Network's (NN) architecture, optimization and performance [39]. Currently, it is broadly recognized that, over the default parameter values set by developers in conventional ML practice, tuned hyperparameter configurations can improve the performance of the NN model significantly [32] [62]. If the process of optimization is aided by some kind of Automated HPO, the manual efforts and elapsed time can be reduced to a great extent [51]. Optuna [2] is one such Automated HPO framework that has presented innovative searching mechanisms for providing hyperparameter optimization in the perspective of above-mentioned scenario. Optuna provides APIs for constructing the search space of different hyperparameters at runtime, efficient strategies for searching and pruning for achieving cost-effective optimization and an easy-to-setup architecture which can work with any experimental setup, as in, from interactive, light-weight experiments to heavy-weight distributed computing [2].

The main objectives of this research work are the following:

- understand the behaviour and assess the impact of an Automated HPO in the context of a case study of a DL application,
- observe and analyze the corresponding individual or combined impact of varying the selected hyperparameter values of the model on its accuracy and resource usage when using diverse training and testing datasets,
- observe and evaluate the results of tuning the model with previous similar research [19] in order to identify if Optuna has been able to find good and/or stable hyperparameter configurations for the considered application, and

- compare the results with the previous work in terms of different accuracy metrics to understand Optuna's practicality.

More details about the algorithms applied in Automated HPO and the Optuna framework will be discussed in Chapter 2.

1.1 Thesis motivation

The Canada First Research Excellence Fund (CREF)[1] has funded the Plant Phenotyping and Imaging Research Centre (P^2IRC)[2] of University of Saskatchewan (USask) to develop a system called, Camera On A STick (COAST)[3] system. In this system, cameras were set up in flower fields and each camera was positioned in an individual plot to capture a series of time-lapse images. The image data recorded at periodic intervals by the cameras from the fields help the researchers to study different growth patterns of Canola plants and understand the impact on crop harvest. These images were captured each day from dawn till dusk, mostly during the summer season of 2016, 2017 and 2018 by using the COAST system from different fields.

The Laboratory in Computer Science department of Usask for Performance Studies of Distributed Computing Systems (DISCUS lab) is allied with P^2IRC for the research of high-performance computing and analysis. One of the projects has been to optimize compute-intensive image processing applications that usually take excessive time to complete. One such application is a DL application, the Canola Flower Counter [19], that has been developed for monitoring the shifting of blooming period and the status of flower growth within the flowering portion of the season. The dynamics of flowering are an indicator of the season's harvest potential and general hardiness (health and robustness of the crop under changing environmental conditions, resistance to drought, pests/weed infestation etc.) of different varieties of crops.

Since DL requires huge datasets and complex image processing algorithms [35], the application needs powerful computational resources and extensive computing time to process results. Fine-tuning of the application using Automated HPO could help researchers to utilize fewer resources and shorten training time by suggesting possible hyperparameter configurations instead of manual trial-and-error to set hyperparameters; at the same time, this may also greatly enable augmentation of the performance with respect to accuracy of the application.

Optuna's features of selecting among the best observed hyperparameter configuration settings and performance optimization capabilities might have significant influence to achieve this performance goal. Thus, the main motivation of this research work is to present a case study to aid the researchers of P^2IRC, the plant scientists who are working with complex NN applications for image processing and other interested parties, in the pursuit of insights regarding the applicability of Automated HPO to their applications, to be able to reduce manual effort and elapsed resources required to provide acceptable prediction accuracy.

[1] https://www.cfref-apogee.gc.ca/home-accueil-eng.aspx
[2] https://p2irc.usask.ca/
[3] https://p2irc.usask.ca/theme-pages/computing/project-3-2.php#PracticalApplications

1.1.1 Research questions

The objective function that has been chosen for Optuna is to minimize the average pixel-wise loss on the validation dataset of the Flower Counter application [19]. The pixel-to-pixel loss is computed by the TensorFlow application and Optuna component uses that result. It is computed individually between every pair of pixels of the prediction and the original images and then averaged over all the pixels; thereby, equal value is assigned to every pixel in the image [78].

According to the objectives mentioned earlier, this research work aspires to answer the following research questions:

1. How should a deployment of Optuna best use the hyperparameter search configurations?
    - should it be applied to optimize one parameter at a time, or multiple (e.g. 2) parameters at once, or all the considered parameters at once?
    - what search range should be given to Optuna for each hyperparameter, a huge range, so as to make sure that Optuna investigates all feasible values of the hyperparameter? or, a narrow range, but then how should such a narrow range be determined?
2. Would the hyperparameter search space set up using Optuna be able to select good configurations for the Flower Counter application?
    - if yes, then how much optimization and performance enhancement in accuracy has been provided by Optuna suggested parameters compared to the previous version of untuned application?
    - if no, then what are the potential reasons that Optuna was unsuccessful in hyperparameter selection?
3. Are the suggestions by Optuna among the best observed hyperparameter values settings within a given range stable or reproducible?
    - if yes, then does it work for all the datasets?
    - If not, then what are the possible reasons?
4. What are the primary reasons behind the performance variability of the tuned Application?
    - Is there any impact of the datasets on Optuna's effectiveness?

1.2 Limitations

There are certain constraints under which the experiments are executed and analyzed; therefore, this might have an impact on the stability of the results, such as

- Since the main aim of this work is to conduct a case study, both the Flower Counter application and the HPO Optuna have been therefore treated as blackboxes; thus, neither the detailed architecture and functionality analysis of the designated application's model nor the tweaking of several other high-functionality APIs offered by Optuna, have been explored deeply in this work.

- The specific datasets and the binning strategy are chosen based on some preliminary experiments for this work; they are not guaranteed to yield the best results from the experiments with respect to prediction accuracy; different datasets, different combinations of binning strategies and their associated distributions can certainly have significant impact on Optuna's performance.

- Under each 'study' (optimization based on an objective function), Optuna runs 'trials' (one single execution of the objective function). There could be multiple number of trials in every study; based on the size of the datasets, time constraints, availability of the machines etc., it has been fixed to 60 per study.

- In this work, a single objective function (which is, minimizing the average pixel-wise loss on the validation dataset) has been chosen for Optuna. The other options for objective functions have not been explored.

- Multiple people were responsible for doing the annotation and labeling of flowers in order to prepare the ground truth images for training the model, so the choices of images for annotation was somewhat dependent on the individual's choice, hence the annotated images from each dataset might lack appropriate consistency for robust learning; therefore, there might be inconsistencies in the labeling. Also, determination of flower locations in dense images is objectively challenging.

1.3 Thesis statement

This thesis determines whether the Automated HPO Optuna can select appropriate hyperparameters that provide good training and flower counting prediction results while reducing the error rates of the accuracy metrics considered to reasonable levels. This will help the researchers of P^{2}IRC to analyze the impact of the Automated HPO framework, Optuna, on the performance of a CNN-based DL application.

Several sets of experiments are conducted to show the circumstances where HPO works well or does not work well. After that, the results derived from the corresponding training-testing on different datasets are analyzed to determine if Optuna enhances the performance of the application in terms of the objective function and resource utilization compared with the previous research [19]. Later, the comparison of the results are investigated thoroughly to infer the potential reasoning of the corresponding performance. Different sets of experiments are conducted to learn how to make the best use of Optuna (at least, for the type of application exemplified by the Flower Counter), so that others (e.g. in P^{2}IRC) can make the most effective and efficient use of this technology.

1.4 Thesis organization

The remaining chapters of this thesis are arranged as follows:

Chapter 2 starts with a brief description of background for understanding the basic concepts of CNN, DL framework Tensorflow, HPO and its popular search techniques, the algorithms of Automated HPO etc. Later, it presents the works related to hyperparameter tuning and the optimization frameworks used for ML/DL applications and their feature-wise elemental comparison along with a detailed description about Optuna.

Chapter 3 presents brief description of the Flower Counter model and its fundamental architecture, a concise review about the characteristics of the datasets used for the experiments and the hardware-software configuration settings used for the experiments along with representative sample images from different datasets and their distribution.

Chapter 4 presents the details of the conducted experiments, the results, evaluations and analysis of the results and comparison of the performance with the previous work.

Chapter 5 draws conclusions regarding the performance comparison of Optuna and then indicates the potential future areas that could be explored on the basis of this work's results and analysis.

2 Background and Related Work

This chapter starts with a brief description of Convolutional Neural Network's (CNN) common terminologies and core concepts of the DL framework Tensorflow that has been used to develop the Flower Counter application being considered. Next, it presents a background on the basic concepts of hyperparameters, hyperparameter tuning and optimization and popular hyperparameter search algorithms for ML/DL applications. Later, the existing research works on Automated HPO frameworks are discussed followed by the detailed description of Optuna's architecture and the reasons of its preference for this work.

2.1 Convolutional Neural Networks (CNN)

Convolutional Neural Network (CNN) is a Deep Learning (DL) algorithm which takes in an input image, passes it to neurons through a series of hidden layers (configurable connected units) [67], which are connected to all the neurons of the next layer, assigns learnable parameters of importance (e.g., weights and biases) to different objects/aspects in the input image to distinguish one object from the other, learn that and generate a prediction of class scores [3].

The number of parameters and the pre-processing required in a CNN is lower and the focus is more on domain-specific features rather than spatial features (CNNs learn the features from the training images based on their attributes regardless of their positions) [69]. Moreover, CNNs have the ability to learn the characteristics automatically by themselves if enough training is provided, allowing CNNs to handle complex image processing tasks. A depiction of a regular NN can be seen in Figure 2.1 (based on Hyndman *et al.* [43]).

2.1.1 Terminologies related to CNN

In a CNN, the most crucial building block is the 'convolution (CONV)' layer which consists of a set of N dimensional arrays called 'filters', which slide across the pixels along the height and width of the incoming input image. During the sliding, dot products are computed between the values of the of the input pixels and the values of the filter; a 2-dimensional activation map is created as a result. Throughout the training time, when there is any presence of certain visual elements such as shapes, patterns, or edges of particular orientations, the CNN models learn those filters that get activated [52]. A CNN then trains a model using a backpropagation [53] [54] technique, learning from errors and rectifying them accordingly. The ultimate goal of a CNN is reducing the value of some type of loss function and achieving as much prediction accuracy as

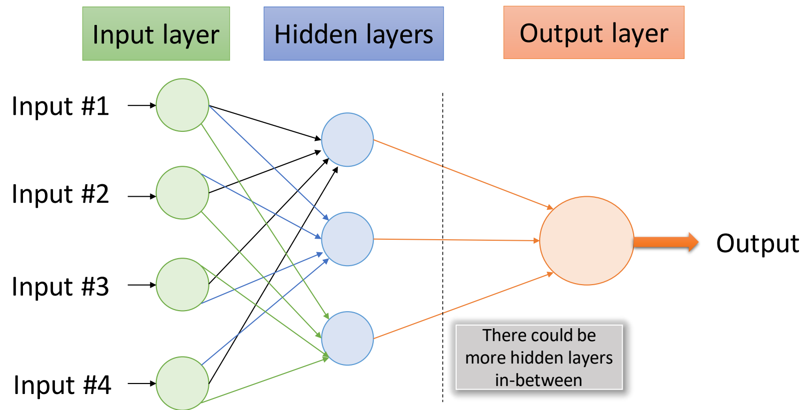


Figure 2.1: A general architecture of Neural Network (NN) (based on Hyndman *et al.* [43])

possible for images outside the training dataset.

There are many essential terms [69] related to CNN which are used repetitively throughout the rest of this thesis document. Therefore, a brief description of those terminologies are explained below:

Filters (Convolution Kernels): A filter (or kernel) refers to an operator which is a smaller-sized matrix (smaller than the input image's dimensions). It is slid over the entire image in order to transform the encoded information in the pixels of that image and then convolved (a process similar as any general matrix multiplication) with the input to obtain 'activation maps' (shown in Figure 2.2). The 'activated regions' are the regions where features identifiable to the kernel have been spotted from the input. Activation maps indicate these regions, so that the network can learn which regions are worthy of extracting features from the input.

Pooling: There is another layer called 'pooling (POOL)' which is placed over the image after CONV layers for retaining the most important features by reducing the dimensionality (the number of trainable parameters) of each feature map. The technique is similar to the CONV layer, there are fixed sized filters which slide over the convolved inputs passed to them. Instead of computing the dot product, the max value of the input region upon which the filter is positioned currently, is taken by the filter. POOL filters can also be set to take the average value instead of taking the max value (max POOL was used for the Flower counter application considered in this work). Despite being destructive in nature, the POOL layer usually saves a lot of computations and helps the CNN to focus on the most important features [80].

Weights: A weight is a value which is multiplied with every input when it enters the neuron. The weights are initialized randomly and updated during the training process. If, after training the model, a higher value of weight is assigned to a particular input, then it is considered to be more important compared to others. The strength of the connections are controlled by the weights between two neurons. The weights are assigned

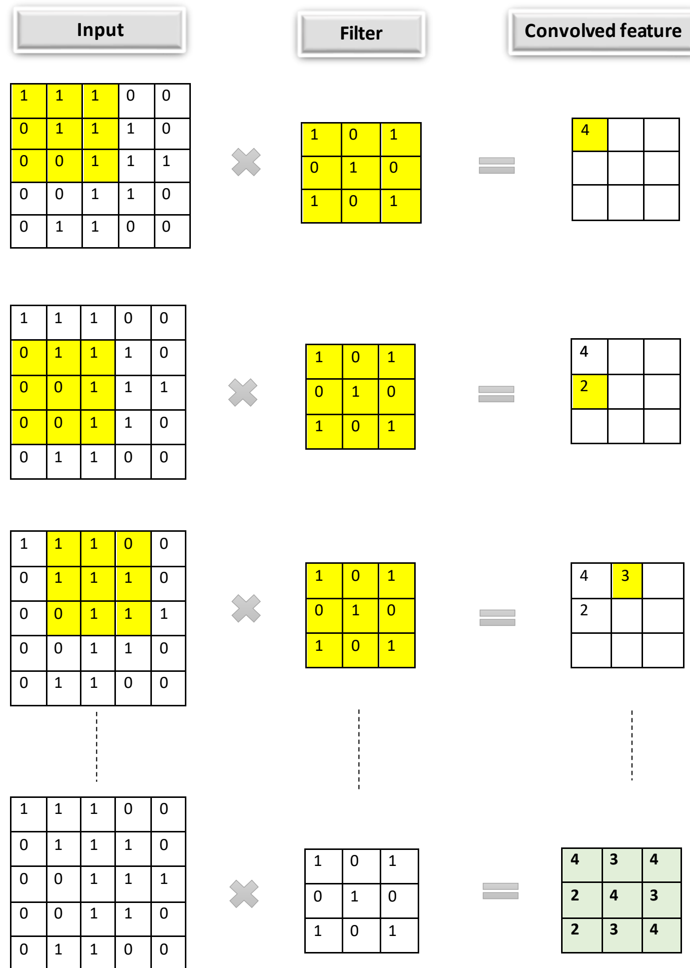


FIGURE 2.2: Implementation of 'filter' operation and 'convolution' operation on the input in a CNN (based on Millstein *et al.* [69])

internally by the CNN model itself based on the amount of influence the input will have on the output. More weight on a particular feature indicates that it is an important feature and vice-versa.

Bias: Another linear component called 'bias' is also applied to the input in addition to the weights. Bias is a constant with the value of 1 and an additional input into the next layer. Since the bias does not have any incoming connections, it is not influenced by the previous layer. However, it has outgoing connections with its own weights. It is guaranteed by the bias unit that there will still be an activation in the neuron, even when all the inputs are zeros.

The weight is multiplied with the input and the bias is added like this:

$$output = sum(weights * inputs) + bias.$$

It is used so that the output adjusts with the weighted sum of the inputs.

Activation Function: It is a non-linear function that is applied to the previous result in order to transform the input signals into output signals. After applying the activation function $f()$, the output looks like

$$output = f(sum(weights * inputs) + bias).$$

This function is used to make a decision whether a particular neuron would fire or not. The most used activation functions are: ReLU, Sigmoid, Softmax, etc. [69].

Loss function: It is the method of evaluating how good or bad the CNN model is performing on training and validation sets after each iteration. For every example in training or validation sets, the sum of the errors made, is calculated; this process is interpreted as 'loss' function.

In general, the loss function is a measurement of the difference between the ground truth value and the predicted value. The objective of the model is to minimize this loss function in order to increase the accuracy.

Epoch: One epoch refers to the traversing of the whole dataset once. For a CNN, training with the same dataset is reiterated several times. It helps the CNN to learn from its mistakes that it made in the prior epochs so that it can rectify the errors in the next time when it sees similar data and provide better prediction results.

Learning Rate: During training, the speed at which an ML model 'learns' (or the rate at which the weights are updated) is known as the 'learning rate' or the 'step size'. The goal is to reach to the minima of the loss function. In response to the estimated error, it controls how much to change the model every time when the model weights are updated. The rate should be chosen very carefully; it should neither be very small that the model takes too long for the model to converge, nor should be very large that the model might miss the optimal solution.

Gradient descent (GD): In order to minimize the loss function, there is an iterative method of optimization called Gradient Descent (GD), which calculates the slope of the loss function and gradually corrects the model parameters, such as the biases and weights as per that slope. The goal of any successful imple-

mentation of GD is to ensure that a model can converge to a state with as low as model errors possible [54].

Mini-batch stochastic gradient descent (mini-batch SGD): Among few variants of GD, Mini-batch Stochastic Gradient Descent (mini-batch SGD) is a kind where only a mini-batch of data is arbitrarily chosen and taken from the dataset instead of loading the whole dataset during each training step. Since the real-life datasets tend to be very large in size, hence for training the model it becomes almost impractical to load the whole of it into memory. For the selected mini-batch, the gradients are calculated and this procedure is repeated so that every batch is used at least once. For instance, if there are N samples in a dataset and if 10 samples are randomly chosen to be picked for every batch, then to complete an epoch there will be N/10 training steps [54].

Backpropagation: This is the technique of propagating the total error or loss back into the Neural Network in order to calculate how much of the loss the nodes are responsible for and update them accordingly. The model learns the features and outputs for a single iteration after training the CNN model using the weights and bias values. Once the output is received, now it is time to calculate the error and feed it back through the hidden layers (in the reverse direction) along with the loss function's gradient in order to update the weights of the model and minimize the error. This process as a whole is called backpropagation. After this, the model corrects itself to an extent and again iteratively keeps updating the weights via backpropagation unless a optimum minima value of the gradient is found [53].

Dropout: In order to prevent over-fitting (the model tends to memorize the training data, thus provides very good accuracy during training but performs poorly on testing data) of the model, dropout sacrifices some precision by randomly dropping a certain number of neurons in the hidden layer during training.

Accuracy: This is a metric for evaluating the performance of the CNN models. It is the fraction of correct predictions that the model could predict correctly. The accuracy of a model is usually calculated in the form of a percentage (%). Compared to the true data, how accurate the model's prediction is coming out, is called the 'accuracy' (for example, among 1000 test samples, if the model identifies 990 samples correctly, then the accuracy of the model is 99.0%.).

There are two other metrics related to accuracy, namely, Precision and Recall; they are also measured in percentages and used to measure how often the model is actually correct when it predicts positive or negative (based on the model's specifications on a certain threshold, the relative error less than that threshold is a 'True' prediction and a large relative error is a 'False' prediction). Their mathematical representations are presented in the following equations.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ positive}, \text{ and} \tag{2.1}$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}. \tag{2.2}$$

There is another metric called 'F1 score' which is calculated using Precision and Recall. If there is an

unbalanced class distribution (e.g., True Negatives are larger in number), then it can be used to seek a balance between Precision and Recall [77].

$$F1\, score = 2\, *\, \frac{Precision * Recall}{Precision + Recall}. \tag{2.3}$$

2.2 Overview of TensorFlow

For facilitating and deploying large scale ML models in diverse heterogeneous environments, researchers developed an open source ML framework called TensorFlow [1]. This framework has made the process of creating NN applications flexible by offering both local and distributed training options by offering a dataflow-based programming abstraction for every types of users. A high-level scripting interface wraps the construction of the dataflow graph which allows the users to conduct experiment with the use of their self-defined optimization algorithms and NN architectures without modifying the core system. For leveraging faster training, there is Distributed TensorFlow that can effectively cope with hundreds of GPU-enabled servers.

2.2.1 Core concepts and execution model

TensorFlow uses an integrated directed dataflow graph to characterize the operating states and computations of an algorithm. The graph encompasses edges and nodes, where the edges correspond to the flow of data and each node or vertex corresponds to a unit of computation. The data that flows through these edges is called a 'tensor'. These can consist of multidimensional arrays of several data types, such as string, int32, float16 or float32, etc. The calculations performed at the nodes are called 'operations'. Some example operations are addition or matrix multiplication [1].

A mutable buffer, called a 'variable'[1] is used during training. It is used to hold 'shared parameters' (they are made of tensors) of the model. It is often required that the dataflow-graph of the NN application has to be rerun several times. However, tensors cannot live past after the completion of a single execution of the graph. Thus, variables are used to persist the tensors.

There are two sections in a TensorFlow application: specifying the computational graph and performing the execution of the defined graph. The graph in the TensorFlow system is executed through the initiation of a 'session'. There is a primary function called 'run' inside the session interface. The 'run' function executes the operations that are invoked for execution by the client program. It also takes care of the process of finding the order of execution before evaluating the requested operation based on the dependencies of the nodes. After that, the executed tensors are fed to computational nodes (operations) by 'run'. The 'run' calls are executed thousands or millions of times in order to execute the graphs or subgraphs once a 'session' is set up[73].

[1] https://www.tensorflow.org/guide/variable

A typical TensorFlow application's dataflow graph can be composed of several subgraphs which can be run simultaneously and via shared variables and queues. The graphical representation of TensorFlow's dataflow graph is given in Figure 2.3 (based on Wongsuphasawat *et al.* [94]).

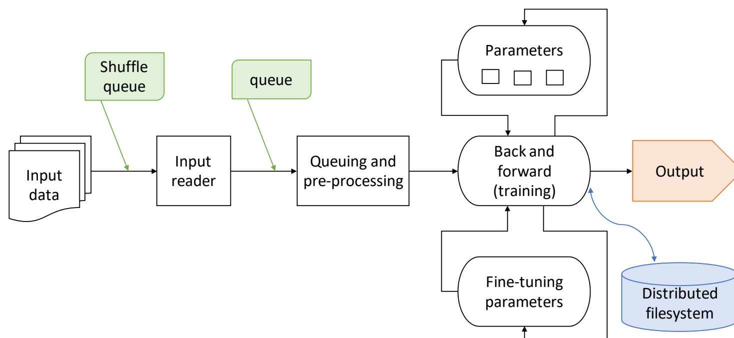


FIGURE 2.3: TensorFlow's dataflow graph (based on Wongsuphasawat *et al.* [94])

To describe briefly, at the beginning of the application, the execution starts with subgraphs that read inputs from the filesystem and preprocess the input data. Next, there is the core training subgraph which takes on the processed input batches, runs a training session and saves the trained model parameters in a distributed file system using a periodic separate checkpointing subgraph to ensure fault tolerance. The checkpointing subgraph and the training subgraph can be run concurrently. Similar concurrency can be achieved in the initial subgraphs of the application as well. User provided inputs from the queues can be decoded by the preprocessing subgraph and concurrently the callable functions can be mapped to the sequence of input data [94]. TensorFlow's distributed execution has been simplified by the method of explicitly defining the communications among the subgraphs in the dataflow graph. A particular device such as CPU or GPU is assigned to execute individual operations.

2.3 Hyperparameter Optimization (HPO)

2.3.1 Hyperparameters

Hyperparameters are those model-specific parameters that are 'fixed' prior to training or testing the ML/DL model. For instance, in case of a Random Forest classifier, the number of decision trees in the forest is a hyperparameter [76]; for an NN, the number of hidden layers, the learning rate, the number of units in each layer, batch size, etc. are the hyperparameters [13].

The method of searching for the 'right set of hyperparameters' that can help to obtain better results and performance for any ML/DL model, is known as Hyperparameter Optimization (HPO). In the process of

building ML models, optimizing hyperparameters constitutes one of the trickiest parts. The goal of HPO is to find the potential good configuration values of hyperparameters of a given model which would yield better performance [36]. In contrast to model parameters, hyperparameters are set by the experiments prior to training. Briefly, the objective is to find the hyperparameters which, provided on the validation set metric (e.g., accuracy and loss function), give a better score [36]. Some examples of objective functions could be the following [42]:

- minimizing the cross-entropy loss function,
- maximizing the log-likelihood function,
- maximizing the total reward/value function in Reinforcement Learning,
- maximizing a fitness function in Genetic programming, minimizing the hinge loss in Support Vector Machine, and
- minimizing the MSE function in linear regression, Classification And Regression Trees (CARTs), decision tree regressions, etc.

An objective function takes hyperparameters that the user wants to optimize as input and provides a real-valued score as output that is to be maximized or minimized [20]. Since the hyperparameters are mapped to the probability of a score on the objective function from a high-dimensional perspective, it is sometimes called a 'response surface'. For example, for a Random Forest regression [85] problem, the user might want to get a good set of values for some of the hyperparameters such as maximum depth of the trees, number of estimators, etc. and minimize the score of RMSE (objective function) [18]. Inspite of the fact that the objective function might appear to be quite simple, its computational cost could be in fact very expensive! If the model is simple, the hyperparameters to be evaluated are simple and small in number, and the amount of data on which the objective function has to be evaluated is small, etc., then it could be calculated quickly like in Grid search [14], that is, by trying every single possible hyperparameter combination. Otherwise, there might be situations where it might take hours or even days for evaluating the objective function, resulting into a very expensive search [97].

Each time when a different hyperparameter set is considered, a model has to be trained on the training data, and the predictions should be made on the validation data and then the validation metric is calculated. With the infinite numbers of hyperparameters available to choose from and complex DNN models that take days to train, selecting suitable and close to optimal hyperparameters could become the prime challenge and infeasible to do manually! Some popular searching methods that are used for HPO are described in the following sections.

2.3.2 Popular search methods and algorithms used for HPO

The most basic HPO method is **Grid search** or **Exhaustive Search**. In Grid Search, a predetermined list of hyperparameter values is taken into consideration, every combination is applied to evaluate the model for each of them and the best accuracy is noted [13]. In this search, all the hyperparameter values are positioned in the form of a matrix. A finite set of values for each hyperparameter is specified by the user and then all the combinations are evaluated. Since the necessary number of function evaluations increases exponentially with the increasing dimensionality of the configuration space, this method suffers from the problem of 'curse of dimensionality'. Grid search facilitates independent searches and evaluations because the current evaluations do not depend on the previous ones, so it is scalable as the searches can be conducted in parallel. However, this strategy wastes a lot of time in exploring parameter values that have poor objective function scores since it treats every evaluation independently and does not keep track of the past evaluations [13].

In **Random Search**, combinations of a range of hyperparameter values are explored at random within the parameter space and are used to obtain better solutions for model optimization. That means some parameters will be sampled within a preset number of iterations at random instead of trying all the parameter values. This searching algorithm actually has no end. Rather, a specific time budget or number of trials is determined as an ending criterion. Bergstra and Bengio [14] showed that Random Search found equal or better results in fewer function evaluations than Grid search and performed more efficiently for HPO while saving a significant amount of time compared to Grid search. However, the tradeoff to the decreased processing time is that there is no guarantee of finding the most optimal hyperparameters combination or even a good one; the search process might miss it because of being random in nature [14].

Both Grid and Random Search often spend a significant amount of time evaluating 'bad' hyperparameters as they are completely uninformed by the past evaluations. Past results can provide significant insights, but these techniques ignore previous results. Therefore, even though the optimal answer might (probably) lie in a small region, they would keep searching across the entire range of the number of estimators [13].

Bayesian optimization (BO) [4] [23] focuses on the most promising hyperparameters. If there is a record of past evaluations, then it would be helpful to use that information to create a probability model for deciding upon the next choice of hyperparameters, whereas Random or Grid search would keep exploring the whole search space. The main idea of BO is to take the objective function and develop a probability model of it and apply it for selecting the most promising hyperparameters for the assessment of the objective function [23].

There are two most important features for characterizing BO method, (1) a surrogate model for the function f, and (2) an acquisition function for making selection about the sampling of the next evaluation points from the hyperparameter space. The **surrogate function** is a model of the original objective function expressed as a probability representation which is built using previous evaluations [9]. It is a cheaper

representation of the original model and is usually used to reduce the time and cost consumption of training and evaluating the original model. The key steps of generating a surrogate model include the selection of representative (evenly spread) samples for training the model across the parameter space, generation of the predictive model using certain statistical distributions (the popular ones used in BO are discussed in the subsequent sections under SMBO technique) based on the labeled training dataset and then evaluating the model.

For an acquisition function, **Expected Improvement (EI)** function [23] is considered as the most popular one. It is defined as $EI(x) = E[max(0, f(x) - f(x^*)]$, where f is the choice of the function used for the surrogate model in order to fit the response surface, x is the hyperparameter to be optimized and x^* is the best value of the hyperparameter so far.

So briefly, the approach goes as follows:

1. a surrogate probability model (often called a 'prior') of the objective function is built,
2. the hyperparameters that perform the best on the surrogate model are explored in each iteration,
3. these hyperparameter values are applied to the true objective function and evaluated using an acquisition function,
4. the surrogate model is updated incorporating the new results, this phase of updating is called 'active learning' (the updated model is often mentioned as a 'posterior'), and
5. steps 2-4 are repeated until maximum time or number of iterations is reached or expected output is received.

In tuning a small number of hyperparameters, BO is efficient, but when the search dimension increases too much, its efficiency degrades quite a bit [25]. This search method is not parallel, in contrast to Random or Grid searches. For launching a new trial, the previous learning needs to be finished. The focus of Bayesian model-based optimization is to choose only the most promising set of hyperparameters based on previous calls to a model of the objective function called 'surrogate' in order to decrease the number of times the objective function needs to be run and evaluated.

Sequential Model-based Bayesian Optimization (SMBO) is a formalization of BO which refers to running trials one after another [41]. By applying Bayes rule each time, it tries to find better hyperparameters and updates the surrogate model accordingly [92]. SMBO differs in the way it builds the surrogate using different methods, but all of those methods depend on information from the history of previous trials in order to propose better candidate set of hyperparameters for evaluating next. The aspects of this model-based HPO go as follows:

1. a domain of hyperparameters is defined over which the search would take place,

2. an objective function is set which takes in the considered hyperparameters as input and outputs,

3. the objective function's surrogate model is formulated,

4. a criterion, also known as a selection function, for choosing next hyperparameters from the surrogate model for evaluation is decided, and

5. to update the surrogate model, a history consisting of (score, hyperparameter) pairs is used by the algorithm.

SMBO has many variations in steps 3 and 4, based on how the surrogate model of the objective function is built and the conditions for selecting the next hyperparameters. The most common choices for the surrogate model in step 3 are: Random Forest Regressions [103], Gaussian Processes [83] and Tree Parzen Estimators (TPE) [83]; and the most frequent choice for the criteria in step 4 is the Expected Improvement (EI) score [13]. These choices of surrogate models are discussed in the subsequent sections.

The efficiency of SMBO lies in the fact that it proposes better candidate hyperparameters for evaluation compared to Random or Grid search. Also, it leads to fewer overall evaluations of the objective function. At the same time, SMBO improves the score on the objective function at a much faster pace than those two search strategies. The computational cost in SMBO is actually much cheaper than evaluating the actual objective function; this method is useful where evaluating the objective function is costly in terms of money or time (e.g.- in Oil exploration [70]). However, significant time is spent in this algorithm for selecting the next hyperparameters in order to achieve the maximum EI. It has been proved that when SMBO is applied, it takes hours for evaluating the actual objective function, but it requires only several seconds to find out the next suggested set of candidate hyperparameters [13]. For choosing the next hyperparameters, less time would be spent in evaluating poor hyperparameter choices if better-informed methods are used. So, in a brief, there are two key advantages of using SMBO:

- overall runtime for HPO would be reduced, and
- it would be easier to achieve better scores on the objective function and also the test sets.

Random Forest (RF) regression is a Supervised ML technique that is a popular choice for constructing surrogate models [4]. It follows Ensemble learning (a technique where multiple ML algorithms make their own predictions which are later combined together for making more accurate predictions compared to any of those individual models) used for regression (also classification) problems. At training time, it constructs a multitude of decision trees and as output, it generates the mode of the classes (for classification problems) or the mean prediction of the individual trees [85]. RF technique is often used to build the surrogate model for HPO [103].

There are several benefits of RF, for which it is a great choice for constructing surrogate models [4]. It acts as an efficient and highly accurate classifier or regressor for large datasets. It can handle several

missing data and numerous input variables but still focus on the important variables for producing correct classification results. During the process of building forests, it produces an internal impartial calculation of the generalization error for accurate outputs.

Gaussian process (GP) [93] is a collection of random variables indexed by time or space, such that every finite linear combination of them is normally distributed, i.e. every finite collection of those random variables has a Multivariate Normal (MVN) distribution [16]. In the MVN or Gaussian distribution, the distribution is generalized to multi dimensions from the one-dimensional normal distribution [4].

In the area of ML, Gaussian Process Regression (GPR) is a form of non-parametric Bayesian approach to regression. The term 'non-parametric' means that it is not limited by any particular functional form, so the curves in the graph can take any shape according to the data. This is a popular method because it works efficiently on small to large scale datasets. Also, based on probabilistic model, it is able to provide predictions on uncertainty measurements. Therefore, GP provides information about both the likely value and the uncertainty range around that value of function f.

Tree-structured Parzen Estimator (TPE) is an iterative method that proposes the next set of hyperparameters to be evaluated based on the probabilistic model created from the history of evaluated hyperparameters (the initial evaluations are performed based on Random Search). TPE method treats categorical hyperparameters in a tree-structured fashion [13] and applies Bayes rule to build a probabilistic model [4]. For TPE, Bayes' rule makes an estimate of an initial surrogate model of the objective function; this gets updated every time when more samples are drawn. Consequently, after a specified number of evaluations on the objective function, it is hoped that the model would almost accurately reflect the behaviour of the objective function and the hyperparameters that have maximized the objective function are the ones which have produced the greatest EI. SMBO with TPE method has shown to perform better in structured and high-dimensional model space [59]. A drawback of TPE is that the interactions between the hyperparameters are modeled less explicitly compared to GPs.

Population-based Training (PBT) is an efficient algorithm for asynchronous training and optimizing a NN within a fixed computational budget of resources [46]. PBT connects and expands the above stated two search methods which are Random Search and Bayesian search. At the beginning of the search, PBT starts training the NN with random hyperparameters like Random Search. PBT collects information from the workers or population for refining hyperparameters and directing the computational resources for promising runs. Here in PBT, a worker can copy or replicate the parameters of a better performing worker and search for new hyperparameters by altering the existing values. With the progress of the search process, PBT identifies better hyperparameters and dedicates more training time and resources to the promising trials, directing to automatic learning of the finest configurations learned from the workers or population like the Bayesian search. Therefore, this method offers faster learning, reduced usage of computational resources and

conservation of time [46].

PBT fuses ideas from genetic optimization algorithms. Some well-known PBT methods are Genetic algorithms [30] [91] , Evolutionary algorithms and Evolutionary strategies [7], Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [44] [61], and Particle Swarm Optimization algorithms [8].

Evolutionary optimization has been greatly used in HPO for statistical ML algorithms, Automated ML, DNN training and architecture search [98]. Evolutionary optimization algorithms are used for noisy black-box functions and originated from the biological 'evolution' concept (via mutation and recombination). The process briefly follows the steps mentioned below.

1. an initial population is created containing random solutions (e.g., tuples of hyperparameters are generated randomly, usually 100+),

2. the hyperparameter tuples are evaluated and their fitness function is calculated accordingly (i.e., with those hyperparameters, 10-fold cross-validation accuracy scores of the ML algorithm is checked),

3. the hyperparameter tuples are ranked by their relative fitness score (e.g., cross-validation score),

4. the worst-performing hyperparameter tuples are replaced with new hyperparameter tuples with the help of mutation (evolution), and

5. steps 2-4 until are repeated unless algorithm performance is satisfactory or it is not improving anymore.

Early stopping-based search strategy is very useful when search spaces are vast in size, containing a mixture of continuous and discrete hyperparameters and also the computation of loss function and the evaluation of a set of hyperparameter result in quite high computational costs. Successive Halving Algorithm (SHA) [55], Irace [63] are some early stopping-based optimization methods which initiate the process as a Random Search, but periodically applies statistical tests and analysis for pruning poor-performing configurations and focusing on the promising ones instead. Asynchronous Successive Halving (ASHA) [55] is the asynchronous version of SHA for further improving SHA's performance by eliminating 'synchronous' evaluation and pruning strategy of underperforming models. Hyperband [56] is another early stopping-based algorithm that calls upon SHA or ASHA several times with different levels of pruning mechanisms with fewer numbers of inputs.

The algorithm in ASHA is performed asynchronously and in parallel for searching high accuracy hyperparameters while escalating the computational proficiency. In a parallel distributed environment and in terms of both efficiency and accuracy, ASHA can accomplish improvements by executing a Random Search with active early stopping. The results in ASHA showed that it could achieve linear scalability with the increasing number of workers in a distributed environment.

2.4 Features of Automated Hyperparameter Optimization (Automated HPO) frameworks

There is neither any ultimate one-stop solution about what is the best HPO to use, nor any universal HPO algorithm which can provide the best performance over all problems, since the types of ML problems vary so much from one to another. All HPO algorithms share the following identical workflow, which comprises major three steps:

1. search space and configuration settings are initialized,
2. values for hyperparameters are proposed, and
3. the model is trained and the result is updated.

2.4.1 Distributed and/or parallel computation and scalability

Hyperopt [15] can implement parallel evaluation of trials by utilizing a cluster of computers. The objective function is provided with a handle to Mongodb [17] databases for running a search in parallel in Hyperopt. Therefore, the objective function can update the database even with partial results, interact with concurrent processes, etc. [15]

Spearmint [83] is a software package designed to perform BO and run experiments automatically iteratively adjusting a number of parameters and minimize some objective functions using as few runs as possible. It uses GP and accomplishes sampling over the hyperparameters. It can be implemented in parallel on a single machine by using multiple processors, as well as in a distributed cluster environment. Spearmint offers 'driver' modules for enabling different environments. For running on a single machine with likely many cores, the 'driver' flag is set to '–driver=local', which will spawn a new process on the current machine. Distributed experiments can be run in parallel on a multi-node cluster with the help of Sun Grid Engine (SGE) [29]; the 'driver' module is then set to '–driver=sge'.

HPOlib [23] is another HPO library where, the function evaluations can be executed in parallel by setting the argument 'n_jobs' offered by HPOLib if the target machine has enough processors/cores. The authors suggested that the value should not be higher than the available number of cores in the machine).

HyperOpt, Spearmint, and HPOLib are open-source systems which can act as distributed model selection and training tools for both the search and evaluation of the model. Nevertheless, they require manual administration of computational resources across a cluster and are tightly coupled to the search algorithm structure [59]. Moreover, they do not allow for intermediate control of trial execution; rather they treat a full

trial execution as an atomic unit, as a result, efficient usage of cluster resources is impeded by them. These solutions are mainly devised for research purposes and hence challenging to expand or incorporate with other algorithms. As a result, users face challenges to switch among these choices without substantially changing their existing code base [59].

The **Covariance Matrix Adaptation Evolution Strategy (CMA-ES)** [61] is an HPO that is very useful for continuous black-box DNN models. Loshchilov *et al.* have shown that, while BO techniques usually work better for function evaluation within smaller numbers of hyperparameters (e.g., when the numbers are below 10), CMA-ES performs better for higher numbers of hyperparameters (e.g., more than 10) [34].

CMA-ES is an iterative algorithm. It samples candidate solutions from an MVN distribution in each of its iterations. The solutions are evaluated in parallel or sequentially and then, the good samples with higher probability distributions are used for the next iteration. CMA-ES has proved to outperform BO and thus it could be a great choice for parallel HPO (it optimized 19 hyperparameters of a DNN in parallel on 30 GPUs [61]). However, a much broader comparison on parallel computation should be executed comprising of various modifications and more test problems to meet various constraints as stated by Gelbart *et al.* [28].

Auptimizer [59] is another HPO framework, where for training models, users can use all available computing resources in a distributed environment. It has integrated Random search, Grid search, and open-source Auto ML/HPO libraries/software such as AutoKeras [47], Spearmint, HyperOpt, Hyperband [56] (discussed later), Bayesian optimization (BO) and Hyperband (HB) (BOHB) [24], etc. to be used as the search techniques.

It deploys a pool of computing resources for scalability and users just need to specify the resources required. For the HPO process, the user can scale out the experiment by specifying the number of jobs that can be executed in parallel (concurrently) on the CPU resources via using 'n_parallel' and number of processes required by the jobs by using 'n_samples' as offered by Auptimizer. Furthermore, it offers extensibility via enabling easy integration of new HPO algorithms according to the specified interface.

However, ensuring smooth communication among the heterogeneous machines that jobs run on and between the jobs themselves, is a key challenge of usability, which has scopes for improvement. In future releases, they intend to launch other services as, model compression, etc.

Ray [72] is a distributed system, maintained by a single dynamic execution engine, that employs a unified interface to convey both actor-based [12] and task-parallel computations; it offers distributed HPO by leveraging its combination with Tune [57]. This distributed cluster-computing framework has provided a single platform for training, simulation, processing large inputs (e.g. images, video), efficiently for ML applications. Generally, model training and hyperparameter tuning of any CNN model comprises vast datasets. Running them on a single GPU can take several days. Therefore, a cluster-based system in a distributed environ-

ment can be very advantageous in such situation by employing several machines in parallel. Ray helps to accomplish it by using their offered APIs, such as 'train_policy.remote()'.

Tune is an open-source unified framework for distributed ML model selection and training based on Ray [72]. It offers various HPO algorithms, such as BO, PBT, HyperOpt, HyperBand/ASHA [56], etc. Tune can be used for executing experiments using large clusters and tuning hyperparameter at any scale. Tune allocates CPU and GPU instances to each 'trial' (a single training run) through the help of Ray. Furthermore, using Ray APIs, individual trials can launch further subprocesses and leverage themselves distributed computation. Nevertheless, their adaptation is noticed to be often ad-hoc in nature to the distributed computing environment.

Katib [104] is a scalable AutoML platform, which is deployed on top of Kubernetes-based distributed systems. A wide range of AutoML algorithms (e.g., BO technique, Hyperband, Grid search, Reinforcement Learning [84]) together with both hyperparameter fine-tuning and Neural Architecture Search (NAS) [58] are supported. It divides the system into discrete components that are encapsulated as microservices and each of them operate within a Kubernetes pod. They have integrated Google Vizier's [31] search techniques with their design. The search can be accelerated by spawning multiple parallel 'trials' in each iteration offered by their controller. The Vizier Core is furthermore connected to a MySQL database that manages the data communication between the containers; hence, flexible management and scalable deployment are ensured. Nonetheless, they have mentioned that they would be integrating parameter sharing for facilitating advanced acceleration techniques.

Autotune [49] is an automatic parallel derivative-free [79] optimization framework. It is capable of applying multiple occurrences of global and local search algorithms (e.g., Genetic Algorithms (GAs) [33], BO technique) in parallel. Multiple worker nodes can be assigned to each model to be trained in Autotune; also, multiple models can be trained and/or tuned in parallel. Autotune operates on SAS Viya [27], which supports cloud computing and enables distributed analytics. The system keeps performing simultaneous global and local searches while uninterruptedly distributing computational resources and function assessments.

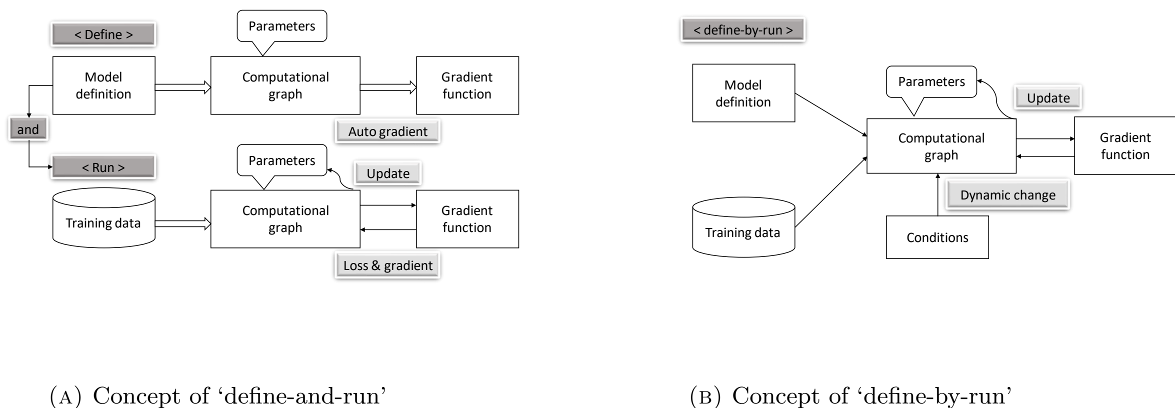
Basically the hybrid approaches use the output of one algorithm to start the second algorithm. Since the information is shared among concurrent searches in Autotune, the strength of this approach can be improved over other the hybrid approaches. More research is required to enable the effective handling of early-stopping of unpromising trials. Also, there are rooms for improvement regarding the mechanism of Bayesian search method and the support for multi-objective tuning with exploration of trade-offs between model complexity and model quality [59].

2.4.2 Static vs. dynamic search space

In DL frameworks, the term **'define-and-run'** refers to the deployment of training models in which the computational graph is built at first, then the model is trained using training data. It leads to inefficient usage of memory because all layers of the NN that comprise of the computational graph must be loaded in memory prior training, even though some layers might be required only at the starting or ending of the training process [88]. For example, this scheme could be easily implemented for static NN models, like CNNs, but this is not useful for RNNs. In RNNs, it is generally trained using backpropagation through time and applying thresholds for the propagation for achieving computational efficiency. The entire computational graph still must remain in the memory even if it does not matter whether certain layers are no longer necessary in backpropagation [68].

This scheme does not allow the user to access the inner workings of the NNs, thus, leading to difficulties while creating an effective model; the user would need to access the inner details for debugging and tuning a model efficiently. However, the computational graph in this technique contains information about the model as a whole like a black-box; therefore, the profiler and debugger would not be able to determine the errors and the required solutions [88].

The term **'define-by-run'** [88] means that, rather than defining in advance, the user is able to construct the search space dynamically during runtime. In this technique, the model's computational graph is not fixed before training. Rather, when the forward computation for the training data is started, the computational graph is initialized accordingly, leading to efficient usage of memory and enough user-flexibility for debugging. These two techniques are shown in Figure 2.4 (based on Tokui *et al.* [88]).



(A) Concept of 'define-and-run' (B) Concept of 'define-by-run'

FIGURE 2.4: Hyperparameter optimization ('define-and-run' vs. 'define-by-run') (based on Tokui *et al.* [88])

Many popular HPO frameworks, such as **HyperOpt** [15], **Spearmint** [83], Autotune [49], Katib [104],

Tune [57], GPyOpt,[2] Sequential Model-based Algorithm Configuration (SMAC) [40], Vizier [31] etc. follow 'define-and-run' style for constructing the hyperparameter search space. They construct the search space at first and then start the computations.

Chainer [88] is an open-source distributed framework for DL models. Existing optimization methods are supported by Chainer, such as AdaGrad [90], Stochastic Gradient Descent (SGD) [64], Adam [102], RMSprop [106]. Chainer does not fix a model's computational graph before the model is trained. When the forward computation for the training data set takes place, the computational graph or the network is defined dynamically [88]. Using the 'define-by-run' approach, Chainer defines complex NNs with user-flexibility and allow them to make modifications during runtime. Therefore, researchers and engineers can easily work on complicated models through trials-and-errors [88].

Optuna presents the 'define-by-run' [88] API, enabling users to construct the search space dynamically. In this process, the user does not have to explicitly define everything about the search space and optimization before execution. Rather, Optuna receives a living trial object when the *'optimize_API'* is invoked on the objective function and through the interaction with that trial object, the objective function is built gradually. During the runtime of the objective function, the methods of the trial object construct the search spaces dynamically [2].

2.4.3 Early-stopping and pruning

The use of a multi-armed bandit strategy [89] [89] has shown promising results for DNN models; it can help in optimizing the hyperparameters as well as the training budget. Therefore, this approach has been considered in the design procedure of **Hyperband** [56]. For optimizing hyperparameters, Hyperband has implemented Random Search through adaptive resource allocation and early-stopping [26]. They have addressed how to assign resources among arbitrarily sampled hyperparameter configurations.

Furthermore, Hyperband counts on an early-stopping strategy to distribute resources and assess orders-of-magnitude more configurations than general black-box techniques like BO methods, feature subsets and iterative algorithms. The finite computational resources (e.g. training epochs or dataset size, data samples, iteration, etc.) are dynamically allocated in Hyperband through random sampling and under-performing hyperparameter settings are eliminated by implementing Successive Halving (SH) [74].

Jamieson and Talwaker [55] have taken into consideration the fact that a huge number of configurations must be evaluated for exploring the large hyperparameter spaces efficiently. Since the costs and time of training models keeps growing in complex ML models, authors in this work have introduced **Asynchronous**

[2] `https://github.com/SheffieldML/GPyOpt`

Successive Halving Algorithm (ASHA), a bandit-based partial training method and a robust hyperparameter tuning algorithm which exploits parallelism and aggressive early-stopping as already discussed earlier.

Google Vizier [31], provides black-box hyperparameter optimization. Automated early stopping is supported in Vizier via an API call to a method 'ShouldTrialStop'. Based on the early-stopping rules implemented in Vizier, this API analyzes and determines the trials that should be stopped. Vizier follows the median-stopping rule, where a pending trial is stopped at a certain time if the trial's best objective value by that time is worse compared to the median value of the completed trials' objective values by that time [31].

2.4.4 Lightweight and ease-of-use

The updated version of **Waikato Environment for Knowledge Analysis (WEKA)** project named **Auto-WEKA** [87] is particularly devised for helping users by exploring through WEKA's ML algorithms using the BO technique. Later, Kotthoff *et al.* [50] have updated the software and added four major features, which made the usage of WEKA easy-to-use and lightweight.

1. previously it had support only for classification algorithms, now it supports regression algorithms,
2. all performance metrics of WEKA is getting support for the optimization from 2.0,
3. parallel executions (on a single machine) is now supported natively; moreover, instead of just the single best configuration, the N best configurations of each run are being saved by the framework, and
4. Auto-WEKA 2.0 is now completely integrated with WEKA.

SMAC, GPyOpt, Spearmint, Hyperopt and Optuna provide the convenience of easy setup and installation for the frameworks. Also, they provide the facility to conduct the computations in a user-friendly lightweight way through their interfaces. In Optuna, the general users can run their experiments in their own local machine with Jupyter Notebook;[3] and instead of dealing with the extra effort for deploying a database by themselves, they can rather use Optuna's in-memory based built-in data-structure as the backend storage.

2.5 Optuna

Optuna [2] is an open-source hyperparameter optimization toolkit, which is particularly designed for ML and DL applications. As long as the objective function can be defined, it can operate on non-ML applications also. It has simplified architectural design and other flexible features, which are the reasons why it has been chosen

[3] `https://jupyter.org/`

as the HPO for the designated Flower Counter application in this research work. Many existing optimization frameworks are not appropriately addressing some major issues. Therefore, they have incorporated these important features in Optuna.

- A **'define-by-run'** style to dynamically construct the search space of different hyperparameters was deployed.
- **Efficient strategies for searching and pruning** for making better cost-effectiveness of optimization has been incorporated in it.
- The architecture is **scalable** for handling any size of experiment. It can be configured in a distributed environment for parallel optimization.
- It is very **easy to configure** and installable with the lowest setup requirements (e.g., via using minimal commands).
- It is available as an **open-source** software and provides the support to incorporate new ideas and methods of optimization from the community, etc.

The HPO process in Optuna basically considers a specific target of maximizing/minimizing an objective function as the input using the given range of hyperparameters and provides the (validation) score as the output. They use the term **'study'** to indicate each process of optimization and **'trial'** to indicate each assessment of objective function. Optuna provides samplers for hyperparameter sampling. For each trial, the user is asked to call **'suggest_API'** sampler within the objective function, e.g., a method called '*suggest_int*' suggests a number for '*n_layers*'; an integer number to decide the number of hidden layers in the Multilayer Perceptron (MLP) [86] or a '*suggest_uniform()*' for suggesting a continuous value for momentum [6] for NN training, etc. Heterogeneous parameter spaces can be described using simple code in Optuna.

Optuna enables HPO by adopting efficient sampling and pruning techniques for unpromising trials. The search space is continually narrowed down by the samplers via using the suggested parameter values based on the records or previous history and evaluated objective function's values. This could lead to stable search space, better values of objective function and better hyperparameter values. The following sampling algorithms are applied by Optuna, they can be accessed under the *'BaseSampler'* class.

- SMBO based GP and TPE sampler,
- CMA-ES based, sampler,
- Random Search,
- Grid Search, and
- User-defined algorithm

Unless specified otherwise by user, the default sampler is *'optuna.samplers.TPESampler'*. [4] Furthermore, Optuna also permits users to use their own personalized sampling procedure.

For the pruning mechanism, there are two phases:

1. monitoring the intermediate objective values periodically and
2. terminating the trial which does not encounter the pre-defined conditions for the objective function.

As already stated previously, the following pruning algorithms are available in Optuna:

- ASHA,
- Hyperband,
- Median pruning algorithm [37], and
- Threshold pruning algorithm [105].

Optuna offers early-stopping and pruning strategies by implementing various APIs under the *'pruner()'* module. It uses the *'report()'* API to monitor the functionality and the *'should_prune()'* API to terminate the unpromising trials prematurely. A boolean value is returned by this module for a given trial and its associated study; the value represents if the trial should be pruned or not. There is a *'BasePruner'* class as the parent pruner class, from which the remaining child classes inherit from for implementing different pruning strategies. Different pruning strategies are offered by this module, such as pruning based on any particular threshold value or median value, any specified percentile (25%, 50%, 75% etc.), pruner using ASHA and Hyperband etc [2]. ASHA is mostly advantageous in distributed applications since at each round of the pruning, no worker waits for the results from further workers. So, parallel computation is achieved as there is a minimal delay during multiple trials.

Optuna is scalable from a single machine to multiple machines. It offers a built-in in-memory data structure as the storage back-end for small scale or personal use. Any Relational Database (RDB) can be defined using *'storage_name'* for working with large-scale datasets and parallel optimization. Databases such as SQLite[5] can be employed as the backend as well (for distributed experiments, they suggest using other RDBs instead of SQLite, such as PostgreSQL [71] or MySQL for better performance). An RDB backend enables access to the history of studies and maintains the persistence of experiments (i.e., facilitates the users to save and resume the studies). Optuna provides a web dashboard for the purpose of visualization and analysis of studies in real-time. Optuna can easily be merged into a container-based system such as Kubernetes. The flexible system design lets the users conduct distributed computations which scales almost linearly with the increasing number of workers.

[4] `https://optuna.readthedocs.io/`
[5] `https://www.sqlite.org/`

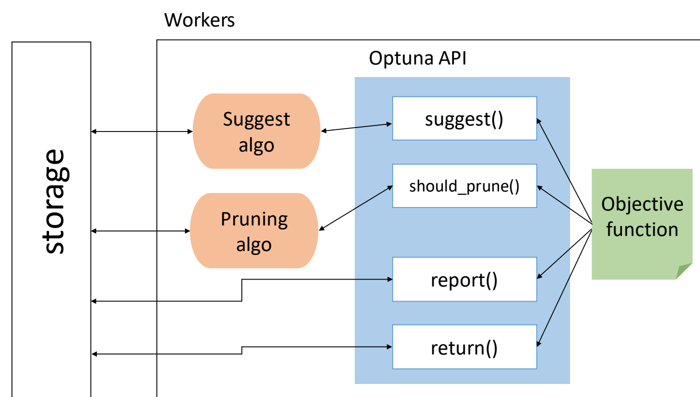


FIGURE 2.5: Architectural overview of Optuna (based on Akiba *et al.* [2])

The architectural diagram of Optuna is shown in Figure 2.5 (Akiba *et al.* [2]). Every worker runs an instance of the objective function within each study. Using the Optuna APIs, the model executes a trial and computes a value for the objective function. The shared storage is accessed by the objective function upon the invocation of the associated APIs. Necessary information from the previous studies can be easily obtained from the storage at anytime. The model runs trials on the objective function using each worker independently and the progress of the current study is shared among them via the storage.

2.6 Review of the HPO frameworks

Some well-known HPO frameworks and DL libraries have been discussed so far. Most of the HPO follow the 'define-and-run' strategy, except Optuna [2] and Chainer [88] follow 'define-by-run' scheme to facilitate dynamic and flexible computational strategy. Optuna, Tune-on-Ray, Hyperopt, Sckit-Optimize,[6] etc. are open-source projects but Google Vizier is a paid service (since it provides cloud storage). Optuna, Tune-on-Ray, Hyperopt and Google Vizier can provide parallelization, use GPUs and handle large datasets but Sckit-Optimize can not. Autotune, Google Vizier, Katib, Tune-on-Ray provide pruning mechanisms, facilitate real-time dashboard for visualizing the experiments and results but being distributed systems in nature, the configuration process could be difficult for general users. Other frameworks such as SMAC, GPyOpt, Spearmint and Hyperopt do not offer a pruning mechanism or a dashboard, but they are lightweight and easy to setup and install.

Optuna almost fulfills all of the features of HPO that have been mentioned above; it offers an efficient pruning mechanism, supports real-time dashboard via Tensorboard [94], can operate both on a single machine or distributed machine and just takes few lines of instructions to install and setup, therefore, making it very simple for general users to easily configure it and use it for their applications. Therefore, based on the

[6] `https://scikit-optimize.github.io/stable/`

overall easiness and technology compatibility, and other useful characteristics, Optuna has been chosen as the designated HPO for this work.

2.7 Summary

In this chapter, essential terminology used in CNN which will be mentioned recurrently in the upcoming chapters have been explained. A brief idea about the DL framework Tensorflow's execution model has been presented that has been used for Flower Counter application. Next, the concepts of HPO and their features, various techniques and algorithms used for searching among the best observed hyperparameter values in HPO are explained. The popular automated HPO frameworks have been categorized according to some fundamental characteristics of HPO methods and their working mechanisms have been discussed in detail correspondingly. Then, the architectural description and the significant characteristics of HPO framework Optuna, which has been selected as the HPO for optimizing the hyperparameters of the designated Flower Counter application, has been described comprehensively along with the reasons of its preference compared with the other frameworks.

3 Experimental Design and Configuration

This chapter begins with a brief description of the architecture of the CNN-based Flower Counter application [19], which is being used as the case study in this work that has been developed in DISCUS Lab. Also, a detailed description and pictorial depiction of the datasets being used for the experiments, the process of generating the ground truths and the configuration details are presented.

3.1 Architecture of the Flower Counter application

3.1.1 Overview of MultiColumn CNN-based (MCNN) Flower Counter model's architecture

The Multi-column Convolutional Neural Network (MCNN) model for estimating crowd count by calculating the density map of a given crowded image was proposed by Zhang *et al.* [100] [101] which inspired the architecture of the CNN based model of the Flower Counter application [19]. The MCNN model of Flower counting is shown in Figure 3.1. The diagram is rotated 90 degrees and it turned the columns into rows just pictorially, conceptually they are the columns. The structure follows this columnar pattern in the CNN: (CONV-POOL- POOL-CONV-DECONV-DECONV). The last layer of the columns stack feature maps from the previous layers and passes it on to a final CONV layer which generates the density map. The loss function is calculated between the ground truth and the predicted density map. The twofold usage of POOL layers reduces the spatial resolution of each image by 1/4 as well as the output resolution of the predicted density map; thus, to prevent this down-sampling and loss of useful information two Deconvolutional (DECONV) layers have been inserted. A deconvolution is a mathematical operation that is the inverse of the convolution function in order to reverse the effect of convolution. The purpose of deconvolutional layer is to find signals which are lost or features that might not have been considered significant previously CNN's task. Due to having been convoluted with other signals, a signal might be lost. The architectural diagram is taken directly from the original work [19] and shown in Figure 3.1.

3.1.2 Datasets

The datasets being used are the images of Canola flowers from various cameras which were positioned in distinct locations of various Canola flower fields [19] as stated in Chapter 1. They were collected as a part of the COAST project. Some filtering was done explicitly by visual inspection to exclude images captured

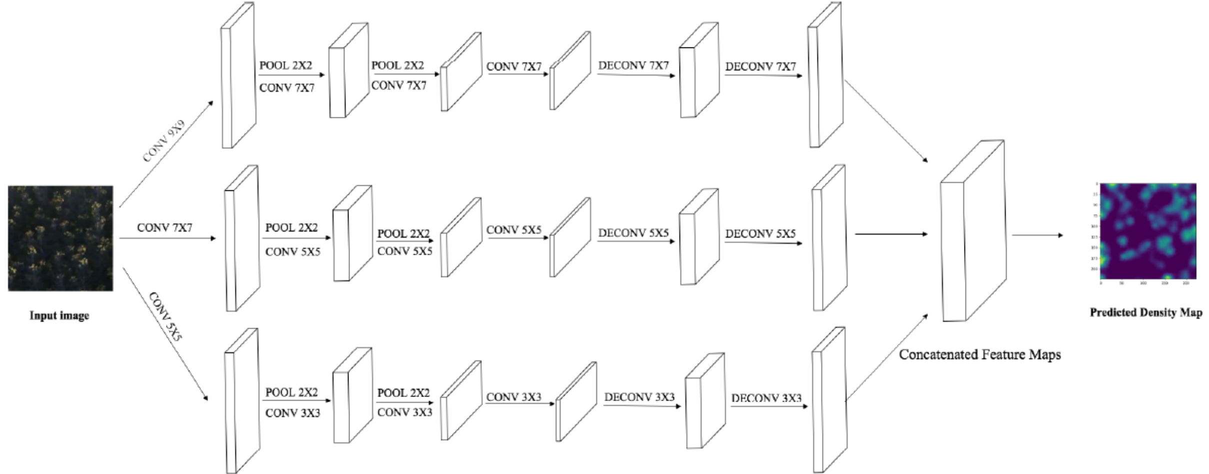


Figure 3.1: Multi-column Convolutional Neural Network architecture of Flower Counter application [19]

during bad weather conditions, such as the days when there was fog or it was very dark or windy, cameras that had extremely out of focus pictures etc. After the filtering, the images were tiled into small sub-images for getting better visuals of the flowers. Then, those sub-images were annotated by the researchers for generating the ground truth and used for the training the model [19].

Throughout the rest of this document, the image capture date/camera ID for all 2016 images will be referred to using the format: 'camera-day', where camera represents the camera id and the day presents the date of data collection. The date is formatted as mmdd, where the mm stands for month and the dd stands for day. For example, the images captured on the second of August by camera 1109 will be represented as 1109-0802. For 2018 dataset, the camera id is represented as: 'canola40-mmdd-split'. For example, the images captured on the 14th of July will be represented as canola40-0714-split.

3.1.3 Generation of ground truths and density maps

To match the software expectation of image resolution, each filtered image was split up into 224x224 sized tiles. To split the images in parallel, a python script was developed which performed in such a way that, when given the single coordinate for each image, it would automatically crop a variable sized window and then split into 224x224 sized tiles.

In order to train the Flower Counting application, the ground truth density maps were required, which were prepared by using the coordinates of the flowers. The procedure of generating density map was like this: a) all the visible flowers from each image was located and manually annotated by putting a red marker into the center of each observable flower with GIMP[1] (the annotations were verified by the supervisors), and then, b) a python program was executed to extract the coordinates of each flower. A two-dimensional

[1] `https://www.gimp.org/`

(2D) empty array was created with the same height and width of input image tile. The value 255 is put in the same locations inside the two-dimensional array as in the annotated images (generated from manual annotation). Finally, a Gaussian kernel [82] normalized to sum to one was used that blurred the non-zero elements of the array. Some sample images and their corresponding density maps for 2016 and 2018 datasets are demonstrated in Figure 3.2 and 3.3 respectively.

3.2 Hardware and software configurations

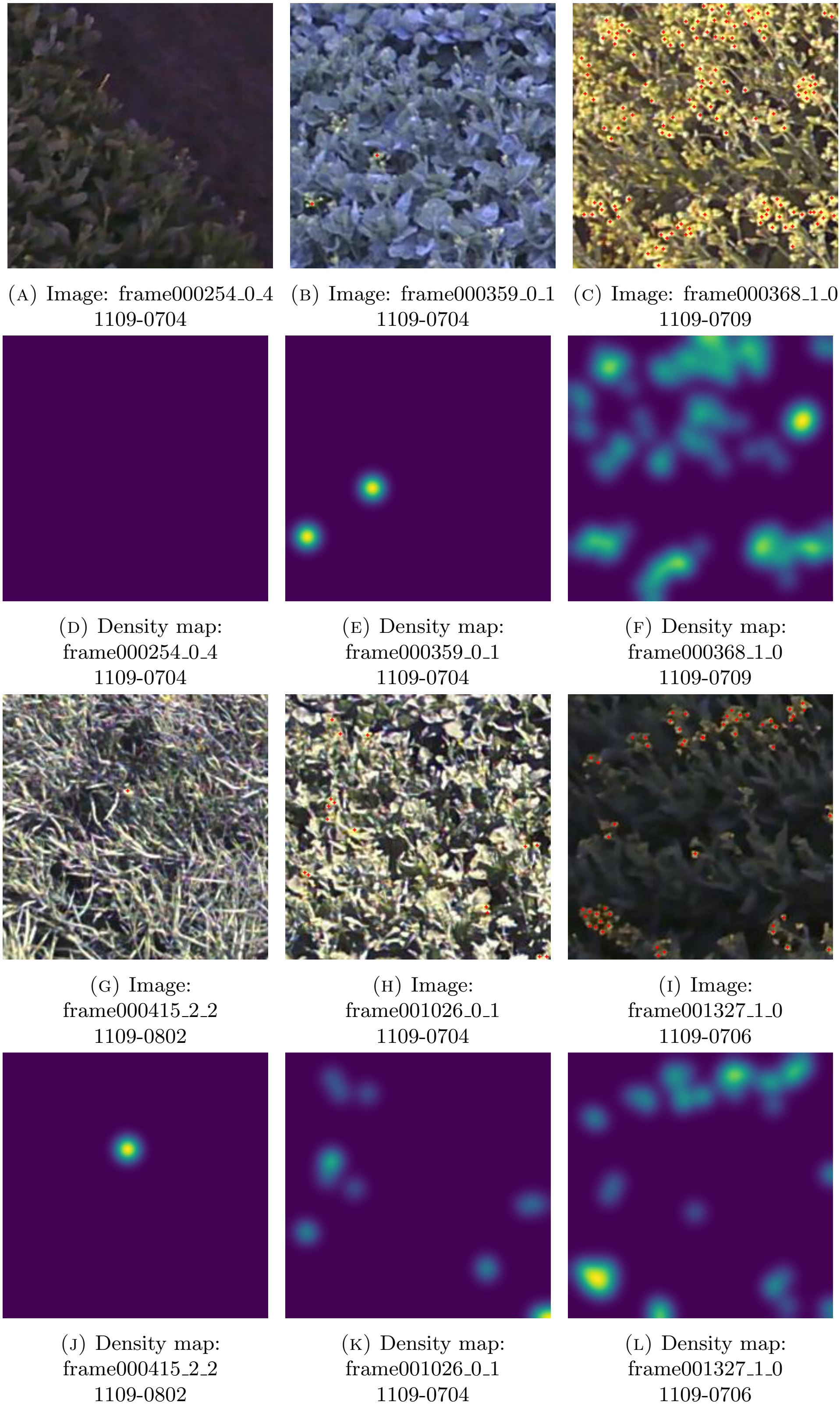
The experiments have been run on four Ubuntu servers. All of them have the same hardware configurations with Ubuntu Linux version 18.04.5 LTS, 64 bit OS, RAM size 1.4 TB, NVIDIA Tesla V100 PCIe 16 GB graphics card. Other software and hardware configurations are listed below in Table 3.1.

TABLE 3.1: Configuration details of the servers used in the experiments

Configuration Parameters	**Characteristics**
Architecture	x86_64
CPU(s)	64
Model name	Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
System memory size	1.4 TB
GPU configurations	Product Name: TeslaV100-PCIE-16GB NVIDIA-SMI Driver Version: 418.181.07 CUDA Version: 10.1
Software versions inside Anaconda	matplotlib: 3.2.2 numpy: 1.18.5 optuna: 1.5.0 pandas: 1.0.5 pip: 20.0.2 python: 3.6.10 seaborn: 0.10.0 tensorboard: 2.0.2 tensorflow: 1.14.0 tensorflow-gpu: 1.14.0

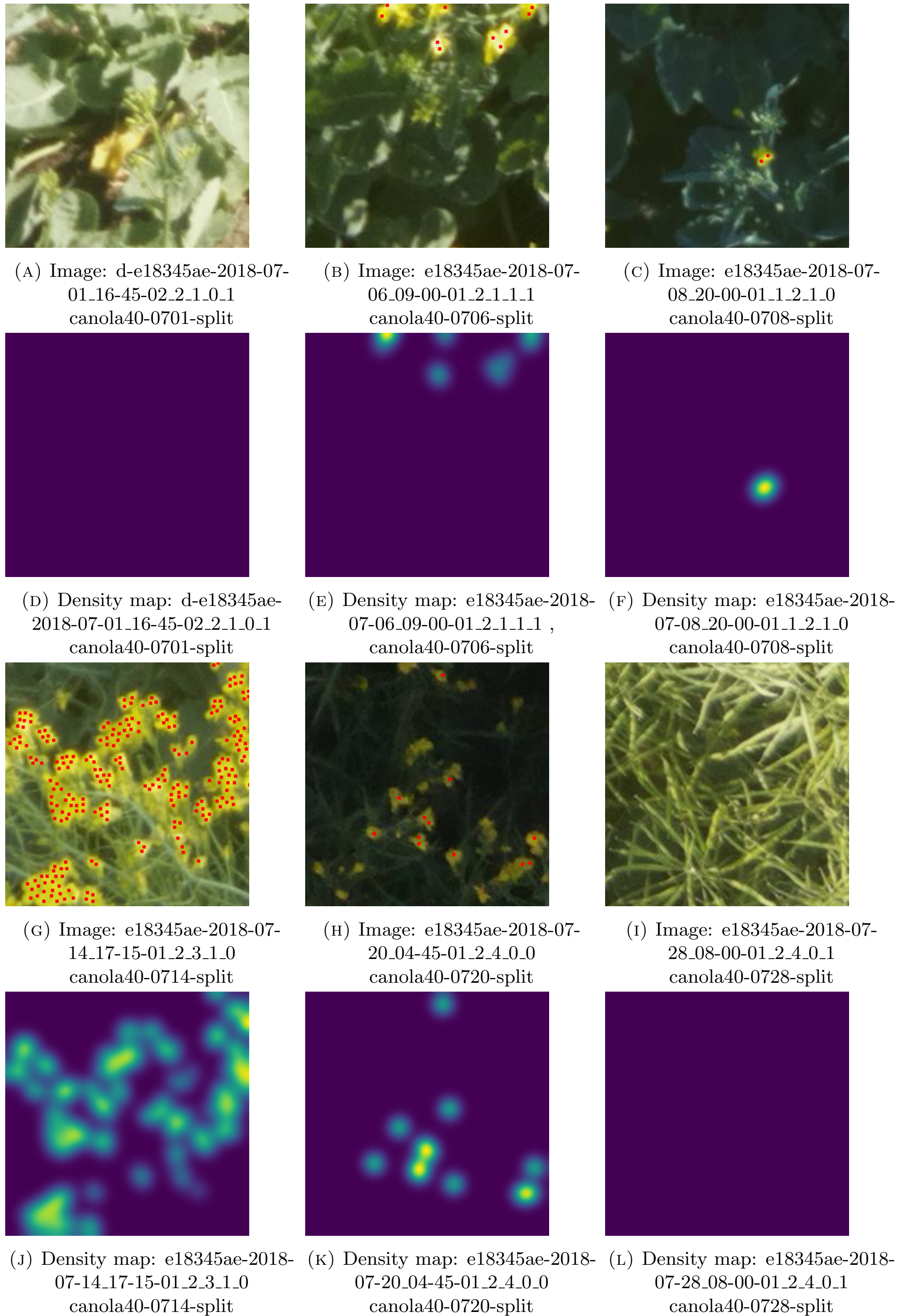
3.3 Dataset characterization

Based on the existing variations of the images of the datasets and image qualities, four datasets have been chosen as representatives in this work to train, validate and test the necessary experiments on the flower



(A) Image: frame000254_0_4 1109-0704
(B) Image: frame000359_0_1 1109-0704
(C) Image: frame000368_1_0 1109-0709
(D) Density map: frame000254_0_4 1109-0704
(E) Density map: frame000359_0_1 1109-0704
(F) Density map: frame000368_1_0 1109-0709
(G) Image: frame000415_2_2 1109-0802
(H) Image: frame001026_0_1 1109-0704
(I) Image: frame001327_1_0 1109-0706
(J) Density map: frame000415_2_2 1109-0802
(K) Density map: frame001026_0_1 1109-0704
(L) Density map: frame001327_1_0 1109-0706

FIGURE 3.2: Sample images from different camera-days of 2016-all dataset and corresponding density maps



(A) Image: d-e18345ae-2018-07-01_16-45-02_2_1_0_1 canola40-0701-split

(B) Image: e18345ae-2018-07-06_09-00-01_2_1_1_1 canola40-0706-split

(C) Image: e18345ae-2018-07-08_20-00-01_1_2_1_0 canola40-0708-split

(D) Density map: d-e18345ae-2018-07-01_16-45-02_2_1_0_1 canola40-0701-split

(E) Density map: e18345ae-2018-07-06_09-00-01_2_1_1_1 , canola40-0706-split

(F) Density map: e18345ae-2018-07-08_20-00-01_1_2_1_0 canola40-0708-split

(G) Image: e18345ae-2018-07-14_17-15-01_2_3_1_0 canola40-0714-split

(H) Image: e18345ae-2018-07-20_04-45-01_2_4_0_0 canola40-0720-split

(I) Image: e18345ae-2018-07-28_08-00-01_2_4_0_1 canola40-0728-split

(J) Density map: e18345ae-2018-07-14_17-15-01_2_3_1_0 canola40-0714-split

(K) Density map: e18345ae-2018-07-20_04-45-01_2_4_0_0 canola40-0720-split

(L) Density map: e18345ae-2018-07-28_08-00-01_2_4_0_1 canola40-0728-split

FIGURE 3.3: Sample images from different camera-days of 2018 dataset and corresponding density maps

counter model. They are: 2016-all dataset, 2018-split dataset, 2016-1109-5-12 dataset and 2018-july-2-23 dataset.

The images of the 2016-all dataset were captured 1x per minute, usually only in daylight, but some cameras took pictures 24 hours a day. The resolution of the images are 1280x768. Images from cameras were retrieved every 14 days. The timestamps are on the image itself, and follows an ordinal filename for the image within the day. Several of the cameras had unusable pictures which were very out of focus. There are 6 cameras under 2016 camera: 1108, 1109, 1122, 1207, 1225, and 1237. The images of camera 1108 are very fuzzy, and thus, unusable until the date of July 15. Camera 1109 is good the whole time, but is missing for the middle two weeks, because the camera was not functioning. Camera 1207 is blurry again; images are sampled from 1225 and 1237.

The images of 2017 dataset were captured 1x every 5 minutes. Clocks were broken, so there are no reliable timestamp. There were approximately 5-6 cameras taking pictures of Canola; among them one camera's images are obtained which gave better visibility. The resolution of 2017 camera is 3280x2464.

The images of 2018 dataset were captured 1x every 15 minutes. There are timestamp in the filename, so it is easy to distinguish the cameradays. Images from the cameras were retrieved at end of season. There were probably 5 or 6 cameras that had data; among them, data from 3 cameras with better quality are retrieved. The resolution of the images are the same as 2017 images and the cameras were of better quality.

The 2016-all dataset and the 2018-split dataset have lots of images and so many images with zero flowers relative to the total number of images. So, it would take several hours/days to complete a single trial for these huge datasets and also those zero-flower images would impact the learning of the model significantly (the training of the model would be biased towards zero flowers while learning).

Hence, some pre-experiments have been conducted to decide upon a suitable binning strategy for facilitating suitable learning and training of the CNN model. This strategy has been explained, used and its accuracy has already been proved in the chowdhury's work [19]. Briefly, this strategy can be described as follows: in order to select sample images to reduce the bias towards sparse images, a fixed number of bins are loaded with certain number of images that fit the criteria. An equal number of images from each bin are selected based on the smallest bin. If it is not binned, then the model would be biased and lead towards the problem of 'overfitting', meaning memorizing everything and not being to able generalize well on unseen data. So the training and the validation of the model would be greatly affected, such as it would provide very good accuracy scores on the similar test populations but poorer scores on unseen test polulations. This strategy of under-sampling the majority group and over-sampling the minority group reduces the total number of training images; it also reduces overall model bias [69].

Therefore, based on pre-experiments, nine bins have been selected for this work to be loaded with the particular number of flowers respectively (shown in Table 3.2) for the large datasets. The trade-off between the equal ranges of flowers and equal numbers of images in each bin as shown in Table 3.2 can be explained as follows: 1) the size of the smallest bin has been kept as large as possible to get a larger number of samples,

2) the number of samples within each bin have been fairly selected, and 3) it has been ensured that there was an adequate amount of diversity. As an example, if 0-9 was chosen as a bin, there would have been far more images with zero flowers than that of 9's. The distribution would have looked like this (42196, 7156, 2808, 2522, 567) for 0-9 flowers. But between 25-29 flowers, the distribution would have looked like this (76, 64, 59, 66, 57) where the number of images were more evenly distributed. There are 42196 images with 0 flowers. A bin which is this large, would underrepresent the image with 9 flowers. For the 2016-1109-5-12 dataset and the 2018-july-2-23 dataset, the whole datasets are used as the total number of images are comparatively fewer in number already. The bins have been loaded with certain number of flowers to include enough flowers in each bin according to the availability of the annotated images at hand. Based on the applied binning strategy, the total number of flowers in each bin for training-validation-testing (the ratio that is followed is 70-20-10(%)) are shown in Table 3.3. They show the number of images in each bin according to the number of flowers as per Table 3.2 and finally, the number of samples that would be selected according to the smallest number among all the bins for different datasets respectively.

Table 3.2: Binning strategy being used for the 2016-all dataset and the 2018-split dataset

Bin index	**Flower count range**
1	0-4
2	5-9
3	10-14
4	15-19
5	20-24
6	25-29
7	30-39
8	40-69
9	70 and above

A histogram of the annotated images used for model training from these 4 datasets is shown in Figure 3.4; it shows the distribution of the number of flowers in each bin. The distributions are skewed to the right end because more images contain lower numbers of flowers. The reason is the choice of sparse images by individuals while doing manual annotation which takes less efforts and scarcity of images with larger numbers of flowers available at hand.

Figure 3.4 shows the distributions of training datasets: the distributions shown in Figure 3.4a and Figure 3.4b are the distributions for the 2016-all dataset and the 2018-split dataset respectively before binning, and 3.4c and 3.4d are are the distributions for the 2016-1109-5-12 dataset and the 2018-july-2-23 dataset respectively without any binning. In the 2016-all dataset, for 4 days (0802, 0803, 0805, 0806 cameradays), images were annotated at the end of the season with nearly all zero flowers since blooming was finished. This is a limitation, because this was the camera with the clearest pictures, but it was not functional between

Table 3.3: Total number of images (before and after binning) used for training for datasets

Dataset	**Number of Images (train-validation-test ratio: 70%-20%-10%)**
2016-all dataset	*Before binning*: Total number of images = 76,576
	After binning: 53,603 + 15,315 + 7,658 = 76,576 Images per Bin: [66502, 4202, 1388, 765, 472, 325, 374, 909, 1639] Number of samples to be collected from each bin: 325
2018-split dataset	*Before binning*: 2924
	After binning: 2,047 + 585 + 292 = 2924 Images per Bin: [1254, 473, 318, 140, 142, 140, 186, 155, 116] Number of samples to be collected from each bin: 116
2016-1109-5-12 dataset	*Without binning*: 3720 + 1063 + 532 = 5315
2018-july-2-23 dataset	*Without binning*: 1296 + 370 + 186 = 1852

July 13 and August 1.

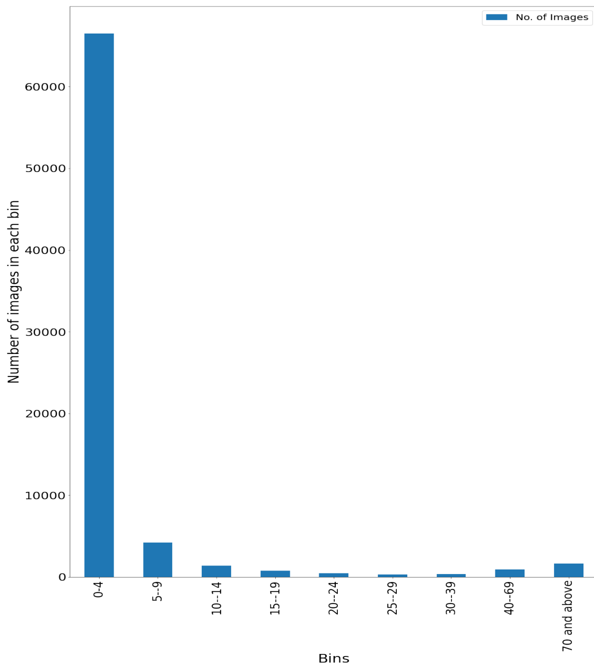
Figure 3.5 shows the distributions of testing only datasets; the distributions shown in Figure 3.5a and Figure 3.5b are the distributions for 2016 camera-1237 dataset and the 2018-split canola40-0706 dataset respectively, and 3.5c and 3.5d are are the distributions for the 2016-other-cameras dataset and the 2018-july-20-28 dataset respectively. The above mentioned four datasets have following number of images as shown in Table 3.4.

Table 3.4: Training and test datasets for different population experiments

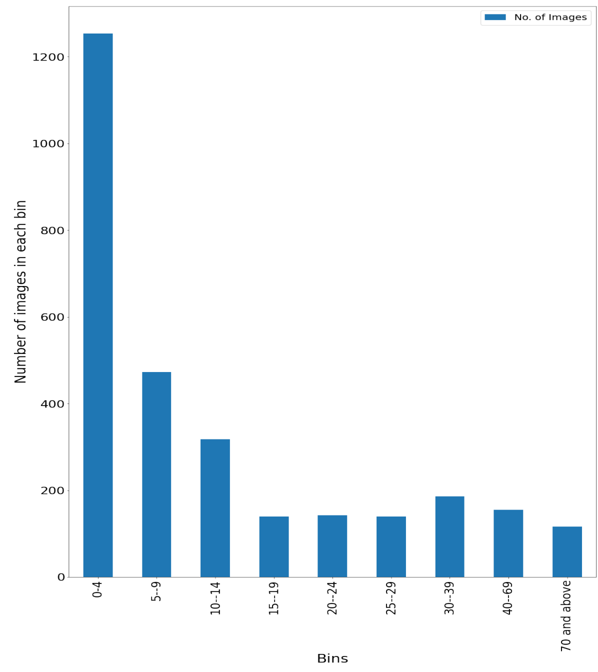
Training dataset	**Test dataset**	**Number of Images in Test datasets**
2016-all dataset	2016 camera-1237	354
2016-1109-5-12 dataset	2016-other-cameras dataset	562
2018-split dataset	2018-split canola40-0706	124
2018-july-2-23 dataset	2018-july-20-28 dataset	864

Four training datasets and four test datasets have been taken from two years 2016 and 2018. The camera-days of the test datasets for both the 2016-all dataset and the 2018-split dataset, are also part of the population of training datasets. Therefore, a bash script has been run to exclude the test images from the training datasets and prepare a separate test dataset. The test datasets are similar to the training datasets, but not exactly the same; rather, they are taken from a different camera and different day in order to check the overfitting problem [96] of the model. The model has been trained-tested on different datasets respectively. Also, the number of test populations in each test dataset are mentioned in Table 3.4.

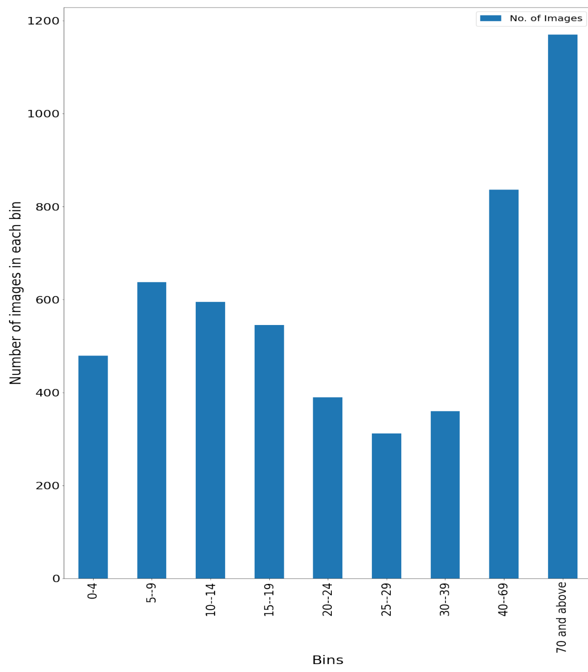
The datasets have images captured in various weather conditions which might significantly impact the



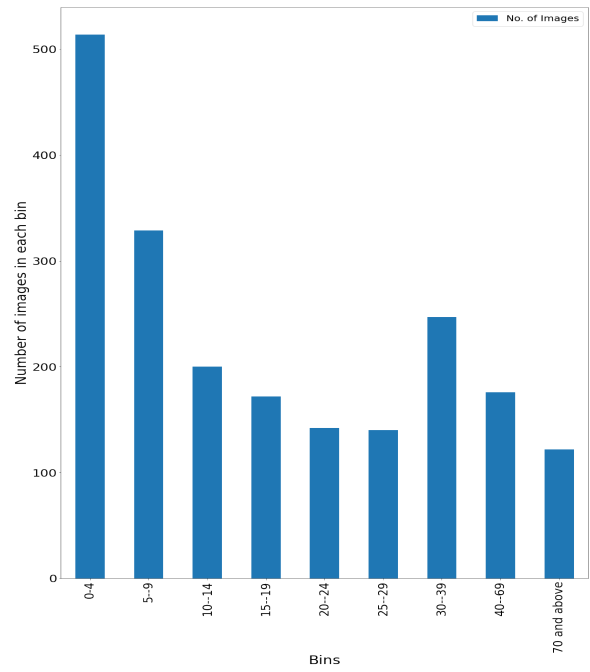
(A) 2016-all dataset



(B) 2018-split dataset

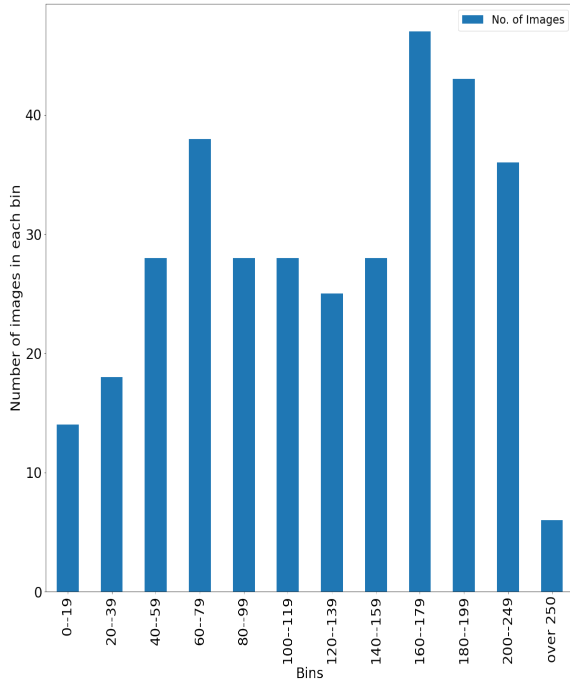


(C) 2016-1109-5-12 dataset

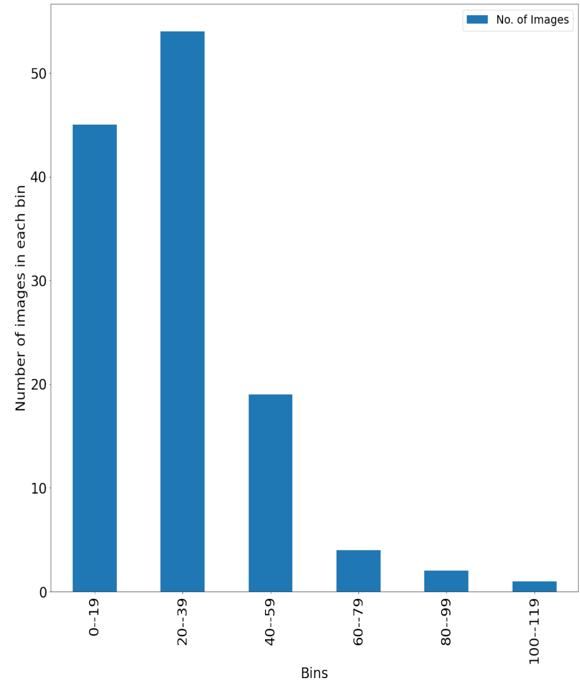


(D) 2018-july-2-23 dataset

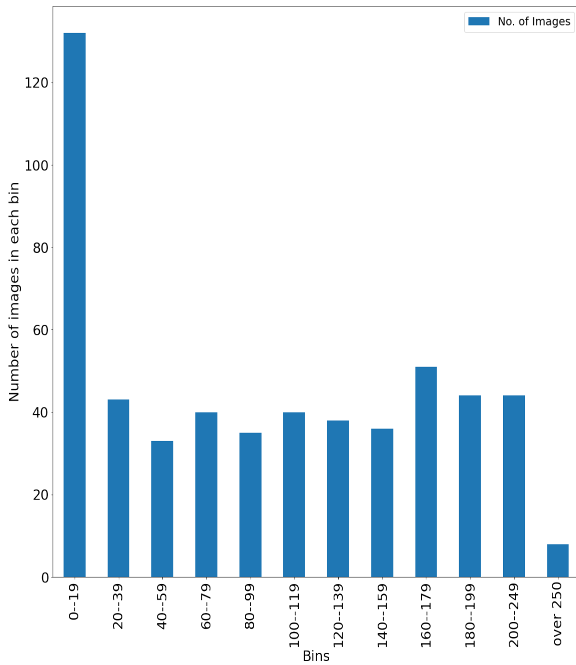
Figure 3.4: Density histograms of training datasets



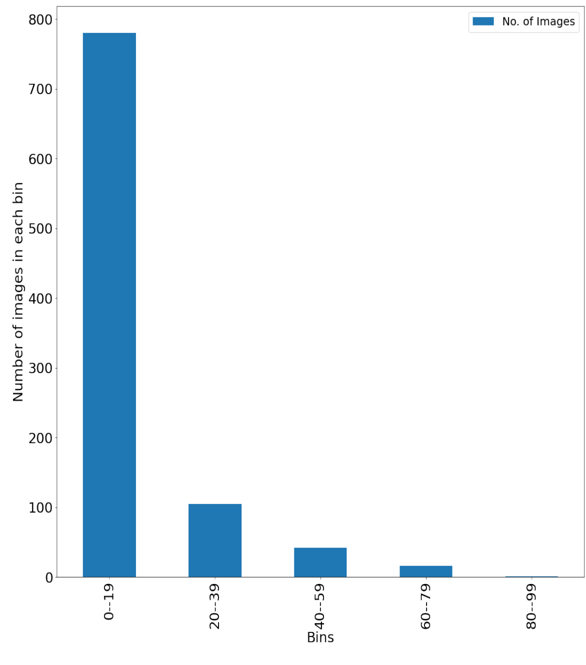
(A) 2016-camera-1237 dataset



(B) 2018-split canola40-0706 dataset



(C) 2016-other-cameras dataset



(D) 2018-july-20-28 dataset

Figure 3.5: Density histograms of testing datasets

learning of the model. In the 2016-all dataset, in the early morning, it was foggy some days or some days it was windy; so those images were not annotated. In the mid-days, it was sunny and thus, the visibility was comparatively better. In some images, there were just pods with glare on the leaves and pods. The datasets have both sparse and dense images (depending on the user's choice for any threshold, 'k', images could be considered as sparse or dense). Some images just did not have any flower except leaves or pods. The dark images are either from very early in the morning or evening, so giving poorer visibility etc.

Comparatively, the 2018 dataset has better camera resolution than the 2016-all dataset but with similar types of weather conditions as mentioned. All these variabilities certainly provided the model with a wider range of characteristics while training and testing and indeed, this could have a significant impact during training-testing on all the datasets.

3.4 Experimental design and methodology

In the beginning of this section, the accuracy metrics that are used in this work are explained, then the general concept of a boxplot distribution is explained for understanding the experiments conducted in Chapter 4. Later, the configurations and range of values selected for different experiments and their associated reasoning are described.

3.4.1 Accuracy metrics used in Flower Counter

The following accuracy metrics have been used in the testing and training phases of the Flower Counter application [19]. The Loss metric is used in both training and testing and is pixel/channel-based, while the other three metrics are based on total number of predicted flowers and are only used in testing.

Loss: The loss function used for training the model in Flower Counter model is the average pixel-wise loss based on the Euclidean distance metric between the density map values from the manually annotated images and the predicted density maps as stated in Chowdhury's work [19].

Mean Absolute Error (MAE): This metric calculates the error difference between the predicted value and the actual value and takes the absolute of the difference; therefore, the arithmetic mean of all the absolute prediction error values is the Mean Absolute Error (MAE) [81]. Mathematically, it can be computed as follows.

$$MAE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}, \tag{3.1}$$

where n is the total number of observations, y_i is the actual value for ith observation and $\hat{y}_i$ is the predicted value. So, for the Flower Counter model, n would be the total number of images in the test datasets, y_i would be the number of flowers in the original density maps and $\hat{y}_i$ would be the number of flowers in the predicted density maps.

Mean Relative Error (MRE): The metric Mean Relative Error (MRE) measures the uncertainty of measurement in percentage (%) [81] compared to the size of the measurement. For instance, if the total length

of a measurement is 15 cm, then an error of 1 cm would be a lot, but if the length was 15 km, it would be just insignificant. Mathematically, the absolute error of each instance is first divided by its corresponding actual value and then the arithmetic mean of all the instances of the test dataset is calculated. Mathematically, it is computed as follows:

$$MRE = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{y_i}. \tag{3.2}$$

An important note regarding the calculation of MRE metric is that, if the original flower count is zero, then the value of the predicted count itself is considered as the MRE (as seen from the equation above, the denominator is the predicted count and thus, it cannot be divided by zero) in the Flower Counter application. So, if there are zero flowers but the predicted count is 10 then the MRE is 10% and so on.

Root Mean Squared Error (RMSE): The Root Mean Squared Error (RMSE) takes the square root of the mean of the square of all differences of actual and predicted values [81]. In RMSE, the errors are squared and then they are averaged. When large errors are particularly undesirable, this criteria becomes more useful to detect them. The mathematical formulae is shown below.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}. \tag{3.3}$$

3.4.2 Measurements/visualization

Figure 3.6 shows a general boxplot which is a standard statistical method of summarizing data distribution using five numbers: 'minimum' value MIN, first quartile ($Q1$) (25th percentile), median value, third quartile ($Q3$) (75th percentile) and 'maximum' value MAX [66]. The difference between first and third quartile is called 'Interquartile range (IQR)'; the MIN is $(Q1 - (1.5 \times IQR))$ and the MAX is $(Q3 + (1.5 \times IQR))$. The boxplot indicates important statistics regarding the distribution of the measurements, including outliers, which are values below the MIN or above the MAX. It also shows how tightly the measurements are grouped and whether the distribution is symmetrical or not [66]

3.4.3 Batch Size

For training CNNs, all data cannot be fed to the model at a time unless it is comparatively a small dataset, because it is impossible to process vast amount of data at once using regular machine's processing power and memory resources. So, a particular number of examples of the training data are passed everytime to train the model. Then the gradient descent is calculated to estimate the errors between the predicted values and the original values. Next, the errors are fed back to the model using backpropagation technique for updating the weights of the neurons accordingly in order to reduce the loss values. This learning process keeps repeating until the errors are reduced to a desired minimum level [69].

The model's one circulation through the entire training dataset is called one training 'epoch', where samples are split into arbitrarily selected 'batch size' units. The batch size has significant effects not only on

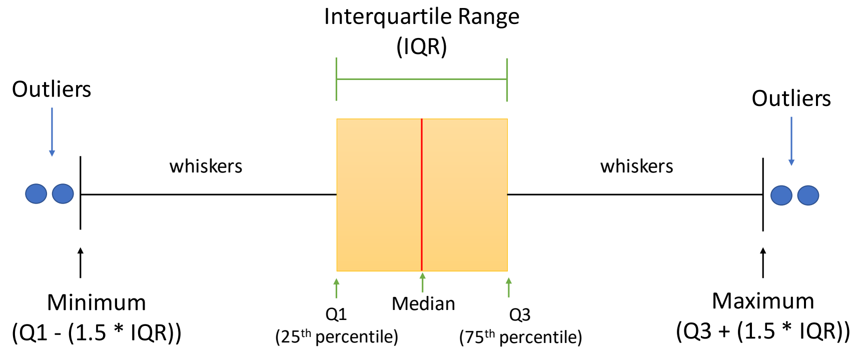


FIGURE 3.6: Boxplot interpretations

the speed of a model's learning, but also on the steadiness of the learning process [5] [65].

Since a larger batch size would most likely allow practitioners to exploit the parallelism of GPUs and computational speedups, thus they often want to train their model using a larger batch size. It can be assumed that the learning process would be more accurate and the weights of the neurons of the CNN model would be adjusted more accurately if more examples are included per batch during the training time. On the contrary, poor or noisy estimates regarding the error gradient would be generated as a result of using smaller number of examples. This can in turn, often cause noisy updates of the model weights; there would be several different updates with different error gradient estimates because of the lack of enough samples to converge to a certain estimate. However, practitioners had exposure to poorer generalization capabilities because of large or very large batch sizes (e.g., 256, 512, 1024 etc.) [38].

The larger batch sizes take fewer steps for a fixed number of training epochs which might lead to poor generalization capability [60]. Sometimes these noisy updates can rather lead to faster learning by providing a normalizing effect, smaller generalization error and a good model. Thus, training the model for a longer time instead by using smaller batch sizes might be beneficial. Smaller batches do not need to make a full pass through the entire training data for updating the weights, therefore, they tend to converge quickly. Also, fitting one batch of training data in GPU memory becomes easier because of using smaller batch sizes [60].

In many cases, batch size ≤ 32 has been demonstrated to provide the best outcomes [69]. A moderate batch size of 32 examples is often chosen as the default value [11] for the models and has been proven to achieve good stability of training and performance of generalization. It has been chosen as the default value for the batch size for this work as well.

Considering the GPU memory constraints of the systems that are being used for the experiments in this work, a range of batch sizes of 8, 16 and 32 has been considered for experiments on different datasets. The

other two hyperparameters were kept at their respective default vales during the individual experiments on batch size; they are the learning rate of 0.00001 and the number of epochs of 600. These fixed values have been decided upon the compatibility check based on some preliminary experiments on the considered datasets. Moreover, Bengio *et al.* [11] have suggested to pair up a high learning rate with a large batch size and a low learning rate with a small batch size, meaning that, the ratio should be compatible.

3.4.4 Learning Rate

Learning rate is a significant hyperparameter for achieving a good performance in an NN every time the model weights are updated after every iteration, learning rate controls the model's degree of change based on the estimated error. Here, the rate at which the model learns or the rate at which the weights get updated while the model is going through the training phase, is known as the 'step size' or often, the 'learning rate' [22].

The learning rate can be tuned by hand; the range is often kept greater than 10^{-6} and less than 1.0 [75]. Unfortunately, there is no optimal learning rate to be calculated analytically for any given model based on a particular dataset. There is no other option except discovering a good learning rate just via trial and error. A 'too low' learning rate will make the progress of training very slow, requiring a greater number of training epochs to reach to the minimum point and also, it might get stuck into local minimas; hence, may not be able to reach the global minima [45]. Larger learning rates would require fewer training epochs, but the case might happen too where a 'too high' learning rate would result in producing unusual divergent behaviour. It might 'jump over' the ideal minimum point, the updates might cause a noisy convergence, so it would keep bouncing within in-between but might never hit the minima. An appropriate learning rate will lead the training process to reach the minimum point swiftly and might help to converge at some time, resulting in lower score for loss values [99].

The range of learning rate from 1.0×10^{-5} to 1.0×10^{-4} has been chosen for experimenting on the considered four datasets. The other two parameters were kept at their respective default values while experimenting on learning rate individually; they are the batch size of 32 and number of epochs of 600.

An important note should be mentioned here to distinguish between the 'learning rate' used for training the model and the 'learning rate' suggested by Optuna. The learning rate which is used to train the NN, updates the gradient descent accordingly to minimize the loss function. The Adam optimizer [102] is applied in the Flower Counter model by Tensorflow, which is responsible for adaptively updating the true learning rate and training the model using that. Optuna does not directly train-and-update the weights of model; it uses different searching algorithms as mentioned in chapter 2, to explore the configuration space for a potential good value of learning rate within the given range. Using that suggestion, the dedicated Adam optimizer would train-and-update the model accordingly, trying to find stablility with the goal of yielding loss values as low as possible [48].

3.4.5 Number of Epochs

Number of epochs is another important hyperparameter that can impact the functioning of any NN. If the model provides a lower error rate (better accuracy score) on training data but a higher error rate (poorer accuracy score) on validation data, then it means that the model has memorized the dataset and also the unwanted noise from training phase well enough to provide good training accuracy score, but not learn the actual patterns to be able to generalize well on validation (new) data. If the loss becomes too low, then there is a potential of overfitting [69]. The term 'underfitting' is used to describe the concept when the model provides a higher error rate (poor accuracy score) on both training data and also validation data; it means that the model has neither been able to learn the actual patterns, nor been able to generalize well on validation (new) data. Overfitting can often be solved by stopping the training of the model after a certain number of epochs while the validation error rate is minimum; increasing it would lead to overfitting otherwise [69]. Training for a longer period of time to a certain number of epochs would help the model to learn the patterns well enough to prevent underfitting as well. Therefore, it is significant to identify an appropriate number of epochs for any individual NN model [95] to be able to ensure good training.

Different ranges of epochs have been chosen for experimenting on each of the four datasets. The other two parameters were kept at their respective default values while experimenting on epochs individually; they are the batch size of 32 and learning rate of 0.00001. One study considers epochs in the lower range of (25-50-100-200) and another in the higher range of (200-400-800-1600). Optuna's '*trial.suggest_categorical*' API has been used to run these experiments.

3.4.6 Combined impact of hyperparameters

After the individual analysis on the hyperparameters, the combined effect of all the three hyperparameters have been taken together into consideration. In particular, all three parameters are varied together for a study of 60 trials, with the purpose of acquiring knowledge about the possible good configuration settings for each of those four datasets. Two seperate experiments have been conducted on each of these datasets: one with lower number of epochs (25, 50, 100, 200) and another with higher numbers (200, 400, 800, 1600).

3.4.7 Analysis of Optuna's performance on test datasets

After that, based on the suggestions from Optuna about possible suitable hyperparameter values for batch size and learning rate, the model has been trained 10 times again for different higher number of epochs (200,400,800,1600) on the same training dataset.

In order to assess Optuna's performance and for checking the overfitting problem, it was tested on both the same populations and different populations. As stated in section 3.2.3, the same population indicates the 10% of dataset that has been shown in Table 3.3 and different test populations are chosen from a different camera-day as shown in Table 3.4. From 10 sets of testing, the results of different accuracy metrics, which

are MAE, MRE, RMSE and loss, have been averaged and the results are used to generate respective boxplot graphs. After that, the model is trained and tested 10 times using the default values (batch size 64, learning rate 1.0×10^{-5} and number of epochs 3000) used in Chowdhury's work [19]. The results are averaged for the considered accuracy metrics and used to generate respective boxplots for four different datasets. Afterwards, the testing results of the current and the previous work are compared and analyzed.

3.5 Summary

In this chapter, an overall overview of the CNN-based Flower Counter application, being used as the DL application as a part of the case study has been provided. After that, the techniques for annotating the images, preparing ground truths and corresponding density maps for training the model has been described. Followed that, some representative sample images from different datasets and their density maps have been depicted. Next, a brief description of the major characteristics training and test datasets have been presented along with the number of images per bin according to the binning technique applied in this work. Moreover, the histogram distribution of the training and test datasets have been illustrated. Following that, the statistical concepts of boxplots for understanding the experimental results have been shown. Furthermore, the significance of the three hyperparameters that have been chosen for the experiments in this work have been discussed in detail. Later, the experimental design, settings and methodologies with corresponding reasoning have been explained.

4 Results and analysis

This chapter presents the details of the experiments conducted on different sets of hyperparameters and the analysis of the results. Later, for the sake of assessing Optuna's parameter selection and associated prediction accuracy, the results are compared with previous experiments with the Flower Counter application [19].

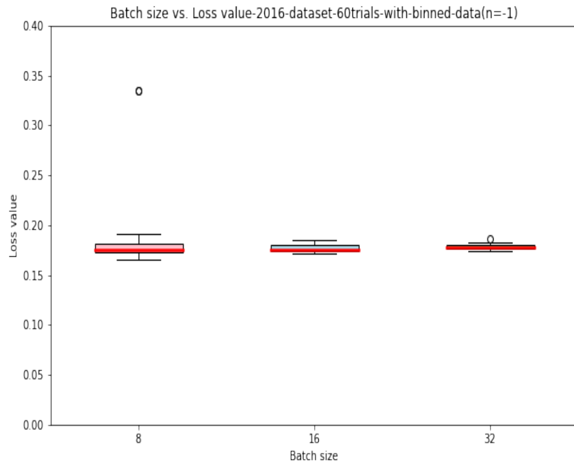
Among several possible hyperparameters of any CNN model, three major hyperparameters have been chosen for this work. They are batch size, learning rate and numbers of epochs. The chapter begins with the analysis of the individual impact of these hyperparameters, then their combined impact is analyzed and later a comparison with Chowdhury's work [19] is presented.

4.1 Experimental results and analysis of batch size

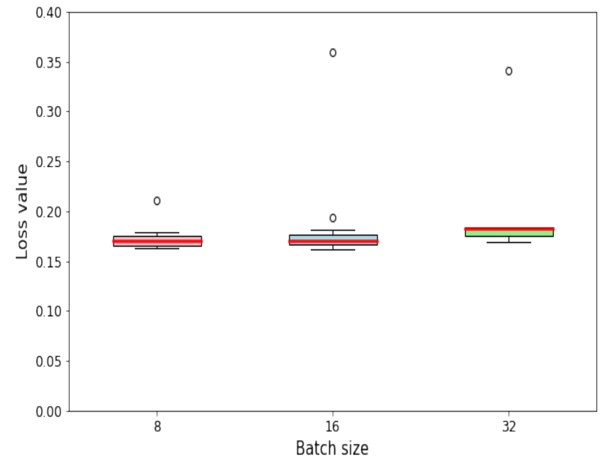
In this section, the Optuna-conducted experiments varying 'batch size' on four datasets are described. Figure 4.1 shows the Optuna-driven impact of batch size vs. pixel-wise loss values on the validation set for each of the four datasets. Table 4.1 shows the number of trials per batch size from the studies conducted on the datasets. Since the width of the IQRs of each batch size are visually quite similar, the numbers aid in understanding whether there were actually few points chosen by Optuna itself or there were in fact many points which were tightly grouped.

Figure 4.1a shows the results of batch size vs. loss values on validation images of the 2016-all dataset. The median values of the validation loss values for the batch sizes (8, 16, 32) are ∼0.17 and less than ∼0.20; the *MIN* values start from ∼0.16 and the MAX values cover the range up to ∼0.19. This is very tightly grouped. Table 4.1 shows that the batch size 8 was the most popular as it was chosen 45 times and the other two batch sizes 16 and 32 were less popular as they were chosen 7 and 8 times respectively. Figure 4.1b shows the results for the 2018-split dataset with the loss values in the similar ranges but with few outliers. Batch sizes 8 and 16 were more popular with almost equal number of trials 28 and 25 respectively but batch size 32 was the least popular choice with just 7 trials for this dataset.

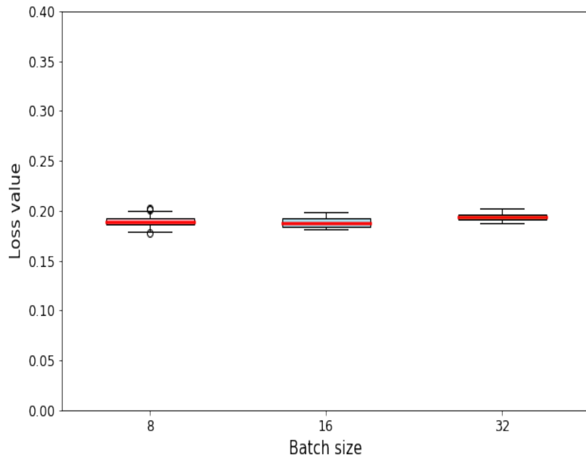
Figure 4.1c and Figure 4.1d show the results for the 2016-1109-5-12 dataset and the 2018-july-2-23 dataset respectively. The results for the 2016-1109-5-12 dataset are again very tightly grouped for batch size 8 and 16 with equal number of trials for each of them, but batch size 32 is the least popular choice again. The batch size 16 is the most popular choice for the 2018-july-2-23 dataset; this dataset shows the lowest ranges of loss values compared to the other datasets because it is overall a less dense dataset. There are more images with zero or fewer flowers from the begining of the growing season on July 2-5 particularly. All the datasets have



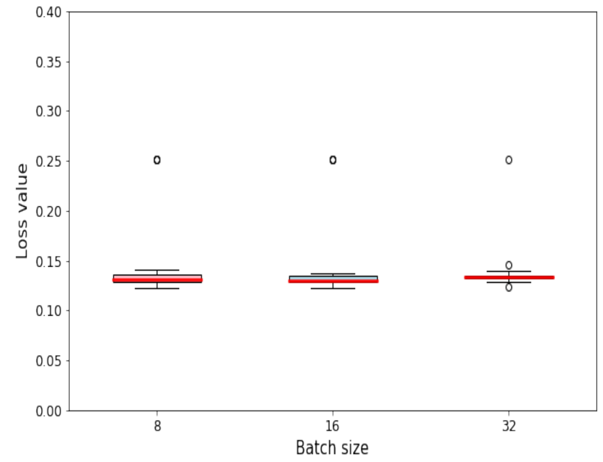
(A) 2016-all dataset with binned data



(B) 2018-split dataset with binned data



(C) 2016-1109-5-12 dataset with all data



(D) 2018-july-2-23 with all data

FIGURE 4.1: Batch size vs. validation loss (single parameter experiment)

shown bimodal behaviour for the experimental results presented here; either it trains or it does not.

Table 4.1: Total number of trials in study per batch size used in batch sizes experiment

	Batch size		
Dataset	8	16	32
2016-all	**45**	7	8
2018-split	**28**	**25**	7
2016-1109-5-12	**27**	**27**	6
2018-july-2-23	**19**	**30**	11

Overall, Table 4.1 shows that the distribution for batch size 32 is less stable, because there are very few datapoints and Optuna considered this to be a less-promising value, even though the results from the loss function were not that different. So except for a few random outliers, there is no major difference in the IQR ranges of the loss values for each of the batch sizes that has been considered for the datasets. It did not impact the range of MAX or MIN or even the median values of the loss values, even though the number of trials per batch size vary considerably. These observations indicate that there is no distinct individual impact of the range of values of the hyperparameter 'batch size' on the loss values.

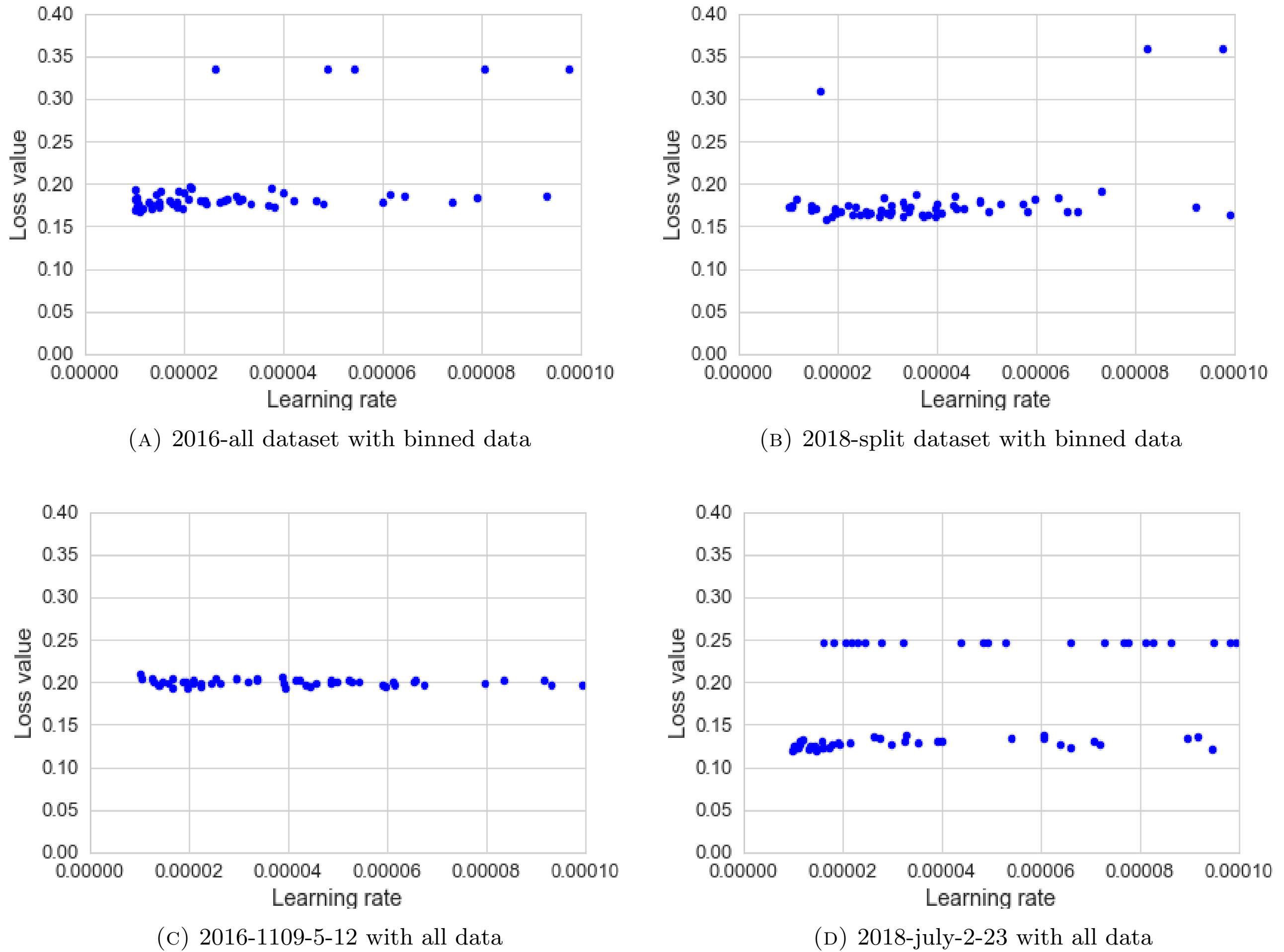
4.2 Experimental results and analysis of learning rate

Figure 4.2 shows the effect of the hyperparameter 'learning rate' on the validation loss values for different training datasets. For all these datasets, Optuna has explored the given search range entirely. Optuna kept searching more in the lower scale in order to find more stable regions. That is why, in Figure 4.2a and 4.2b, and also for Figure 4.2c, the number of trials are quite dense in the lower region. The loss values are in the same ranges ($\sim$0.15 to $\sim$0.20) for validation runs that correspond to successful training. A successful training would indicate to the fact that the model has been trained to the point where it can identify identify objects in the image as flowers that somewhat correspond to the annotated ground truth. Therefore, predicted density map is not uniformly zero.

The 2018-july-2-23 dataset exhibits unstable behaviour as shown in Figure 4.2d. Almost half of the points yield the loss values $\sim$0.13 and the rests $\sim$0.25 within the provided learning rate range. This means the model was trained properly for some trials (those with lower loss values $\sim$0.13) not at all during the others (yielding consistent higher loss values $\sim$0.25). There could be many reasons behind this behaviour for that particular dataset. As the whole dataset was used and there are many zero flowers as seen in Figure 3.4d, it did not train successfully for certain trials. Depending on the manner in which images were divided up into batches, the training may have had difficulties determining the insights to reduce the loss values for the model. Some batches may provide good loss values; others could not converge so the model remained poorly trained.

Overall, as observed from the graphs, Optuna is pointing towards the lower range of values of learning

rate (as in ≤ 0.00006) to be a good choice for this model. That should be sufficient for training the model and be able to detect flowers from various images. A wider range of learning rate values could be unstable due to the data skew towards sparse images and random selection of images, even though this effect is dampened for the binned datasets. Figure 4.2 shows that the values of the hyperparameter 'learning rate' selected by Optuna for the trials have a significant impact on the validation loss metric obtained from the training of the model.



(A) 2016-all dataset with binned data

(B) 2018-split dataset with binned data

(C) 2016-1109-5-12 with all data

(D) 2018-july-2-23 with all data

FIGURE 4.2: Learning rate vs. validation loss (single parameter experiment)

4.3 Experimental results and analysis of numbers of epochs

Figure 4.3 shows the Optuna-driven impact of hyperparameter 'epochs' on different datasets for different ranges. Table 4.2 aids in understanding the IQR spreads according to the number of trials selected per epoch by Optuna in the conducted studies respectively for different datasets. It shows how many datapoints are in each column.

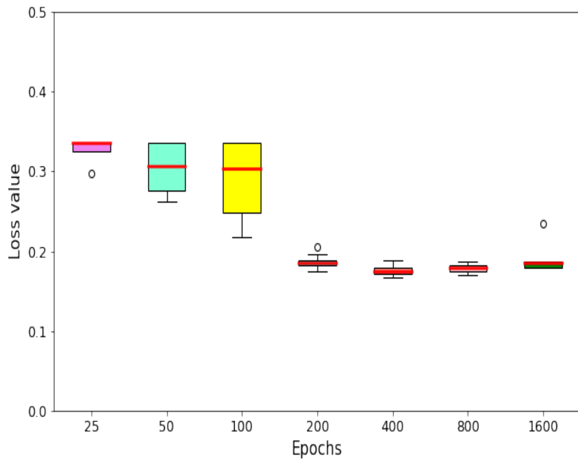
For the 2016-all dataset in Figure 4.3a, 200 and 400 epochs were popular choices with 48 and 38 trials respectively; they were tightly grouped producing lower loss values ∼0.20. The IQR ranges are widespread and dispersed with few trials for lower number of epochs, specially for 50 and 100 epochs. For the 2018-split dataset in Figure 4.3b, 200 and 1600 epochs were more popular than the others and 50 epochs was the least popular one. Except for a few outliers, the loss values were lower for 1600 epochs, but a very wide IQR is noticeable for 200 epochs with comparatively higher loss values. In spite of being a popular choice by Optuna, training for 200 epochs was not able to provide stability and lower loss values. For the 2016-1109-5-12 dataset in Figure 4.3c, 100 and 800 epochs were popular choices; 800 epochs provided comparatively lower loss values than others also. For the 2018-july-2-23 dataset in Figure 4.3d, 200 and 400 epochs were popular choices and they yielded low loss values except some outliers.

The loss values are mostly in lower ranges of the scale starting from (or after) 200 epochs for the datasets except for the 2018-split dataset (Figure 4.3b). For this dataset, 200 epochs was the most popular choice with 48 trials (as seen in Table 4.2) with a wide range of values, leading to a wide IQR spread in the boxplot. 400 epochs had just 6 trials with a wide IQR spread in the boxplot as well. Therefore, both 200 and 400 epochs were quite unstable and provided higher loss values at times, suggesting that more epochs were necessary to train the model to learn all the diverse characteristics it might have encountered. Overall, Figure 4.3 indicates that the loss values were in higher ranges for lower number of epochs irrespective of any dataset. So the model does not train well for lower number of epochs (25-50-100) as the loss values are in the range of ∼0.30 (Figure 4.3d) or above ∼0.30 (4.3a, 4.3b and 4.3c) to ∼0.40. There is also noticeable variation in the IQR ranges for the lower number of epochs. Though Optuna tried different lower number of epochs (as seen in Table 4.2), they were not enough for ensuring good training of the model and producing low loss values.

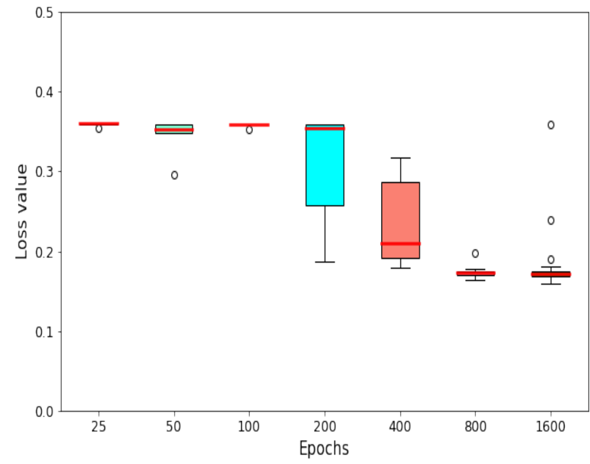
Each of these datasets exhibits a similar range of loss values from 200 epochs to 1600 epochs the range starting mostly from ∼0.20 (for the 2018-july-2-23 dataset in Figure 4.3d, it is below ∼0.20). The median values and the IQR range for all of them for those epoch numbers are also similar. Therefore, higher number of epochs (starting from or after 200) are suitable for the Flower Counter model to train sufficiently to identify flowers from various images of various weather conditions. From this analysis of the values in Figure 4.3, the Optuna-driven values of the hyperparameter 'epochs' has a substantial impact on the validation loss obtained by the model of the Flower Counter.

Table 4.2: Total number of trials in study per epoch used in epochs experiment

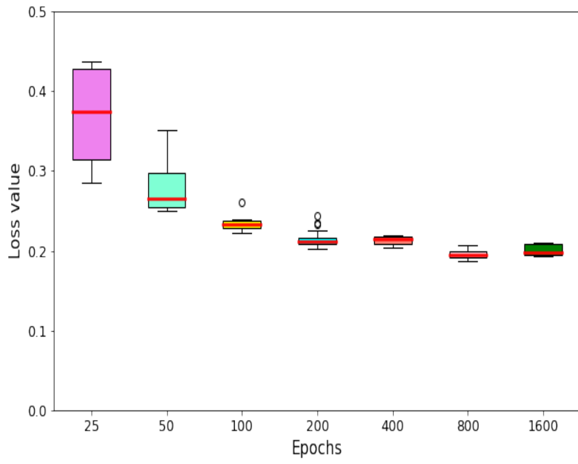
	Epochs						
Dataset	25	50	100	200	400	800	1600
2016-all	5	5	8	**48**	**38**	11	5
2018-split	7	5	8	**48**	6	16	**30**
2016-1109-5-12	9	6	**45**	11	6	**37**	6
2018-july-2-23	6	6	6	**54**	**32**	11	5



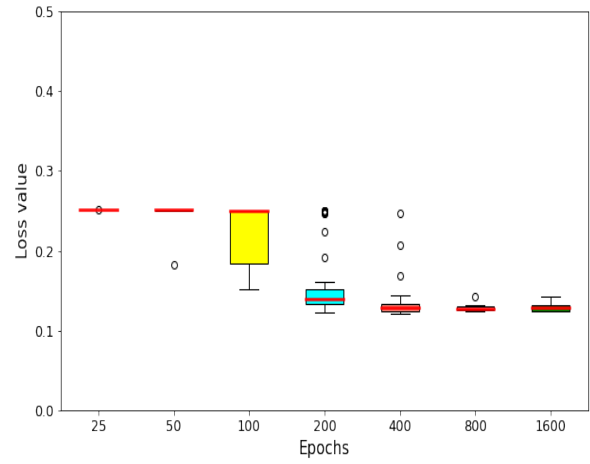
(A) 2016-all dataset with binned data



(B) 2018-split dataset with binned data



(C) 2016-1109-5-12 dataset with all data



(D) 2018-july-2-23 dataset with all data

FIGURE 4.3: Epoch vs. validation loss (single parameter experiment)

4.4 Simultaneous parameter search space

4.4.1 Analysis for lower number of epochs

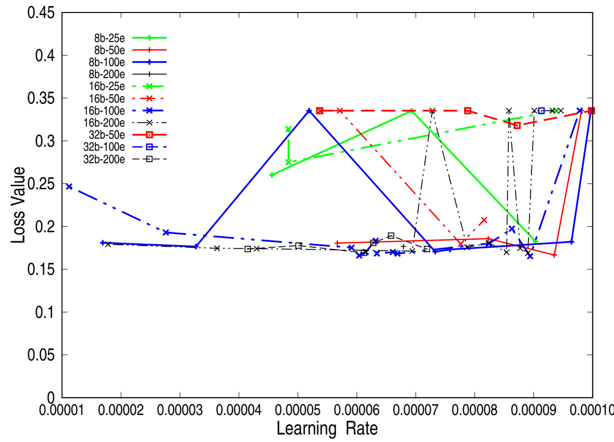
In this section, the analysis of simultaneous combined impact of three hyperparameters, namely batch size, learning rate and lower number of epochs (fewer or equal to 200), is presented on four different datasets. Figure 4.4 shows the result for the following range of hyperparameter values: the batch sizes are 8, 16 and 32; the learning rate is from 1×10^{-5} to 1×10^{-4} and the numbers of epochs are 25, 50, 100 and 200. The colours used in Figure 4.4 are green, red, blue and black, representing 25, 50, 100, and 200 epochs respectively. The plus signs with the solid lines, the stars with the dashed lines and the squares with the dotted lines indicate the batch size 8, 16 and 32 respectively.

For the **2016-all dataset**, Figure 4.4a shows that 100 epochs was a popular choice as indicated by the presence of blue lines. 100 epochs with batch size 16 produced very stable results of low loss values $\sim$0.20 till higher learning rate $\sim 9 \times 10^{-5}$. Also, 200 epochs with batch size 16 and 32 produced stable results till learning rate $\sim 7 \times 10^{-5}$. 25 epochs was totally unstable as indicated by the green lines. 50 epochs was unstable for different batch sizes as well (specially, for batch size 32) except with the batch size 8 in the range of learning rates from $\sim 5 \times 10^{-5}$ to $\sim 9 \times 10^{-5}$.

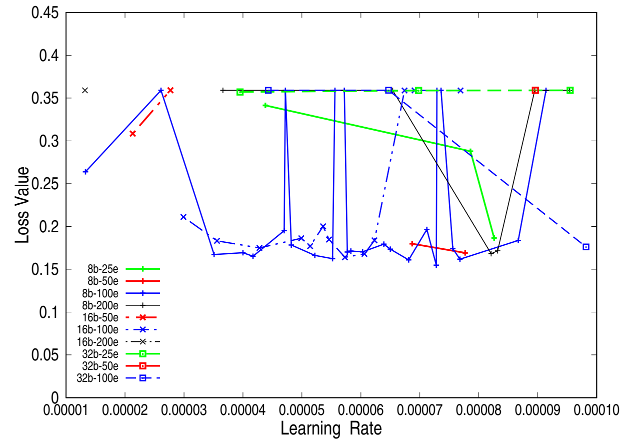
For the **2018-split dataset**, Figure 4.4b shows that Optuna explored very few points for the three combinations of batch sizes and epoch numbers 25 and 50. In some cases, it did not train at all; as a result, it produced a straight line hitting high loss values $\sim$0.36 (for instance, batch size 32 and 25 epochs). 100 epochs was very popular since many blue points are observed in the graph. Optuna explored many combinations for batch size 8 and 16 than batch size 32. Compared to the other combinations, 100 epochs with batch size 16 was more stable producing lower loss values $\sim$0.18 consistently upto the learning rate $\sim 6.0 \times 10^{-5}$.

For the **2016-1109-5-12 dataset**, Figure 4.4c shows that it yields few random points with epochs 25, 50, and 100 (red, green and blue) regardless of any batch size. For these three batch sizes and 200 epochs, it had fluctuated between the range of loss values between $\sim$0.20 and $\sim$0.42 within the range of learning rate from $\sim 4.0 \times 10^{-5}$ to $\sim 9.0 \times 10^{-5}$. Among all the combinations, Optuna found 200 epochs with batch size 8 comparatively the most stable configuration for learning rate below $\sim 6.0 \times 10^{-5}$.

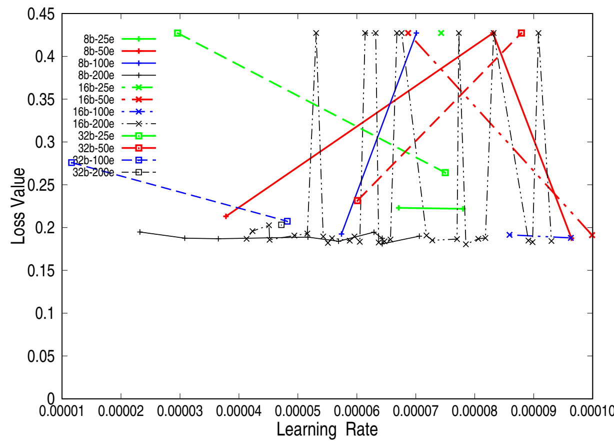
The **2018-july-2-23 dataset** in Figure 4.4d produces the lowest loss values for any combination compared to the other three datasets. As mentioned previously, this is due to the average sparsity of the dataset, so each of the density map would be zero. The existence of many blue points in the graph indicate that 100 epochs was the most popular choice. However, Optuna appeared to explore arbitrary points for all the combinations of different batch sizes and epochs. It kept exploring randomly between the low and high ranges and produced loss values from $\sim$0.10 to $\sim$0.20. For several combinations, they are consistently in the unstable region producing straight lines; meaning that it did not train at all for those combinations. Overall, for the whole dataset, it is quite unstable for all epochs.



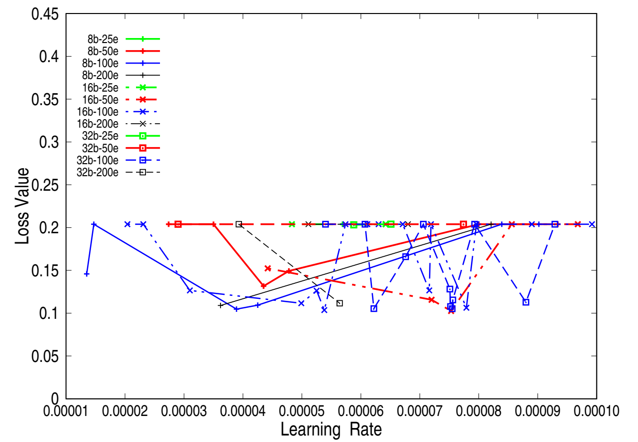
(A) 2016-all dataset



(B) 2018-split dataset



(C) 2016-1109-5-12 dataset



(D) 2018-july-2-23 dataset

FIGURE 4.4: Batch size, learning rate and epochs vs. validation loss (fewer epochs)

To summarize, very little green and/or red shows that Optuna did not want to use 25 or 50 epochs respectively; lots of blue and/or black shows that 100 or 200 epochs were popular respectively compared to the former two. Lots of plus signs or star signs or square signs indicate that batch size 8 or 16 or 32 was a popular choice respectively. Therefore, 200 epochs and batch size of 16 was quite popular for the 2016-all dataset as chosen by Optuna (in Figure 4.4a), but it was the most stable and successful with batch size of 32 with 200 epochs. For the 2018-split dataset (Figure 4.4b), batch size 8 with 100 epochs was the popular choice but it was inconsistent and unstable, whereas, batch size 16 with 100 epochs was comparatively more stable, so it was better than the former choice. Batch size 16 with 200 epochs was the most popular choice but it fluctuated a lot for the 2016-1109-5-12 dataset (Figure 4.4c), but batch size 8 with 200 epochs comparatively generated more stable, lower loss values. Lastly, though both of the batch sizes 16 and 32 with 100 epochs were popular choices for the 2018-july-2-23 dataset as spotted in Figure 4.4d, no combination was stable for this dataset.

All these observations from four different datasets indicate that most of the combinations for lower number of epochs kept exploring with little incremental success and were not able to yield very stable regions while training. Only 200 numbers of epochs performed comparatively better than the others for some datasets. The loss values for many of these combinations do seem to fluctuate a lot and Optuna cannot find stable regions; this indicates that the model does not train consistentlly enough for lower number of epochs, and that Optuna realizes this and chooses fewer combinations with those parameter values.

4.4.2 Analysis for larger number of epochs

In this section, the analysis of simultaneous combined impact of three hyperparameters has been presented on four different datasets for higher number of epochs (starting from 200 and higher). Figure 4.5 shows that result for this range of hyperparameter values: batch sizes are 8, 16 and 32; learning rate is from 1×10^{-5} to 1×10^{-4} and numbers of epochs are 200, 400, 800 and 1600. The color notation used in Figure 4.5 is the same as that of Figure 4.4.

For the **2016-all dataset** in Figure 4.5a, 800 epochs is a popular choice as indicated by the presence of lots of blue lines. 800 epochs with both of the batch sizes 16 and 32 produce stable results and low loss value scores $\sim$0.20 upto the learning rate of 8×10^{-5}. 400 epochs does not provide stable results (especially with batch size 8); although it does train with batch size 32 even in the higher learning rate 9×10^{-5}. 200 epochs is not a popular choice and unstable as indicated by the green lines. 1600 epochs with batch size 16 does not train for higher learning rates and thus produces a constant line in higher loss value range $\sim$0.35 as indicated by the black dotted line.

For the **2018-split dataset** in Figure 4.5b, the loss was more stable for 400 epochs across the range of batch sizes, so it was the most preferred number of epochs. More red points indicate that 400 epochs was more popular than the others. Optuna did not explore 200 epochs much, since very few green points are observed. For batch size 8 and all selected epoch parameters, Optuna explored as few as either two or three

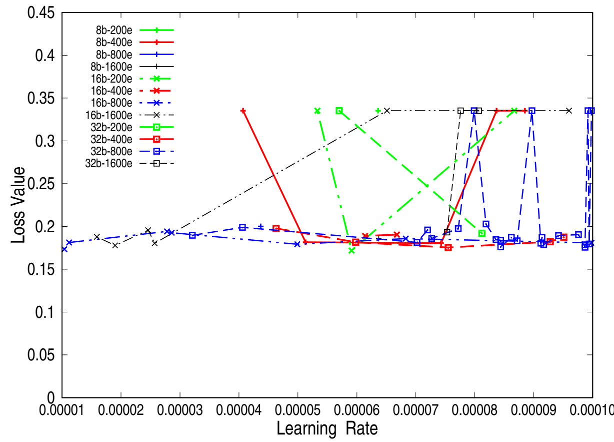
points only. Batch size 16 with 400 epochs was noticed to explore more combinations than the other epochs; it has explored many combinations with low loss values $\sim$0.18. For batch size 32 and 400 and 800 epochs, the similar behaviour is noticed again.

For the **2016-1109-5-12 dataset** in Figure 4.5c, Optuna chose very few combinations of any epoch with batch size 8 because very few plus signs with solid lines are observed in the graph. 200 and 1600 epochs were not at all popular for this dataset since very few green and black points are noticed in the graph. For batch size 16, Optuna selected the most combinations to have 400 epochs; this showed very good stability and low loss values $\sim$0.19 up to the learning rate of $\sim 6 \times 10^{-5}$. Though few points for batch size 16 and 1600 epochs were selected, those combinations showed good stability also. For batch size 32, Optuna again selected very few points. Comparatively, batch size 16 with 400 epochs showed better stability than others below a learning rate of $\sim 6 \times 10^{-5}$.

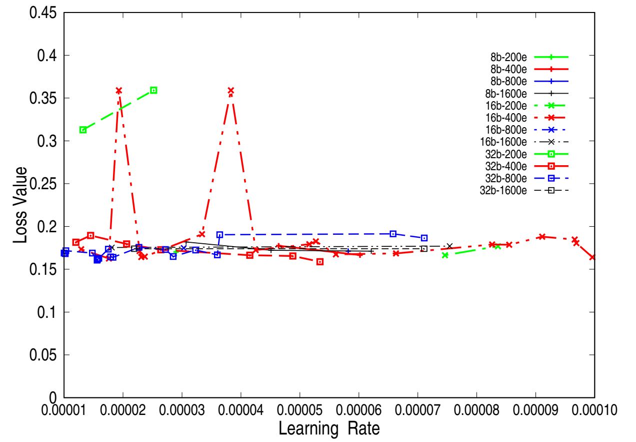
For the **2018-july-2-23 dataset** in Figure 4.5d, it did not train at all for batch size 8 with any epoch combinations, resulting into a constant loss value of $\sim$0.26. It explored a lot of combinations with 200 epochs, but few for 1600 epochs. For batch size 16 and 200 epochs, Optuna has selected many points within the range of learning rate from $\sim 3.5 \times 10^{-5}$ to $\sim 9.0 \times 10^{-5}$, yielding loss values in the entire range of $\sim$0.12 to $\sim$0.26. Though 200 epochs seems to be a popular choice for the 2018-july2-23 dataset, it exhibits instability again as seen in the previous figures. For the other epochs, Optuna just visited either one or two points. For batch size 32 and 200 and 400 epochs, Optuna has again explored quite a few combinations with random inconsistencies. For 800 and 1600 epochs, it did not train at all. Batch size 16 with 200 epochs comparatively showed a little bit of stability upto the learning rate $\sim 6.0 \times 10^{-5}$; but the whole dataset is mostly unstable for any number of epochs.

To summarize, lots of green and/or red shows that 200 or 400 epochs respectively were popular choices to Optuna compared to less blue and/or black for 800 and 1600 epochs respectively and different signs refer to different batch sizes. Therefore, 800 epochs was the most preferred choice and also more stable than other combinations for the 2016-all dataset as seen in Figure 4.5a. The 2018-split-dataset was more stable for 400 epochs with different batch sizes as observed in 4.5b. Batch size 16 with 400 epochs is noticed to provide the lowest error scores and stability for the 2016-1109-5-12 dataset as chosen by Optuna (in Figure 4.5c). Though 200 epochs seem to be a popular choice for the 2018-july-2-23 dataset, but again, it exhibits instability as seen from Figure 4.5d.

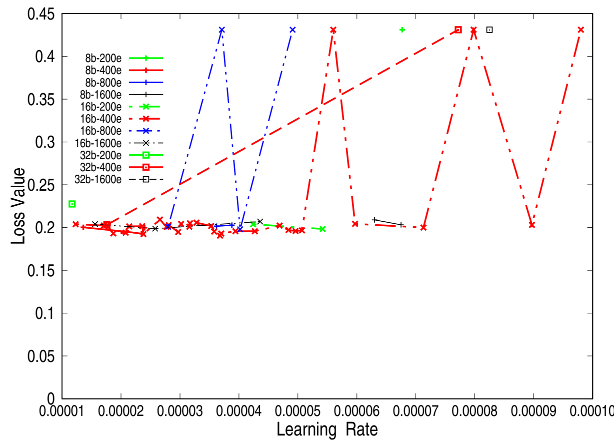
In general, Figure 4.4 and 4.5 showed that there is stability as well as inconsistencies for different combinations for different datasets. Specifically, the 2018-july-2-23 dataset had lots of inconsistencies and points producing the constant straight lines indicated that it did not train properly for those combinations. Similar behaviour was noticed for learning-rate based experiments as shown in Figure 4.2. Yet, compared to the experiment with lower number of epochs, the higher number of epochs trained better and found good stability for various combinations overall as seen from these two figures (better results for higher number of epochs is also evident from Figure 4.3). Therefore, for training the Flower Counter model better and for finding



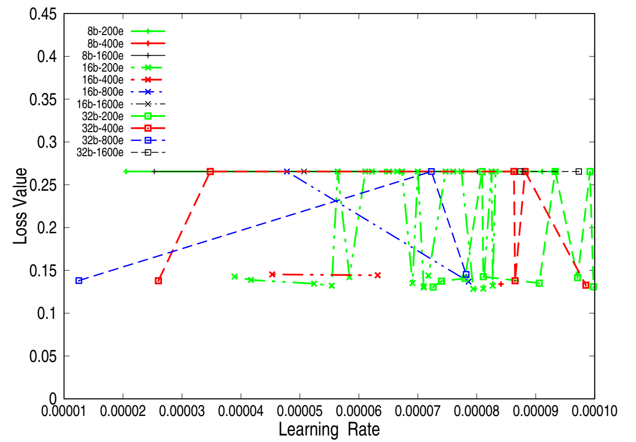
(A) 2016-all dataset



(B) 2018-split dataset



(C) 2016-1109-5-12 dataset



(D) 2018-july-2-23 dataset

FIGURE 4.5: Batch size, learning rate and epochs vs. validation loss (more epochs)

comparatively more stability in further experiments, hyperparameter combinations in the range of higher number of epochs should be selected. The stability is observed to be comparatively better than that yielded by the lower number of epochs.

From the results observed so far, it is quite evident that the search techniques which are applied by Optuna, sometimes randomly choose a lot of selections in an unstable region. In that case, either the search space should be further constrained or a different optimization function could be used for Optuna to increase stability. The optimization function that has been used here has taken a single train-test run and that is given to Optuna to make the loss value as low as possible according to the choice of hyperparameters for each trial. During this process, Optuna has not been advised that stability is a concern.

To address this issue, a more complicated objective function could be given to Optuna. For example, rather than doing a single train-test run with Tensorflow, the objective function could be given "k" train-test runs and the average or the minimum of those loss values can be selected to train the model for some values of "k". In the current experimental setup, for each training, Tensorflow is trying to minimize the pixel-wise loss. Based on that, the testing is run and each test run yields a loss value. In order to assure that Optuna is operating in a stable region, one way it could be done is that, run that train-test "k" times in an 'unstable region' initially. It is very likely that one of those "k" train-test sequences, even with the same set of hyperparameters, the model would not end up training well and therefore, a high loss value would be produced for the test dataset. Now if the Optuna function is made to take the maximum of the loss value among those "k" test runs, a high loss value would be reported when it would be operating in one of those unstable regions. Now Optuna would be made to reject that particular choice of hyperparamters that generated that high loss value and the search space can be constrained accordingly. Another way to provide more stability in the value of the objective function might be to restrict the search space. From the preliminary experiments, the search range could be restricted just to stable regions.

4.5 Testing accuracy results

4.5.1 Same population experiments

In this section, the results of the four accuracy metrics (MAE, MRE, RMSE and Loss) using Optuna's parameters and Chowdhury's [19] parameters on the same populations has been presented, compared and analyzed as shown in Figure 4.6. The results for the four different accuracy metrics have been plotted for the four different datasets. The boxplot distributions are demonstrated using a grid of $4 \times 4 = 16$ square boxes where each column represents one dataset and every row represents an accuracy metric.

The following configurations as shown in Table 4.3 are suggested by Optuna to be the best configuration parameters after the corresponding studies conducted in the section 4.4.2. Since these configurations provided the lowest loss values from their validation training, they are considered the best ones (to be noted, they are not guaranteed to produce the best results for repeated training and testing). As well, the configurations

maybe in unstable regions. Thus, only runs that produced non-zero predictions are used. Then based on the associated model generated by the training, the model is tested on the generated dataset from the same population (10%, as described in Table 3.3).

Table 4.3: Parameters for testing suggested by Optuna experiments

Dataset	Learning rate	Batch size	Epochs	Validation loss value
2016-all	8.94×10^{-5}	16	100	0.165
2018-split	5.34×10^{-5}	32	400	0.159
2016-1109-5-12	3.68×10^{-5}	16	400	0.191
2018-july-2-23	7.94×10^{-5}	16	200	0.128

Overall, the results for the median values are close for the 2016-all dataset for both of the experiments. 2018-split dataset has conflicting results as the MRE follows a different direction than the other metrics. The results for the median values are close for the 2016-1109-5-12 too except for RMSE. 2018-july2-23 is only better for MRE and RMSE for Chowdhury [19], but the results are quite unstable, given the wide range of the IQRs.

The 2016-all dataset and the 2018-split dataset are the binned datasets. Therefore, the impact of data skew towards the sparse images are reduced by the process of binning and their corresponding training-testing datasets now have a balanced mixture of all kinds of images. For the 2016-all dataset, the median values for MAE are ∼8.30 and ∼8.11, for MRE it is ∼0.32, for RMSE are ∼17.12 and ∼16.74, and for Loss are ∼0.17 and ∼0.19 for Optuna and Chowdhury's [19] work respectively. Since the medians/means are very similar and the coefficient of variation is nearly zero, it means that the experiments did not show a difference in the metrics for both of the works for this dataset.

For the 2018-split dataset, the median values for MAE are ∼6.40 and ∼3.24, MRE are ∼0.22 and ∼0.34, RMSE are ∼13.23 and ∼3.27, and Loss are ∼0.17 and ∼0.12 for Optuna and Chowdhury's [19] work respectively. Therefore, Chowdhury's parameters produced lower error results in general. An exception is the MRE score. Mathematically, if the larger errors are for dense images then that would enhance the RMSE. If the larger errors are for the sparse images, then the MRE would be higher and the RMSE would be lower. After examining the predicted vs. actual flower counts, it appears that the latter case was more prevalent. Some sample testing outputs of actual vs. predicted counts for Optuna and Chowdhury's [19] work for 2018-split dataset and 2016-1109-5-12 dataset are shown in Figure 4.7 respectively. Among 10 runs, three sample test results have been shown here using three different colors and shapes, such as, orange triangles, green plus signs and blue circles. The black diagonal line shows the line where the actual count is equal to the prediction for all images correspondingly. Since the number of dense images is fewer than the sparse images (i.e. the image flower counts are concentrated below 60) in 2018-split dataset (Figure 4.7a and 4.7c), the error calculations will be more representative of the sparse images. Hence, the opposite trend of MRE and RMSE is noticed in Figure 4.6 for this dataset. Another thing that is noticeable is, with the increasing

number of flowers the results are more underpredicted for one of the three runs for both of the works, such as, the blue in Figure 4.7a and green in Figure 4.7c respectively. The selection of the training run in case of underprediction is random for them and the reason might be again because of the model's less exposure to the diversity of denser flowers.

For the 2016-1109-05-12 dataset, the median values for MAE are ∼8.60 and ∼9.02, MRE are ∼0.25 and ∼0.27, RMSE are ∼16.33 and ∼9.60, and Loss are ∼0.19 and ∼0.20 for Optuna and Chowdhury's [19] work respectively. For this dataset, there are close results for three of the metrics. It means that the experiments did not show a difference for those metrics; therefore, the medians/means are very similar and the coefficient of variation is nearly zero. It is only RMSE that is different, so some sample testing outputs of actual vs. predicted counts for Optuna and Chowdhury's [19] work are shown in Figure 4.7b and 4.7d respectively. Again, three sample test results have been shown here using different colors and shapes as mentioned previously. It is evident from the figure that there are a lot of images above 250 that are underpredicted compared to the actual count, which has contributed to the RMSE. 2016-1109-5-12 dataset has comparatively more numbers of dense images than 2018-split dataset and the model performs almost equally and the results are actually pretty close for both of the works as observed in Figure 4.7b and 4.7d. Therefore, the visual difference or gap of the MRE and RMSE trend is less prevalent for this dataset compared to 2018-split dataset as observed in Figure 4.6. For this dataset, the model mostly underpredicts for both of the works as observed from the corresponding three runs.

For the 2018-july-2-23 dataset, the median values for MAE are ∼5.23 and ∼4.45, MRE are ∼0.34 and ∼0.22, RMSE are ∼11.05 and ∼6.64, and Loss are ∼0.13 and ∼0.11 for Optuna and Chowdhury's [19] work respectively. For this dataset, Chowdhury's [19] work's configurations are less stable, because wide IQR ranges are noticeable in the accuracy metrics compared to Optuna. The median values for the metrics MAE and Loss are close, the MRE is unstable, and RMSE is different. Due to the instability of the dataset, running for more epochs is likely to reduce the instability, because the model that is generated has much more opportunity to find a descent to lower training loss values, given 15 times the number of epochs. This may also lead to overfitting, which will be evaluated in more detail in the following section 4.5.2.

This dataset has shown unstable behaviour and instances of not training in the other experiments as observed earlier; even here during the training process, it was sometimes training, sometimes not. Therefore, the training was redone several times in order to yield the testing results for which it could train. The histogram distribution in Figure 3.4d shows that this dataset is quite imbalanced because of the existence of relatively more sparse images. There are relatively few dense images in this dataset and the whole dataset is used for the experiments. So the training might have become biased towards the sparse images, causing the inconsistencies in the training process and overfitting. This fact might have had the influence on the model's learning process and thus, impacted its ability to generalize well on the test dataset. The observations indicate that the model could not always train better with increasing numbers of epochs in Chowdhury [19] either, leading to widely dispersed IQRs, hence poor testing results overall.

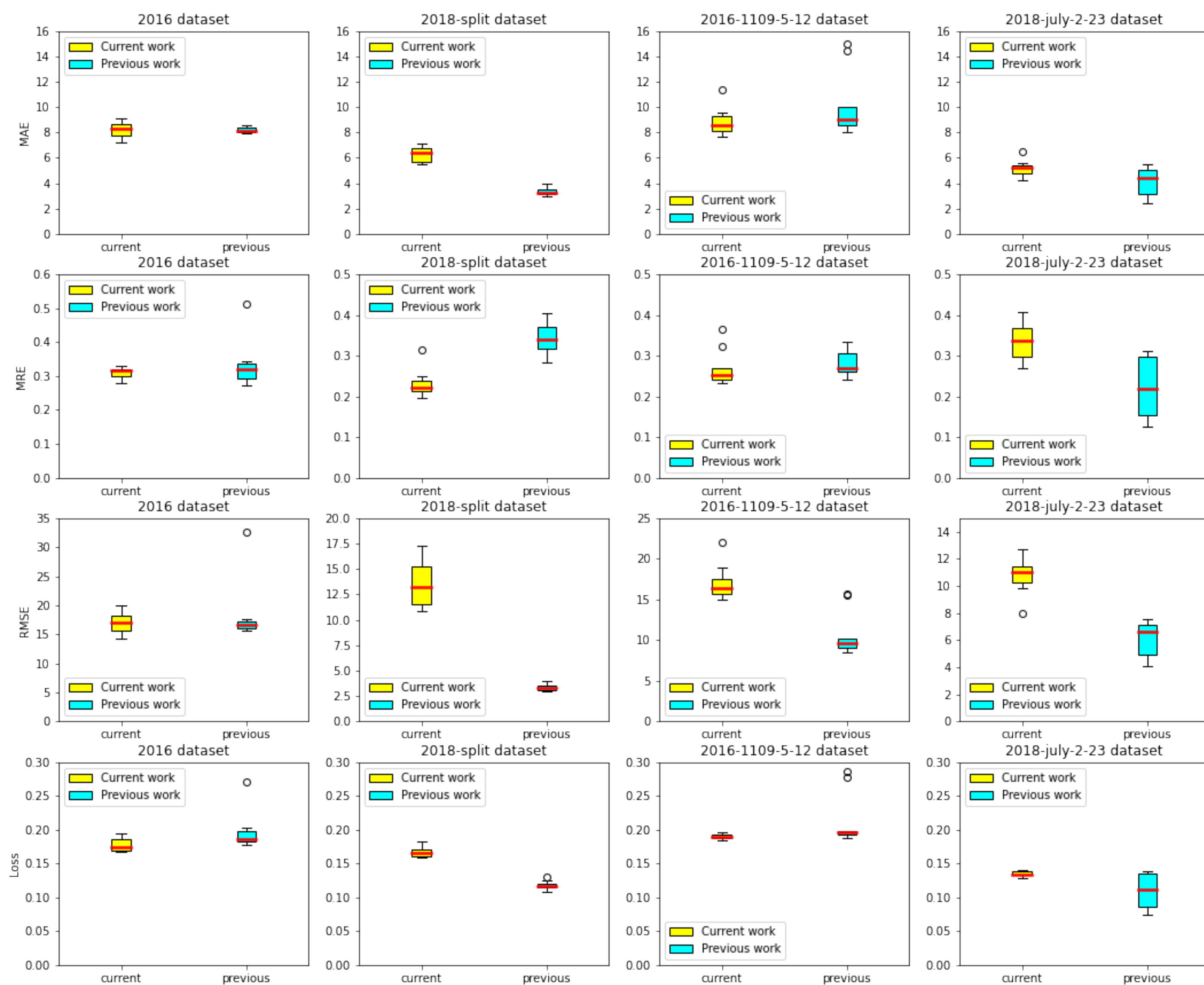
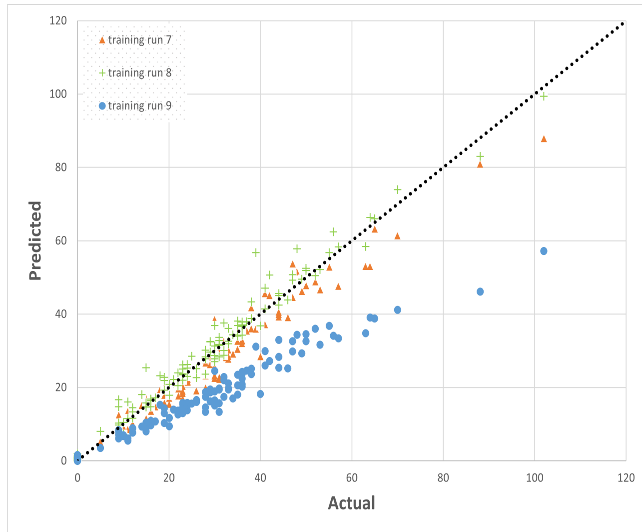
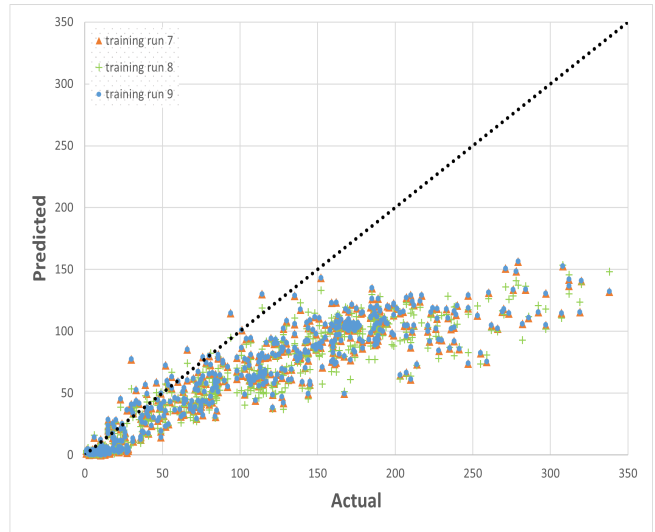


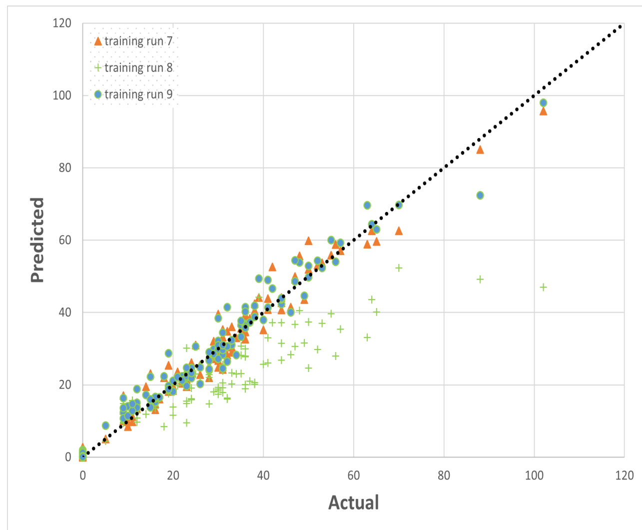
Figure 4.6: Same population experiments (Optuna vs. Chowdhury [19])



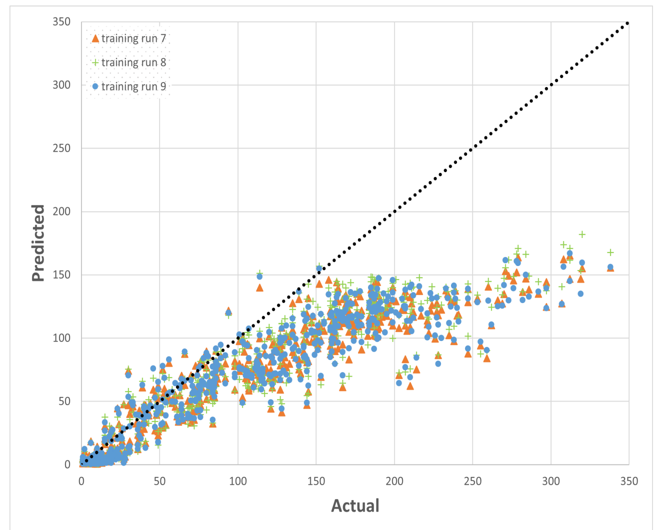
(A) 2018-split dataset (Optuna)



(B) 2016-1109-5-12 dataset (Optuna)



(C) 2018-split dataset (Chowdhury [19])



(D) 2016-1109-5-12 dataset (Chowdhury [19])

Figure 4.7: Samples for Actual vs. Predicted counts (same population experiments)

The reason behind Chowdhury's [19] work performing comparatively better overall than Optuna could be explained by the fact that, it has been trained using more epochs. Therefore, better accuracy scores for different metrics for Chowdhury's [19] parameters have been provided than the ones suggested by Optuna while testing on the same populations. Any ML model produces better accuracy scores with the increasing number of epochs, because the model starts to memorize the characteristics with the ongoing training for the longer period of time and therefore, yield lower error scores (which might sometimes lead to 'overfitting' [96]).

4.5.2 Different population experiments

In this section, the results of the four accuracy metrics using Optuna's parameters and Chowdhury's [19] parameters using a test dataset from a different population (from a different camera day in the same year with somewhat similar images) have been presented, compared and analyzed as shown in Figure 4.8. The y-axis for different datasets have been changed accordingly so that the boxplots can be analyzed better is where the IQRs are very small or tightly coupled compared to the others.

In this case, there are comparatively smaller differences overall between the results of different accuracy metrics for Optuna's and Chowdhury's [19] work in their respective datasets. The medians/means are similar and the coefficient of variation is nearly zero. The experiments did not show a difference in the metrics. The overall performance of Chowdhury's [19] work did not improve much with extra numbers of epochs. A possible occurrence of overfitting can be noticed for Chowdhury's [19] work for the 2018-split canola-0706 dataset for the MRE metric. It performed worse with higher number of epochs than Optuna (to recall, similar trend for the MRE has been noticed in Figure 4.6 for the 2018-split dataset and the potential reasons have been discussed). Although 1600 epochs were used in the experiments, Optuna rarely selected that number of epochs, instead of selecting no more than 400 epochs as the best parameter.

For the 2016-camera-1237 dataset, the median values for MAE are ∼31.70 and ∼29.20, MRE are ∼0.23 and ∼0.22, RMSE are ∼43.74 and ∼41.64, and Loss are ∼0.44 and ∼0.48 for Optuna and Chowdhury's [19] work, respectively. Since this dataset contains a mixture of sparse images and dense images in the test populations as seen in Figure 3.5a, the model had exposure to more variety during training. Therefore, with increasing number of epochs, the model's performance on different test populations has provided slightly better results for Chowdhury's [19] work compared to Optuna.

For the 2018-split-canola40-0706 dataset, the median values for MAE are ∼2.32 and ∼2.42, MRE are ∼0.16 and ∼0.42, RMSE are ∼3.44 and ∼3.33, and Loss are ∼0.10 and ∼0.11 for Optuna and Chowdhury's [19] work, respectively. This dataset mostly contains sparse images as seen in Figure 3.5b, so there might had been less exposure of the model to dense images while training. Therefore, a wide IQR spread and outliers are noticeable for the MAE metric for Optuna and Chowdhury respectively; the MRE is better for Optuna; the corresponding median values for RMSE and Loss metrics are very close for both of the works.

For the 2016-other-cameras dataset, the median values for MAE are ∼40.37 and ∼38.81, MRE are ∼0.41

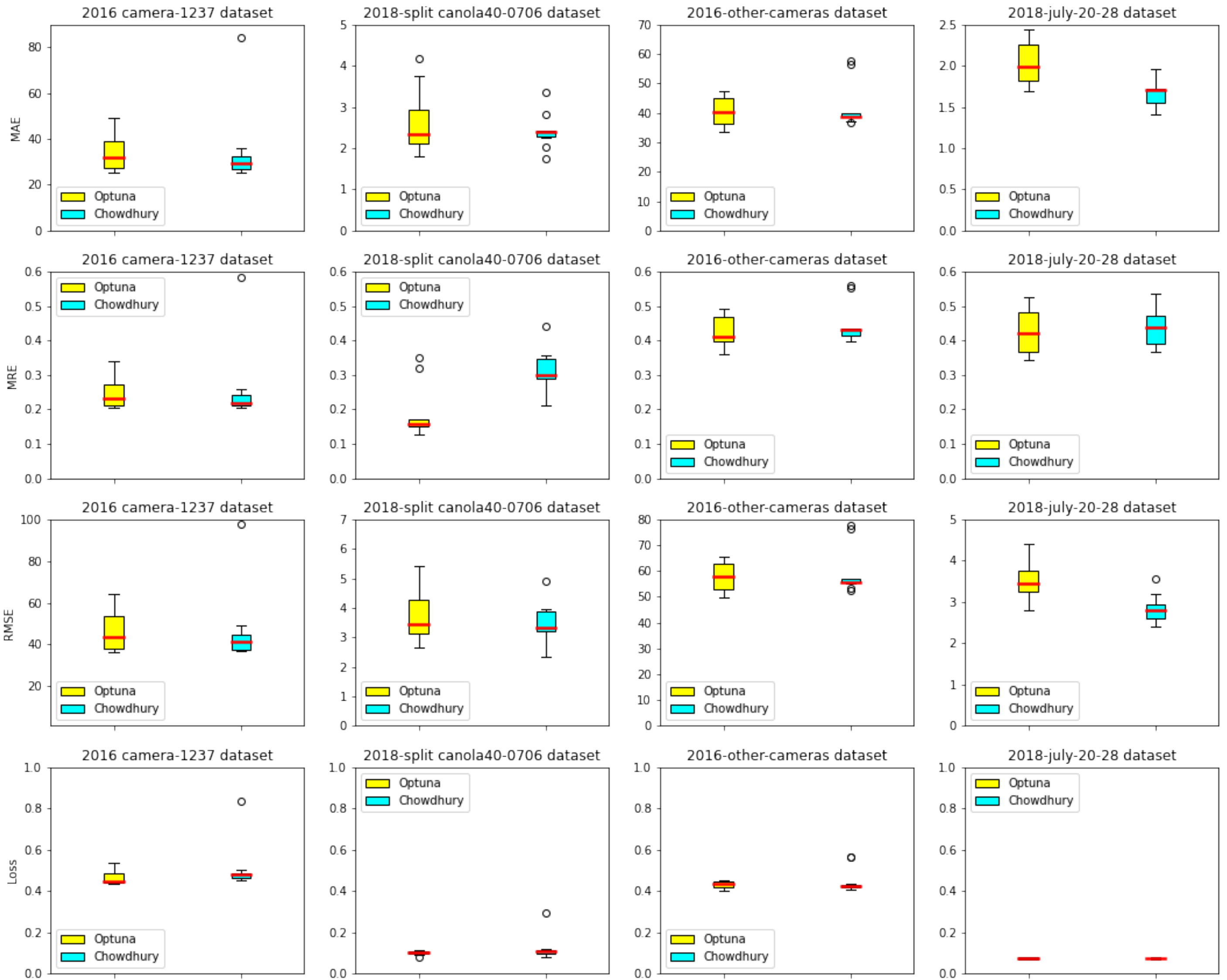


Figure 4.8: Different population experiments (Optuna vs. Chowdhury [19])

and ∼0.43, RMSE are ∼58.04 and ∼58.76, and Loss are ∼0.44 and ∼0.42 for Optuna and Chowdhury's [19] work, respectively. Like 2016-camera-1237 dataset, this dataset contains diversity in terms of sparse and dense flowers as well as seen in Figure 3.5c. Hence, the results are slightly better for Chowdhury's [19] work with increasing number of epochs than Optuna.

For the 2018-july-20-28 dataset, the median values for MAE are ∼1.98 and ∼1.71, MRE is ∼0.42 and ∼0.43, RMSE is ∼3.45 and ∼2.81, and Loss is ∼0.08 and ∼0.07 for Optuna and Chowdhury's [19] work, respectively. This dataset has mostly sparse flowers images as seen in Figure 3.5d. Therefore, it might have happened that the model had less exposure to variety of images which led to wider spreads of the metrics (except the Loss) for both Optuna and Chowdhury's [19] work.

As observed from the histogram distributions of the test populations in Figure 3.5b and 3.5d, they mainly have images with sparse flowers and few dense images. Therefore, the results on test populations for the 2018-split-canola40-0706 dataset and the 2018-july-20-28 dataset provided better accuracy scores (low loss values) than compared to the 2016-camera-1237 dataset and the 2016-other-cameras dataset. Certainly, the model is prone to make larger absolute errors if it would have to make predictions for dense flowers compared to predicting sparse flowers. Therefore, while testing on the latter two datasets, the error scores are naturally higher.

The MAE, the loss and the RMSE values of the 2018-split canola40-0706 dataset and the 2018-july-20-28 dataset are comparatively smaller compared to other two datasets. The reason is that with the fairly large numbers of sparse images, these values would be small. The loss is small, because the density map is mostly 0. The RMSE is small, because squaring the small MAE values (below 1) makes things even smaller. The MRE has wider spread but as explained previously, it does not require to necessarily follow the trend of the MAE and the RMSE.

However, the error scores of all the performance metrics have overall increased a lot for the different populations compared to the performance on the same populations. This is because the model would supposedly do better with the increasing numbers of epochs while predicting the similar test samples than compared to the unseen test samples from a different camera-day. Neither Optuna nor Chowdhury's [19] parameters have been able to generate a model that performs better on the different (unseen) test populations compared to its performance on the same populations. This is an indication that the models overfit somewhat.

Overall, it is apparent that almost equal test results for different accuracy metrics have been generated using fewer resources by this work if compared to Chowdhury's [19] work. The highest numbers of epochs used for the experiments in this work was 1600, whereas the default values of Chowdhury's [19] work was 3000. This indicates that resource (e.g., CPU/GPU processing power, time, memory usage etc.) utilization has been reduced using Optuna-suggested hyperparameter values compared to Chowdhury's parameters.

4.6 Summary

In this chapter, the Optuna-driven experiments on four different datasets of the Flower Counter application and their analysis have been presented. Experiments started with investigating the impact of individual hyperparameter on the pixel-wise validation training loss values. After that, the combined impact of the three hyperparameters were investigated for lower vs. higher number of epochs. The results showed that the hyperparameter batch size does not have distinct individual impact on various datasets, whereas, the model provided lower error scores for comparatively lower learning rates and higher number of epochs. Apart from some inconsistencies, higher number of epochs with lower learning rates and different batch sizes performed better than the similar experiments with the lower number of epochs when the combined impact of the hyperparameters was assessed. For the sake of better handling the inconsistencies, more complex objective functions could be implemented to ensure the stability and yield consistent results.

After experimenting on the validation datasets, the Optuna-suggested better hyperparameter values were derived from the corresponding studies. More experiments were conducted using those values for evaluating the generalization capability using the hold-out test populations from the same camera-day datasets, as well as test populations from different camera-days. The results of the experiments using Optuna vs. Chowdhury's [19] suggested parameters were compared on the Flower Counter application. The potential reasoning and the analysis of the results were presented according to the distributions of their respective training-test datasets. Though for the same populations Chowdhury's [19] work produced comparatively better accuracy scores, almost equal results were derived for the different populations overall. Therefore, it can be implied that, the model is likely to learn better with more epochs for the populations from the same datasets. However, the model seems to perform almost equally well using the Optuna-suggested hyperparameter combinations for the populations from similar but different datasets with the potential of resource-optimization as observed in the latter case.

5 Conclusion and Future Work

This chapter starts with the summary of the results from Chapter 4. Based on the results, necessary conclusions about the observations and findings have been presented according to the research questions mentioned in Chapter 1. Following that, the major contributions of this work is mentioned. At the end, some possible future scopes of this work is pointed out.

5.1 Summary

Based on the results executed so far, the findings and observation according to the research questions can be summarized as follows:

1. Initially, the effects of all the hyperparameters has been assessed individually with the help of Optuna, where for different datasets, batch size did not appear to have strong individual impact for different values. The other two hyperparameters, learning rate and number of epochs, comparatively showed to have noticeable impacts within given search ranges.

   For determining the suitable search space for Optuna, different ranges of values were provided initially. Based on the comparative results of the experiments for those ranges, it was expanded gradually and decided upon which range provided good results (e.g. two ranges of epoch numbers were provided; at first, the experiments were conducted using lower numbers, then based on the performance observations, the higher numbers were considered). That means, top-down approach was followed here. The bottom-up approach would have led to otherwise: starting with a broader search space and then narrowing it down accordingly.

   Therefore, based on the nature of the application, various ranges of hyperparameter values can be experimented using either top-down approach or bottom-up approach on Optuna and via observation of its impact on the accuracy metrics, potential stable regions can be chosen for the considered model.

2. Regarding the suggestions about the best observed hyperparameter configurations by Optuna, it has been quite helpful in pointing out towards a direction for making choices about potential good configurations about the considered hyperparameters, such as it preferred the higher number of epochs (at least, starting from or after 200 epochs depending on the datasets) which yielded better stability compared to the lower number of epochs; it could recommend that the points are more dense in the low to mid ranges for learning rate (e.g., upto 6.0×10^{-5}) which yielded better training and stability

than higher learning rate etc.

In terms of accuracy score enhancement, it has not been able to yield better results for the same test populations compared to the previous work. The reason is that, it is very obvious for the model to learn and/or memorize the characteristics of the dataset better with the increasing number of epochs. However, their median values were almost in the same ranges while testing on different populations. Therefore, Optuna-driven model could perform almost equally well on the unseen test populations as the previous work on different accuracy metrics, that again, using fewer resources.

3. The results of the experiments are not presumably reproducible since everytime a different study is started afresh and produces variable results based on the randomly shuffled training dataset it gets; thus, the best hyperparameter configuration values received at the end of each study and the findings regarding the combinations in each study, are not guaranteed to reproduce the same results in the next studies. It has been observed that the regions were not stable for the results; perhaps, higher level of complex objective function for defining and/or restricting the search range should be implemented to address this issue.

4. Performance variability has been noticed for Optuna in the conducted experiments. For some datasets, it could comparatively yield better stability, for others, it could not (e.g., in the combined hyperparameters impact-checking based experiments, there were steady lines for 2018-july-2-23 dataset, for which did not train); while testing on datasets, it showed very low error scores and very small IQR spreads for some datasets (e.g., for testing on 2018-split canola40-0706 and 2018-july-20-28 dataset), but for others it had comparatively longer IQR spreads (e.g., testing on 2016 camera-1237 dataset). Therefore, it could be said that, based on the diverse characteristics (e.g., sparse flowers vs. dense flowers etc.) of different datasets, the performance of Optuna also varies.

   To mention, the detailed analysis of the results according to the characteristics of individual datasets is not the part of this study.

5.2 Thesis contribution

This work has been conducted as a case study for demonstrating the usability of an Automated HPO, namely Optuna, for any ML/DL application. Some points about this research work's contribution could be mentioned as follows:

- This case study could certainly help the plant scientists, researchers, practitioners, enthusiasts, general users etc. for giving idea about the applicability of Automated HPO, Optuna for their ML/DL applications. Using the provided APIs offered by Optuna, the search spaces can be constructed dynamically and easily based on the type (e.g., integer, float, categorical etc.) of the hyperparamters. By conducting experiments on different hyperparameters, the more influential ones can be found by Optuna too

(some hyperparameters might have more impact than the others when considered individually and/or combinedly). Therefore, the researchers could get directions about the best observed hyperparameter configurations specific to their applications, which would lead them to better tune their applications.

- The scores yielded from different accuracy metrics from Optuna-driven experiments could give the plant scientists added knowledge about the efficiency of the Flower Counter application. The scientists would be able to tune the model accordingly (even, treating the application and the framework as black-boxes) using the suggested best observed hyperparameter configurations and apply the tuned model for flower-counting as well as, gain more knowledge of flower yields and general hardiness. This technique could be applicable for other similar CNN applications also.

- The process of gradual exploration of the search space (the approach of exploration should be chosen based on the nature of the application) for finding among the best observed hyperparameters by Optuna is indeed effective. Though more research is necessary for confirming better stability, suggestions about possible hyperparameters among the best observed values have been received from Optuna based on the progressive exploration. Following the suggestions of those hyperparameters has shown the potential of less resource consumption over the fixed set of values of hyperparameters.

- Furthermore, potential direction towards better stability and useful suggestions from Optuna would be able to considerably reduce manual efforts and resources for deciding upon suitable hyperparameter configurations. Also, potential indication towards any particular dataset, that has not been able to produce very good stability, is indicated by Optuna; therefore, Optuna has certainly been helpful to point out towards the dataset that requires further analysis.

5.3 Potential future scope of the work

There could be certain potential scopes to be explored based on the findings of this work. Some areas which could be worthwhile of exploring as an extension of this work or future work, are mentioned as follows:

A single objective function has been chosen for Optuna for this work. There could be other potential options (e.g., minimizing or maximizing other accuracy metrics, using more complex objective functions to yield more stability and consistency etc.), that could be explored in the future. Also, the impact of the increasing number of trials for finding better stability of the hyperparameters could be examined.

The potential impacts of the characteristics of the datasets on Optuna's performance could be analyzed in detail. As per the observations from the results, datasets can be categorized according to sparse flowers vs. dense flowers and their individual impacts can be assessed and compared accordingly. Also, different binning techniques could be applied for sampling the images and the associated impacts on Optuna's performance could be evaluated.

Optuna could be deployed using distributed systems for achieving scalability and further analysis could

be conducted on performance gain etc. Since efficient pruning techniques are offered by Optuna, different pruning techniques could be applied in order to check if there is any performance improvement in terms of saving time, memory usage, CPU/GPU processing power enhancement etc.

Three hyperparameters have been considered in this work; there are many other hyperparameters (such as the number of hidden layers, dropout rate etc.), whose impact could be analyzed. After that, the performance of Optuna and the other HPO frameworks on the Flower Counter application could be compared in order to assess more about Optuna's efficiency.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation OSDI*, pages 265–283, Savannah, GA, November 2016.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, Anchorage, AK, August 2019.

[3] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *Proceedings of the 2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, Antalya, Turkey, August 2017.

[4] Francesco Archetti and Antonio Candelieri. *The Surrogate Model*, pages 37–56. Springer International Publishing, Cham, 2019.

[5] Divya Arora, Mehak Garg, and Megha Gupta. Diving deep in deep convolutional neural network. In *Proceedings of the 2020 2nd International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pages 749–751, Greater Noida, India, December 2020.

[6] Nii O. Attoh-Okine. Analysis of learning rate and momentum term in backpropagation neural network algorithm trained to predict pavement performance. *Advances in engineering software*, 30(4):291–302, 1999.

[7] Thomas Back. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996.

[8] Qinghai Bai. Analysis of particle swarm optimization algorithm. *Computer and Information Science*, 3(1):180, 2010.

[9] Lukáš Bajer and Martin Holeňa. Surrogate model for mixed-variables evolutionary optimization based on GLM and RBF networks. In *Proceedings of the SOFSEM 2013: Theory and Practice of Computer Science*, pages 481–490, Špindlerův Mlýn, Czech Republic, January 2013.

[10] Rémi Bardenet, Mátyás Brendel, Balázs Kégl, and Michele Sebag. Collaborative hyperparameter tuning. In *Proceedings of the International Conference on Machine Learning*, pages 199–207, Atlanta, GA, June 2013.

[11] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[12] Federico Bergenti, Agostino Poggi, and Michele Tomaiuolo. An actor based software framework for scalable applications. In *Proceedings of the International Conference on Internet and Distributed Computing Systems*, pages 26–35, Calabria, Italy, September 2014.

[13] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, page 2546–2554, Granada, Spain, December 2011.

[14] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13:281–305, February 2012.

[15] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, 2015.

[16] Manuel Blum and Martin A Riedmiller. Optimization of Gaussian Process Hyperparameters using Rprop. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*, pages 339–344, Chicago, IL, April 2013.

[17] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. Mongodb vs oracle–database comparison. In *Proceedings of the 2012 Third International Conference on Emerging Intelligent Data and Web Technologies*, pages 330–335, Bucharest, Romania, September 2012.

[18] Tianfeng Chai and Roland R Draxler. Root mean square error (RMSE) or mean absolute error (MAE). *Geoscientific Model Development Discussions*, 7(1):1525–1534, 2014.

[19] Mohamed Rashid Chowdhury. Scaling a convolutional neural network based Flower counting application in a distributed GPU cluster. Master's thesis, University of Saskatchewan, 2019.

[20] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning. In *Proceedings of the MIC 2015 11th Metaheuristics International Conference*, pages 14-1 -- 14-4, Agadir, Morocco, June 2015.

[21] Sibanjan Das and Umit Mert Cakmak. *Hands-On Automated Machine Learning: A beginner's guide to building automated machine learning systems using AutoML and Python.* Packt Publishing Ltd, 2018.

[22] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 3460–3468, Buenos Aires, Argentina, 2015.

[23] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an Empirical Foundation for Assessing Bayesian Optimization of Hyperparameters. In *Proceedings of the NIPS Workshop on Bayesian Optimization in Theory and Practice*, pages 1–5, Lake Tahoe, NV, December 2013.

[24] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1437–1446, Stockholm, Sweden, July 2018.

[25] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing Bayesian Hyperparameter Optimization via Meta-Learning. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, pages 1–8, Austin, Texas, January 2015.

[26] Adrian Cătălin Florea and Răzvan Andonie. A dynamic early stopping criterion for random search in SVM hyperparameter optimization. In *Artificial Intelligence Applications and Innovations*, pages 168–180, Cham, 2018. Springer International Publishing.

[27] Engy Fouda. SAS visual statistics: Viya. In *Learn Data Science Using SAS Studio*, pages 187–209. Springer, 2020.

[28] Michael A. Gelbart, Jasper Snoek, and Ryan P. Adams. Bayesian optimization with unknown constraints. In *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*, UAI'14, page 250–259, Arlington, Virginia, July 2014.

[29] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36, Brisbane, Australia, May 2001.

[30] David E Goldberg. Genetic Algorithms in Search. *Optimization, and MachineLearning*, 1989.

[31] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495, Halifax, Canada, August 2017.

[32] Isabelle Guyon, Lisheng Sun-Hosoya, Marc Boullé, Hugo Jair Escalante, Sergio Escalera, Zhengying Liu, Damir Jajetic, Bisakha Ray, Mehreen Saeed, Michèle Sebag, Alexander Statnikov, Wei-Wei Tu, and Evelyne Viegas. Analysis of the AutoML challenge series 2015–2018. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *Automated Machine Learning*, pages 177–219. Springer, 2019.

[33] Ji-Hoon Han, Dong-Jin Choi, Sang-Uk Park, and Sun-Ki Hong. Hyperparameter optimization using a genetic algorithm considering verification time in a convolutional neural network. *Journal of Electrical Engineering & Technology*, 15(2):721–726, 2020.

[34] Nikolaus Hansen. *The CMA Evolution Strategy: A Comparing Review*, pages 75–102. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[35] Peter Harrington. *Machine Learning in Action*. Simon and Schuster, 2012.

[36] Elad Hazan, Adam Klivans, and Yang Yuan. Hyperparameter optimization: a spectral approach. In *Proceedings of the International Conference on Learning Representations*, pages 1–18, Vancouver, Canada, May 2018.

[37] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4340–4349, Long Beach, CA, June 2019.

[38] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: Closing the generalization gap in large batch training of neural networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1729–1739, Long Beach, California, December 2017.

[39] Holger Hoos and Kevin Leyton-Brown. An Efficient Approach for Assessing Hyperparameter Importance. In *Proceedings of the International Conference on Machine Learning*, pages 754–762, Bejing, China, June 2014.

[40] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An Evaluation of Sequential Model-Based Optimization for Expensive Blackbox Functions. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, pages 1209–1216, New York, NY, July 2013.

[41] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proceedings of the International Conference on Learning and Intelligent Optimization*, pages 507–523, Trento, Italy, January 2011.

[42] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated Machine Learning*. Springer, 2019.

[43] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018.

[44] Christian Igel, Nikolaus Hansen, and Stefan Roth. Covariance Matrix Adaptation for Multi-objective Optimization. *Evolutionary Computation*, 15(1):1–28, 03 2007.

[45] Robert A Jacobs. Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307, 1988.

[46] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, hrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *CoRR*, abs/1711.09846:1–21, 2017.

[47] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956, Anchorage, AK, August 2019.

[48] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, pages 1–15, San Diego, CA, May 2015.

[49] Patrick Koch, Oleg Golovidov, Steven Gardner, Brett Wujek, Joshua Griffin, and Yan Xu. Autotune: A Derivative-free Optimization Framework for Hyperparameter Tuning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 443–452, London, United Kingdom, 2018.

[50] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-WEKA 2.0: Automatic Model Selection and Hyperparameter Optimization in WEKA. *The Journal of Machine Learning Research*, 18(1):826–830, 2017.

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, pages 1097–1105. Curran Associates, Inc., 2012.

[52] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[53] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[54] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[55] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *Proceedings of the Machine Learning and Systems*, volume 2, pages 230–246, Virtual, 2020.

[56] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel Bandit-Based Approach to Hyperparameter Optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[57] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph Gonzalez, and Ion Stoica. Tune: A Research Platform for Distributed Model Selection and Training. In *Proceedings of the 35th International Conference on Machine Learning AutoML Workshop*, pages 1–8, Stockholm, Sweden, July 2018.

[58] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, Munich, Germany, 2018.

[59] Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. Auptimizer–An Extensible, Open-Source framework for Hyperparameter Tuning. In *Proceedings of the 2019 IEEE International Conference on Big Data (Big Data)*, pages 115–123, Los Angeles, CA, December 2019.

[60] Xuliang Liu, Daolun Li, Jinghai Yang, Wenshu Zha, Ziqi Zhou, Liping Gao, and Jiahang Han. Automatic well test interpretation based on convolutional neural network for infinite reservoir. *Journal of Petroleum Science and Engineering*, 195:107618, 2020.

[61] Ilya Loshchilov and Frank Hutter. Cma-es fpr hyperparameter optimization of deep neural networks. In *Proceedings of the ICLR Worskshops Posters*, pages 1–8, San Juan, Puerto Rico, May 2016.

[62] Gang Luo. A Review of Automatic Selection Methods for Machine Learning Algorithms and Hyper-parameter Values. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 5(1):18, 2016.

[63] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43 – 58, 2016.

[64] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the International Conference on Machine Learning*, pages 2113–2122, Lille, France, July 2015.

[65] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *ArXiv*, abs/1804.07612:1–18, 2018.

[66] Robert McGill, John W Tukey, and Wayne A Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.

[67] Umberto Michelucci. *Advanced Applied Deep Learning: Convolutional Neural Networks and Object Detection*. Apress, 1st edition, 2019.

[68] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association*, pages 1045–1048, Makuhari, Chiba, Japan, September 2010.

[69] Frank Millstein. *Convolutional Neural Networks In Python: Beginner's Guide To Convolutional Neural Networks In Python*. CreateSpace Independent Publishing Platform, North Charleston, SC, 2018.

[70] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.

[71] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[72] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation OSDI*, pages 561–577, Carlsbad, CA, October 2018.

[73] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-Serving: Flexible, High-Performance ML Serving. In *Proceedings of the Workshop on ML Systems at NIPS*, Long Beach, CA, December 2017.

[74] Yu-Hsiang Peng, Chia-Chuan Chuang, Zhou-Jin Wu, Chia-Wei Chou, Hui-Shan Chen, Ting-Chia Chang, Yi-Lun Pan, Hsin-Tien Cheng, Chih-Chi Chung, and Ken-Yu Lin. Machine learning hyper-parameter fine tuning service on dynamic cloud resource allocation system-taking heart sounds as an example. In *Proceedings of the International Symposium on Big Data and Artificial Intelligence*, pages 22--28, Hong Kong, Hong Kong, December 2018.

[75] V. P. Plagianakos, G. D. Magoulas, and M. N. Vrahatis. *Learning Rate Adaptation in Stochastic Gradient Descent*, pages 433–444. Springer US, Boston, MA, 2001.

[76] Philipp Probst, Marvin N Wright, and Anne-Laure Boulesteix. Hyperparameters and tuning strategies for random forest. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(3):e1301, 2019.

[77] Russell Reed and Robert J MarksII. *Neural smithing: supervised learning in feedforward artificial neural networks*. MIT Press, 1999.

[78] Peter Riegler-Nurscher, Johann Prankl, Thomas Bauer, Peter Strauss, and Heinrich Prankl. A machine learning approach for pixel wise classification of residue and vegetation cover under field conditions. *Biosystems Engineering*, 169:188–198, 2018.

[79] Luis Miguel Rios and Nikolaos V Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.

[80] Mohit Sewak, Md Rezaul Karim, and Pradeep Pujari. *Practical Convolutional Neural Networks: Implement Advanced Deep Learning Models using Python*. Packt Publishing Ltd, 2018.

[81] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning - From Theory to Algorithms*. Cambridge University Press, 2014.

[82] Simon J Sheather. Density estimation. *Statistical science*, pages 588–597, 2004.

[83] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[84] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT Press, 2018.

[85] Vladimir Svetnik, Andy Liaw, Christopher Tong, J. Christopher Culberson, Robert P. Sheridan, and Bradley P. Feuston. Random forest: A classification and regression tool for compound classification and qsar modeling. *Journal of Chemical Information and Computer Sciences*, 43(6):1947–1958, 2003.

[86] H Taud and JF Mas. Multilayer perceptron MLP. In *Geomatic Approaches for Modeling Land Change Scenarios*, pages 451–455. Springer, 2018.

[87] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 847–855, Chicago, IL, August 2013.

[88] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2002–2011, Anchorage, AK, August 2019.

[89] Joannes Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *Proceedings of the 16th European Conference on Machine Learning*, pages 437–448, Porto, Portugal, October 2005.

[90] Rachel Ward, Xiaoxia Wu, and Leon Bottou. AdaGrad stepsizes: Sharp convergence over nonconvex landscapes. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97, pages 6677–6686, Long Beach, CA, June 2019.

[91] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[92] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Hyperparameter search space pruning–a new component for sequential model-based hyperparameter optimization. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 104–119, Riva del Garda, Italy, September 2015.

[93] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Scalable gaussian process-based transfer surrogates for hyperparameter optimization. *Machine Learning*, 107(1):43–78, 2018.

[94] Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mane, Doug Fritz, Dilip Krishnan, Fernanda B Viégas, and Martin Wattenberg. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE Transactions on Visualization and Computer graphics*, 24(1):1–12, 2017.

[95] Xing-xing Wu and Jin-guo Liu. A new early stopping algorithm for improving neural network generalization. In *Proceedings of the 2009 Second International Conference on Intelligent Computation Technology and Automation*, volume 1, pages 15–18, Changsha, China, October 2009.

[96] Qi Xu, Ming Zhang, Zonghua Gu, and Gang Pan. Overfitting remedy by sparsifying regularization on fully-connected layers of cnns. *Neurocomputing*, 328:69–74, 2019.

[97] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020.

[98] Steven R Young, Derek C Rose, Thomas P Karnowski, Seung-Hwan Lim, and Robert M Patton. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-performance Computing Environments*, pages 1–5, New York, NY, 2015.

[99] Xiao-Hu Yu, Guo-An Chen, and Shi-Xin Cheng. Dynamic learning rate optimization of the backpropagation algorithm. *IEEE Transactions on Neural Networks*, 6(3):669–677, 1995.

[100] Cong Zhang, Hongsheng Li, Xiaogang Wang, and Xiaokang Yang. Cross-Scene Crowd Counting via Deep Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 833–841, Boston, MA, June 2015.

[101] Yingying Zhang, Desen Zhou, Siqin Chen, Shenghua Gao, and Yi Ma. Single-Image Crowd Counting via Multi-Column Convolutional Neural Network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 589–597, Las Vegas, NV, June 2016.

[102] Zijun Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–2, Banff, Canada, June 2018.

[103] Yuan Zheng, Xiaogang Fu, and Yanwen Xuan. Data-driven optimization based on random forest surrogate. In *Proceedings of the IEEE 6th International Conference on Systems and Informatics (ICSAI)*, pages 487–491, Shanghai, China, November 2019.

[104] Jinan Zhou, Andrey Velichkevich, Kirill Prosvirov, Anubhav Garg, Yuji Oshima, and Debo Dutta. Katib: A Distributed General Automl Platform on Kubernetes. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*, pages 55–57, Santa Clara, CA, May 2019.

[105] Michael Zhu and Suyog Gupta. To prune, or not to prune: Exploring the efficacy of pruning for model compression. In *Workshop Track Proceedings of the 6th International Conference on Learning Representations*, pages 1–11, Vancouver, Canada, May 2018.

[106] Fangyu Zou, Li Shen, Zequn Jie, Weizhong Zhang, and Wei Liu. A sufficient condition for convergences of Adam and RMSProp. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11127–11135, Long Beach, CA, June 2019.