Hyperparameter optimization using Bayesian optimization on transfer learning for medical image classification

Rune Johan Borgli

Thesis submitted for the degree of
Master of science in Informatics: Programming and Networks
60 credits

Institutt for informatikk
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

Hyperparameter optimization using Bayesian optimization on transfer learning for medical image classification

Rune Johan Borgli

© 2018 Rune Johan Borgli

Hyperparameter optimization using Bayesian optimization on transfer learning for medical image classification

http://www.duo.uio.no/

Abstract

The field of medicine has a history of adopting new technology. Video equipment and sensors are used to visualize areas of interest in the human allowing for doctors to make diagnoses based on imagery observations. However, the detection rate of the doctors towards diseases and abnormalities is heavily dependent on the experience and state of mind of the doctor doing the examination. Computer-aided detection systems are systems designed to aid the doctor in improving the detection rate, and they are using or experimenting with machine learning.

Deep convolutional neural networks, a type of machine learning, are shown to be highly efficient at image detection, classification, and analysis [103]. However, these networks require large datasets to train properly. Transfer learning is a training technique where we use a pre-trained machine learning model and transfer some of the attained knowledge from other application domains over to a new model. This way, we can use small datasets and train a model in much shorter time. In this respect, transfer learning works fine but has many configurations called hyperparameters which are often not optimized.

Our work aims to address the lack of automatic hyperparameter optimization for transfer learning by experiments utilizing a known hyperparameter optimization method and creating a system for running those experiments. We decided to focus on the field of gastroenterology by utilizing two publicly available datasets showing images from the gastrointestinal tract. We used a specific transfer learning method and chose hyperparameters suitable for automatic optimization. The optimization method we chose was Bayesian optimization because of its reputation for being one of the best methods for hyperparameter optimization [45, 70]. However, Bayesian optimization has its own hyperparameters, and there are also different versions of Bayesian optimization. We chose to limit the thesis, so we use standard Bayesian optimization with standard parameters.

We created a system for running automatic experiments of three different hyperparameter optimization strategies. With the system, we ran a set of experiments for each dataset. Between the strategies, one was successful in achieving a high validation accuracy, while the others were considered failures. Compared to baselines, our best models was around 10% better. With these experiments, we demonstrated that automatic hyperparameter optimization is an effective strategy for increasing performance in transfer learning and that the best hyperparameters are nontrivial to select manually.

Contents

1 **Introduction** **1**
- 1.1 Background and Motivation 1
- 1.2 Problem Statement 4
- 1.3 Scope and Limitations 4
- 1.4 Research Method 5
- 1.5 Outline 7

2 **Background** **9**
- 2.1 The Medical Background 9
  - 2.1.1 Gastroenterology 9
  - 2.1.2 Medical data 11
  - 2.1.3 Medical Multimedia Systems using Handcrafted Image Features 15
  - 2.1.4 Related work on Deep Convolutional Neural Networks and Transfer Learning 17
  - 2.1.5 Medical Multimedia Systems using Deep Convolutional Neural Networks and Transfer Learning 20
  - 2.1.6 Summary 20
- 2.2 Machine Learning 21
  - 2.2.1 Deep Convolutional Neural Networks 22
  - 2.2.2 Transfer learning 24
  - 2.2.3 Hyperparameter optimization 25
- 2.3 Summary 27

3 **Methodology** **29**
- 3.1 Datasets 29
  - 3.1.1 Kvasir 29
  - 3.1.2 Nerthus 32
- 3.2 Metrics 34
- 3.3 Hyperparameters 36
  - 3.3.1 Pre-trained Model 36
  - 3.3.2 Model Optimization Function 37
  - 3.3.3 Learning Rate 38
  - 3.3.4 Delimiting Layer 39
- 3.4 Proposed System 41
  - 3.4.1 Libraries 41
  - 3.4.2 System Description 45

3.4.3 Test Suite 47
3.4.4 Optimization Strategies 48
3.4.5 Bayesian Optimization 50
3.4.6 Model Setup 53
3.4.7 Classification Block Training 54
3.4.8 Fine-tuning 56
3.4.9 Layer optimization 57
3.5 Nonconvergence filtering 57
3.6 Summary 61

4 Experiments **63**
4.1 Design of Experiments 63
4.2 Results and Discussion 66
4.3 Results for Kvasir 66
4.3.1 Shared Hyperparameters Optimization Strategy . . . 70
4.3.2 Separate Hyperparameters Optimization Strategy . . 82
4.3.3 Separate Optimizers Optimization Strategy 91
4.3.4 Best Trained Model 104
4.4 Results for Nerthus 108
4.4.1 Best Trained Model 112
4.5 Summary 114

5 Conclusion **119**
5.1 Summary and Main Contributions 119
5.2 Future Work 122
5.3 Final Remarks 123

A Winning Poster from Autonomy Day 2018 **139**

B Plots From the Nerthus Optimization **141**
B.1 Shared Hyperparameters Optimization Strategy 142
B.1.1 Model Optimization 144
B.1.2 Layer Optimization 146
B.2 Separate Hyperparameters Optimization Strategy 149
B.2.1 Model Optimization 151
B.2.2 Layer Optimization 153
B.3 Separate Optimizations Optimization Strategy 156
B.3.1 Classification Block Optimization 157
B.3.2 Fine-tuning Optimization 159
B.3.3 Layer Optimization 162

C Permission To Use Illustrations **165**

List of Figures

2.1 Illustration of the lower grastrointestinal anatomy. The whole gastrointestinal tract is illustrated, but only the lower part is highlighted. 10
2.2 Illustration of the colon with pointers to anatomical landmarks. 12
2.3 Illustration of a colonoscopy procedure. The doctor operates a flexible tune with a camera attached. The tube is then passed through the anus and provides a video of the insides of the colon. 13
2.4 Image of a polyp from the Kvasir dataset 14
2.5 Illustration of the colon with polyps growing inside. 14

3.1 Images from each class in the Kvasir dataset [98]. 30
3.2 Images from each class of the Nerthus dataset [97] 33
3.3 Schematic diagram of Inception-ResNet-v2 40
3.4 Overview of the TensorFlow architecture 42
3.5 Overview of the proposed system flow. 46
3.6 Separate Optimization Strategy 49
3.7 Bayesian Optimization for shared hyperparameters in our system 51
3.8 Overview over the last optimization step for finding the delimting layer 58

4.1 Plot of full test on the Kvasir dataset in epochs 66
4.2 Plot of full test on the Kvasir dataset in time 67
4.3 Shared hyperparameters optimization strategy run in steps . 70
4.4 Shared hyperparameters optimization strategy run in time . 71
4.5 The shared hyperparameters optimization strategy's model optimization run for each epoch 73
4.6 Convergence plot of the shared hyperparameters optimization strategy's model optimization run 74
4.7 The shared hyperparameters optimization strategy's layer optimization run for each epoch 78
4.8 Convergence plots for the shared hyperparameter optimization strategy's layer optimization run 80
4.9 Gaussian Process surrogate model and Expected Improvement acquisition function for the shared hyperparameter optimization strategy's layer optimization 81

4.10 The separate hyperparameters optimization strategy's plot of training runs for each epoch 82
4.11 The separate hyperparameters optimization strategy's plot of training runs for time 83
4.12 The separate hyperparameters optimization strategy's plot of model optimization training runs for each epoch 84
4.13 The separate hyperparameters optimization strategy's plot of layer optimization training runs for each epoch 86
4.14 The separate hyperparameters optimization strategy's plot of layer optimization convergence 88
4.15 Gaussian Process surrogate model and Expected Improvement acquisition function for the separate hyperparameters optimization strategy's layer optimization 89
4.16 The separate optimizations optimization strategy's plot of training runs for each epoch 91
4.17 The separate optimizations optimization strategy's plot of training runs for time 92
4.18 The separate optimizations optimization strategy's classification block optimization run for each epoch 94
4.19 Convergence plot of the separate optimizations optimization strategy's classification block optimization run 95
4.20 The separate optimizations optimization strategy's fine-tuning optimization run for each epoch 97
4.21 Convergence plot of the separate optimizations optimization strategy's fine-tuning optimization run 98
4.22 Gaussian Process surrogate model and Expected Improvement acquisition function for the separate optimizations optimization strategy's fine-tuning optimization 100
4.23 The separate optimizations optimization strategy's layer optimization run for each epoch 101
4.24 Convergence plot of the separate optimizations optimization strategy's layer optimization run 102
4.25 Gaussian Process surrogate model and Expected Improvement acquisition function for the separate optimizations optimization strategy's layer optimization 103
4.26 Confusion matrix for best model trained on Kvasir 107
4.27 Similar images from two different classes in the Kvasir dataset [98] 109
4.28 Plot of all training runs done on the Nerthus dataset for each epoch [97] 110
4.29 Plot of all training runs done on the Nerthus dataset for each time stamp 111
4.30 Confusion matrix for best model trained on Nerthus 113

A.1 Poster winning the Best Poster Award from Autonomy Day 2018 140

B.1 Shared hyperparameters optimization strategy run in steps . 142

B.2 Shared hyperparameters optimization strategy run in time . 143
B.3 The shared hyperparameters optimization strategy's model optimization run for each epoch 144
B.4 Convergence plot of the shared hyperparameters optimization strategy's model optimization run 145
B.5 The shared hyperparameters optimization strategy's layer optimization run for each epoch 146
B.6 Convergence plots for the shared hyperparameter optimization strategy's layer optimization run 147
B.7 Gaussian Process surrogate model and Expected Improvement acquisition function for the shared hyperparameter optimization strategy's layer optimization 148
B.8 The separate hyperparameters optimization strategy's plot of training runs for each epoch 149
B.9 The separate hyperparameters optimization strategy's plot of training runs for time 150
B.10 The separate hyperparameters optimization strategy's plot of model optimization training runs for each epoch 151
B.11 Convergence plots for the separate hyperparameter optimization strategy's model optimization run 152
B.12 The separate hyperparameters optimization strategy's plot of layer optimization training runs for each epoch 153
B.13 The separate hyperparameters optimization strategy's plot of layer optimization convergence 154
B.14 Gaussian Process surrogate model and Expected Improvement acquisition function for the separate hyperparameters optimization strategy's layer optimization 155
B.15 The separate optimizations optimization strategy's plot of training runs for each epoch 156
B.16 The separate optimizations optimization strategy's plot of training runs for time 156
B.17 The separate optimizations optimization strategy's classification block optimization run for each epoch 157
B.18 Convergence plot of the separate optimizations optimization strategy's classification block optimization run 158
B.19 The separate optimizations optimization strategy's fine-tuning optimization run for each epoch 159
B.20 Convergence plot of the separate optimizations optimization strategy's fine-tuning optimization run 160
B.21 Gaussian Process surrogate model and Expected Improvement acquisition function for the separate optimizations optimization strategy's fine-tuning optimization 161
B.22 The separate optimizations optimization strategy's layer optimization run for each epoch 162
B.23 Convergence plot of the separate optimizations optimization strategy's layer optimization run 163

B.24 Gaussian Process surrogate model and Expected Improvement acquisition function for the separate optimizations optimization strategy's layer optimization 164

C.1 Signed Thesis Copyright Permission Form which gives consent to use the illustrations in the background showing a colonoscopy and gastrointestinal anatomy. 165

List of Tables

3.1 Table of hyperparameter bounds 36
3.2 Table of pre-trained CNN models used as hyperparameters in this thesis 37
3.3 Table of the optimizers used as hyperparameters in this thesis and their default learning rate. 38
3.4 Overview over important libraries 42
3.5 Hyperparameter sets for each optimization strategy. 50
3.6 List of selected relevant default parameters given to the Bayesian optimization function available in the GPyOpt library [6]. 52

4.1 Hardware specifications for our testing environment 63
4.2 Validation accuracy and hyperparameters for each Bayesian optimization iteration in the shared hyperparameters optimization strategy's model optimization 77
4.3 Validation accuracy and hyperparameters for each Bayesian optimization iteration in the shared hyperparameters optimization strategy's layer optimization 79
4.4 Validation accuracy and hyperparameters for each Bayesian optimization iteration in the separate hyperparameters optimization strategy's model optimization 85
4.5 Validation accuracy and hyperparameters for each Bayesian optimization iteration in the separate hyperparameters optimization strategy's layer optimization 88
4.6 Validation accuracy and hyperparameters for each Bayesian optimization iteration in the separate optimizations optimization strategy's classification block optimization 96
4.7 Validation accuracy and hyperparameters for each Bayesian optimization iteration in the separate optimizations optimization strategy's fine-tuning optimization 99
4.8 Validation accuracy and hyperparameters for each Bayesian optimization iteration in the separate optimizations optimization strategy's layer optimization 102
4.9 Table of hyperparameter values of the best trained model on the Kvasir dataset 104
4.10 Metrics for each class after hyperparameter optimization for the best model on the Kvasir dataset 105

4.11 Performance metrics for different methods of image classification done on the Kvasir dataset 106
4.12 Table of hyperparameter values of the best trained model on the Nerthus dataset 112
4.13 Metrics for each class after hyperparameter optimization for the best model on the Nerthus dataset. 112
4.14 Performance metrics for different methods of image classification done on the Nerthus dataset 114

Listings

3.1 Python code of callback for Keras that stops early if the monitored value is below the given threshold in a given number of epochs. We call this technique nonconvergence filtering. The monitor parameter is the value we use for evaluation, the threshold is the threshold the monitored value is required to pass, and the patience is the number of epochs the monitored value is allowed to be under the threshold before being stopped. The presented version is stripped down to only work with accuracies and not loss as that was our use case. 59

Chapter 1

Introduction

The field of medicine has many use cases for data generated from examinations, but there is a shortage of such data annotated for use by machine learning. This thesis is part of the research into using transfer learning methods for solving the problem of datasets being too small to train convolutional neural networks without losing their generality. As a use case, we chose the field of gastroenterology because datasets and previous results are publicly available to use for comparison.

1.1 Background and Motivation

New technologies are emerging every year. They push the boundaries of what is possible to achieve. That what was fiction 30 years ago is now a reality. Ever since the invention of the computer, scientists and writers alike have been imagining a future where the machines are intelligent. Today we are closing in on that goal using self-improving algorithms; so-called "Machine learning".

Machine learning is a field of computer science where the goal is to give algorithms the ability to learn. By learning, we mean algorithms that can improve their performance on a specific task, without being explicitly programmed [34]. There are several machine learning methods, and each has their specialized problem area. For example, for image classification problems, one would typically use deep convolutional neural networks [103] or handcrafted feature based approaches [81, 109]. For speech recognition, recurrent neural networks would be a good choice because of their internal state [78].

Machine learning was coined already in 1959 [120] and has been a research topic over the decades. Nevertheless, it is in the last decade machine learning has become really popular. The popularity is due to factors such as higher processing power, better algorithms, and the emergence of new concepts like deep learning and convolutional networks [33]. Much of the mass research and adoption of machine learning we see today is due to the availability of large datasets and software platforms. The software platforms enable the creation and deployment of machine learning models and make it easy to use large computational resources for training [2]. Several fields are deploying or exploring the capabilities of machine learning. One such field is the medical field where technical advances are needed to support modern medicine [18, 95, 110].

One branch of medicine where machine learning is used for image detection is in gastroenterology. Gastroenterology is the branch of medicine focused on the digestive system, including the gastrointestinal (GI) tract and its disorders. In recent years, detection of diseases in the GI tract has become an important research field due to increasing affliction rates and their economic and human cost to society. A staggering 10% of deaths in the US in 2009 were attributable to an underlying gastrointestinal cause [92]. There is good evidence in research that cancer screening programmes reduce incident and mortality rates of cancers, especially colorectal cancer (CRC) [59, 91]. In the case of CRC, colonoscopies are the golden standard. Colonoscopies are used to not only detect stages of CRC but also to prevent CRC by having polyps found and removed. However, a major disadvantage of colonoscopies is that the effect of the procedure is entirely dependent on the performance of the person conducting it [107].

Research is conducted into computer-aided detection (CAD) systems, to improve the accuracy of the procedure. These are systems designed to assist the operator of the colonoscopy in detecting abnormalities and diseases [18]. Detection is done automatically through image analysis of the images from the colonoscopy. State-of-the-art in the field of automatic image analysis for colonoscopy is using deep convolutional neural network models [96, 112]. However, these are models that require large amounts of data to be effective. Unfortunately, in the medical world of colonoscopies, this data is hard to obtain. There exist a few publicly available datasets, but they include few images. Recent datasets try to overcome this limitation [97, 98]. Transfer learning is the typical approach used by most CAD systems to make the most out of the existing data. Studies related to both transfer learning and automatic polyp detection have shown success in using transfer learning to improve results [20, 144].

Transfer learning is a machine learning method that aims to help improve the performance of machine learning models and reduce the amount of data needed for training [90, 140]. This is done through the transfer of knowledge from one task in one domain to a related task in a different domain. Imagine a model for detecting cars, but the new task is to detect trucks. By transferring knowledge gained from detecting cars, the dataset for detecting trucks can be of lesser quality and quantity and still achieve good results as the new model will have previous knowledge of how a car looks. In the case of CAD systems, models pre-trained on the ImageNet dataset are used [144]. ImageNet contains curated URLs to images belonging to different noun categories. The dataset is more significant in scale and diversity and much more accurate than any other image dataset and is used as a benchmark for applications in object recognition, image classification, and automatic object clustering [60].

In any machine learning model, the goal is to have the model train itself by changing the weights of the model. These weights are the model's parameters. However, how the model is trained is something that can be tuned. These are parameters such as which model to use, the learning rate of the model and which optimizer to use. We separate between model parameters and the parameters mentioned above by calling the latter ”hyperparameters”.

Choosing the best hyperparameters is not trivial [26]. Different problems and different datasets will have different optimal hyperparameters. There are specific configurations known from literature that might give good results [69, 143], but those parameters could still be far from the optimal configuration. Determining the hyperparameters usually requires testing out different ones and checking the results. By observing how the model reacts to the current hyperparameter configuration one can manually tune the hyperparameters and potentially improve the performance. This is ineffective as it is a slow process and one might only find the local minima. An automatic approach is in most cases a better solution as it is fast and can use mathematics and distributions to converge to the global minima.

Hyperparameter optimization is a research field by itself. There exist several papers on the subject [10, 11, 38], but none of those papers mention hyperparameter optimization for transfer learning. Hyperparameter optimization for transfer learning has a unique hyperparameter. This hyperparameter is a layer which decides how many layers of the pre-trained model will be trained. We call this hyperparameter *the delimiting layer*. This hyperparameter is essential to how much of the knowledge from the original task is preserved and how much is tuned to the new task. This is linked to how deep convolutional neural networks are structured [103]. Being structured in a top-down fashion, where bottom layer is input layer and the top layer is output layer, trained convolutional neural networks have a spread of image features. Image features at the top layers will be more abstract than image features at the bottom layers. This means that bottom layer features will be features of natural images, such as lines and colors. Top layers will contain features such as objects from the dataset. When transferring knowledge, we are interested in the bottom layer image features, which are shared among natural images [135]. However, knowing how many layers should be transferred is not trivial. Therefore, we want to automatically detect the delimiting layer, which separates the layers we want to retrain from the layers we want to stay the same.

There are different approaches to optimize the hyperparameter configuration automatically. One such solution is the Bayesian optimization, which, for example, Google has implemented in their cloud engine [45,70]. Bayesian optimization is a sequential design strategy for global optimization of black-box functions [126]. Other solutions are grid search, random search, and Hyperband [77]. In this thesis, we will use a framework for Bayesian optimization for tuning the hyperparameters of image classification models available in Keras being trained with transfer learning on a dataset of medical images of the GI tract.

1.2 Problem Statement

In the medical domain, it is hard to acquire large datasets because there need to be doctors available to annotate the data manually. Additionally, there needs to be consent from the patients. As a result, medical datasets publicly available in gastroenterology are all too limited for training from scratch, except a few which are still much smaller than datasets available in many other fields. Transfer learning is a solution for the problem of training deep convolutional neural networks on small datasets. However, there is a hyperparameter unique to transfer learning which is unintuitive to tune manually. The goal of the thesis is to explore and utilize automatic hyperparameter optimization of this hyperparameter. Additionally, we extend the goal to consider three other hyperparameters which are more related to the training of a CNN than transfer learning itself. These hyperparameters and our solution for optimization will also work on similar tasks outside the medical field. Nevertheless, the dataset we use in this thesis are datasets from the medical field.

The research questions for this thesis are the following:

1. *Can hyperparameters for transfer learning be optimized automatically? If so, how does it influence the performance and what should be optimized?*
2. *How should a system be built that automatically can fulfill the task of automated hyperparameter optimization for transfer learning?*

The ultimate goal of this thesis is to show the benefits of employing hyperparameter optimization on transfer learning, with emphasis on the hyperparameter unique to transfer learning. Hopefully, this can help researchers apply even stronger transfer learning models on issues such as disease detection in the future.

1.3 Scope and Limitations

Based on the research question in section 1.2, the scope of this thesis is on researching experiments running different strategies for automatically optimizing selected hyperparameters for a method of transfer learning. The use case is medical image data of the gastrointestinal tract used for disease detection. Moreover, we need to design a system for automatically running the experiments.

We decided to limit the transfer learning to one method and the datasets used for the experiments to two datasets targeting the gastrointestinal tract. Additionally, we had to limit the number of hyperparameters to optimize since a larger search space has a negative effect on the optimization and we were limited in terms of computing resources. The hardware limitations also affected our choice of hyperparameters as some hyperparameters, such as the batch size, had an impact on the memory used, and as a result, we had to limit the batch size to be small enough so all the tested machine learning models would work with it. The datasets used had some limitations in that they could affect the classification based on irrelevant artifacts such as text in the image and a blue box in the corner of some images.

Another limitation we introduced was to use a single optimization method, which was Bayesian optimization. Bayesian optimization is considered as one of the best methods for hyperparameter optimization for machine learning purposes, but there are several methods building on Bayesian optimization, and one can configure different options for the standard Bayesian optimization as well. Consequently, we limited the Bayesian optimization to the standard method using the default configuration supplied by the software library supplying the Bayesian optimization implementation. Finding the best optimization method and the best configuration of that method could be a thesis itself, so instead, we focused on strategies for using the optimization method on the transfer learning.

1.4 Research Method

The ACM Education Board created in 1989 by a task force on the core of computer science, determines and characterizes the structure of how research in computing, should be approached. This report [28] defines computer science in its essence as an intersection between several central processes. The central processes are applied mathematics, science, and engineering. The paradigms of (i) theory, (ii) abstraction and (iii) design reflect these central processes.

For this thesis, we worked mostly on the engineering aspects of creating experiments exploring hyperparameter optimization for transfer learning. We based these experiments on the pursuit of better results for real-life applications adapting to transfer learning. In our work, we focused on the application of detection of diseases and abnormalities in medical images. As such, the work implements a system to run different hyperparameters with an optimizer wrapped around. Therefore, this thesis touches on the elements of all three processes. The following gives an overview of how the thesis fits into each process.

- **Theory:** The theory process is responsible for defining and characterizing the object under study by formulating and hypothesize possible relationships. Furthermore, it is characterized by determining relationships among objects, verifying their correctness and interpreting the results.

  For the theoretical part, we touched upon deep learning with convolutional neural networks, transfer learning, and optimization methods. We identified the lack of hyperparameter optimization for transfer learning training in previous work and focused primarily on the medical domain where small datasets are convolutional neural networks challenging to generalize.

- **Abstraction:** The abstraction process is used for modeling and emerges from experimental scientific methods. While a researcher is investigating a problem, a hypothesis is formed, a model created, experiments designed and finally data collected and analyzed.

  Most of this thesis falls into this category. We use experiments and different datasets to explore and test our hypotheses. Although the datasets are all from the medical domain, the approach can be used on other system using transfer learning too. We use a well-researched optimization method called Bayesian Optimization. We optimize several hyperparameters and try different ways to do the optimization. Furthermore, we study the performance of the optimization regarding accuracy on a validation set, the number of iterations and speed.

- **Design:** The last process is design, which is closely related to engineering. This involves researchers to state requirements and solutions, followed by designing and implementing a system. This process is concluded with an evaluation of the system.

  The transfer learning training done in this thesis is a solution that could be implemented in computer-aided detection systems. In this thesis, the use case is the medical domain, but our work is general enough to be used in other domains as well. As such, the training system can be implemented as is, or with minor changes to produce transfer learned machine learning models for a given dataset. The Bayesian Optimization wrapper can be added with the same pretense.

We ran extensive experiments on two medical image datasets containing images from the gastrointestinal tract. We showed automatic optimization of four hyperparameters connected to transfer learning using Bayesian optimization. We created a system for automatically running optimization for hyperparameters on given datasets, with detailed plots and numbers being generated as the system finished optimization parts. We analyzed results from two sets of experiments that we ran on two datasets. We calculated metrics suggested by the dataset papers and compared them to methods presented as baselines in the respective dataset papers. We showed that automatic hyperparameter optimization is an effective strategy for increasing detection performance and that automatically adjusting the delimiting layer reveals layers that are nontrivial to select manually. Additionally, we found which optimization strategy is the best and pointed out flaws in the lesser strategies as well as a flaw in our way of optimization. We pointed out this knowledge for future work.

The summarized main contributions of this thesis are:

i. We designed and conducted experiments testing Bayesian optimization for three different optimization strategies with four hyperparameters for a method of transfer learning.

ii. We analyzed and calculated metrics for two sets of experiments, which we ran on two different datasets, which we compared to metrics of methods used as baselines for the respective datasets. The metrics we calculated can be used in the future by researchers for comparisons to heir work.

iii. Technical development of a system for automatically running experimentations of Bayesian optimization in three different hyperparameter optimization strategies for transfer learning of pre-trained CNN models on given datasets.

iv. We introduced a method for stopping training runs which failed to reach a given threshold in a given number of epochs. We called this method *nonconvergence filtering*, and its purpose was to terminate training runs which failed to converge early to save time.

v. We presented a poster and held a 3-minute lightning talk about the thesis at the Autonomy Day 2018, 2nd workshop on Autonomous and Adaptive Systems, at Oslo Metropolitan University, May 3rd, 2018. The website is `http://www.autonomia.no/autonomy-day-2018/`. The poster won the best poster award and is included in appendix A of this thesis.

The summarized main contributions above answers the research questions from section 1.2. Research question one is answered by bullet i and ii. Research question two is answered by bullet iii and iv.

1.5 Outline

The thesis is organized as follows:

Chapter 2: Background: We give more background information about the use case for transfer learning. We talk about the available data in the medical field and the difficulty of annotating this data. We also present related work focused on applying transfer learning on medical image datasets for image classification and detection. This is followed by examples of computer-aided detection systems that implement transfer learning. All of this highlights that hyperparameter optimization is done manually for transfer learning in related work. Finally, we explain transfer learning and how we use Bayesian optimization for hyperparameter optimization.

Chapter 3: Methodology We describe our methodology by presenting and discussing the hyperparameters, the metrics and the datasets we use for our experiments. Moreover, we present the system running the experiments and discuss details in its implementation. Lastly, we introduce a method for saving time during the training by filtering out training runs that fail to converge.

Chapter 4: Results and Discussion We state the design of the experiments and show and discuss results from two experiments trained on each their dataset. We go into detail on each aspect of the results for the largest of the datasets. The lesser dataset results are also presented, but not as in-depth as it showed many of the same behaviors as the larger dataset. At the end of each dataset result presentation, we show the best-trained model and compare it to baseline metrics presented in the specific dataset paper.

Chapter 5: Conclusion: Finally, we summarize and conclude this thesis and present ideas and suggestions for further studies surrounding the work in this thesis and present final remarks about the thesis.

Chapter 2

Background

In this chapter, we will present the background and motivation of our thesis. Will will start with the medical background. Here we will present the motivation and background for our choice of the use case of Gastroenterology. We will show some computer-aided detection systems that use handcrafted image features for detection, and we will present related work into computer-aided detection systems that use machine learning for crafting the image features for detection. Lastly, we will present a computer-aided detection system that uses deep convolutional neural networks for detection of diseases and abnormalities within the gastrointestinal tract.

2.1 The Medical Background

The field of medicine is in ever advancement. Every year new technologies emerge. This development is necessary to meet the health requirements of today's society. The cost to society and the patient cannot be justified, so massive investments are made towards improving medical technologies. These investments have resulted in modern technical equipment being available in several medical branches. This equipment, like cameras and sensors, collect vast amounts of data. After using the data for diagnosis, the data is then mostly archived or deleted. With the rise of machine learning, more and more researchers realize that machine learning algorithms can use this data for analysis and classification. This thesis uses datasets of endoscopies, but primarily colonoscopies for the experiments. As such, the primary focus of the background will be on colonoscopies.

2.1.1 Gastroenterology

In the field of Gastroenterology, there are several challenges. Diseases of the gastrointestinal (GI) tract are in most cases very impactful on the quality of life of the patient. Figure 2.1 shows an illustration of the entire GI tract with the lower anatomy highlighted. They often bring with them symptoms like diarrhea, vomiting, constipation and altered stool [102], in addition to being painful to the patient. Colorectal cancer

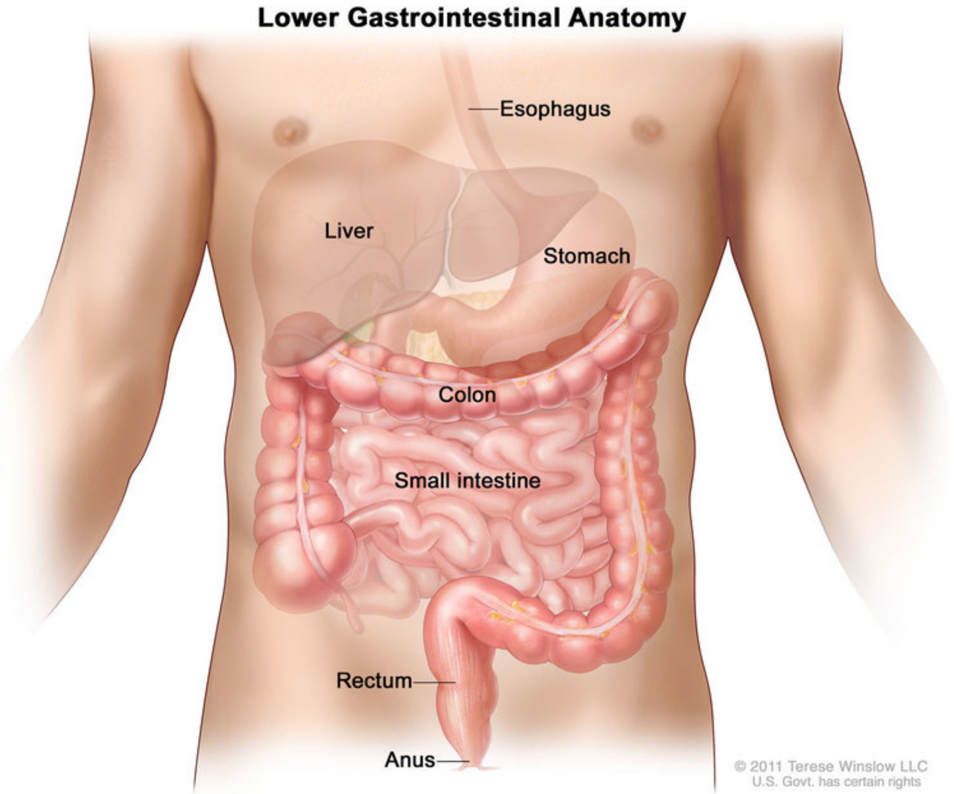

Lower Gastrointestinal Anatomy
Esophagus
Liver
Stomach
Colon
Small intestine
Rectum
Anus
© 2011 Terese Winslow LLC
U.S. Govt. has certain rights

Figure 2.1: Illustration of the lower grastrointestinal anatomy. The whole gastrointestinal tract is illustrated, but only the lower part is highlighted.

(CRC) was the third most common cancer type in 2012 worldwide [16, 19]. It accounted for 8.5% of cancer deaths in the world [39]. The cancer is a major health issue [92], and the 5-year relative survival rate in the later stages of the disease is at a measly 14%. However, if the disease is detected in the early stages, the survival rate increases to 90% [19]. Survival prospects are higher the earlier the cancer is discovered, and studies show that a population-wide screening program improves prognosis and even decreases the incidences of CRC [55]. Because of this, many organizations in the European Union (EU) and United States (US) have guidelines where screening is recommended for the population over 50 years of age and people at increased or high risk of CRC [138]. The aim of the screening is not only to catch the early stages of CRC but more importantly, to prevent CRC by having polyps found and removed [128].

The best way to do screenings today is through colonoscopies as it allows for visualization of the mucosa and the lumen of the entire colon. Figure 2.2 shows an illustration of the entrie colon and figure 2.3 shows an illustration of a colonoscopy procedure. Colonoscopies can be used as either primary screening tool or as a workup tool after other positive screening tests [82]. A major disadvantage of colonoscopy is how the efficiency of the procedure is dependent on the skill of the operator. Even if the operator is a gastroenterologist, substantial operator dependence is consistently observed [7, 15, 21, 53, 63]. On average, 20% of polyps, which are possible cancer predecessors, are missed. This is mainly caused by an unwarranted variation in the endoscopist adenoma detection rate [64, 65], and low rates increase the risk of post-colonoscopy CRC [66]. Furthermore, characterization of detected polyps regarding size, shape and surface structure is essential for the resection strategy and scheduling of further surveillance when needed. High detection rate is not only important for catching cancer early, but also for the public opinion on colonoscopy. The procedure is considered by many to be both painful and dangerous. Painful as the procedure requires air to expand around the colonoscope for the camera to have vision. Dangerous as the procedure can cause perforations which can be fatal if not attended to [42]. Even though we can not help with the public's opinion on how painful or dangerous a colonoscopy is, we can help with the endoscopist's detection rate.

2.1.2 Medical data

The colonoscopy procedure is an endoscopic examination of the large bowel and the distal part of the small bowel [32]. It is conducted with a camera attached to a flexible tube passed through the anus. It makes it possible to provide a visual diagnosis of the lower parts of the GI tract. In addition it allows for biopsies or removal of polyps [106]. Polyps are removed as they are at risk of developing CRC. The polyps have different shapes, sizes and surface structures. Figure 2.4 shows an image of a polyp from one of the datasets we use in this thesis and figure 2.5 shows an illuration of polyps inside the bowel. The shape is described by the Paris classification, and the surface structure most commonly by the NICE

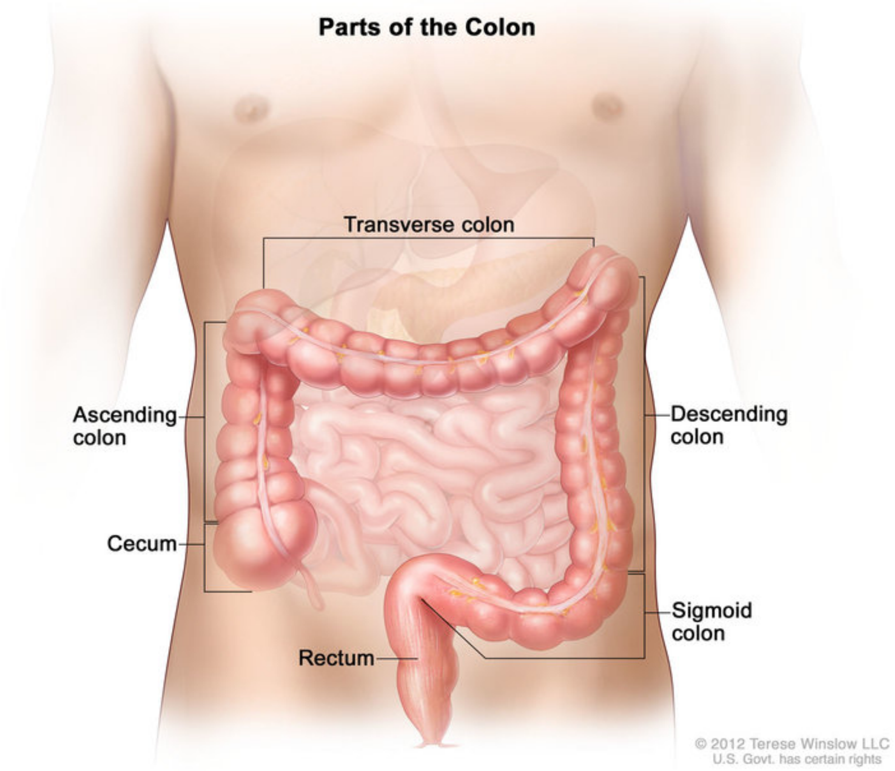


Figure 2.2: Illustration of the colon with pointers to anatomical landmarks.

classification predicting the microscopic histological diagnosis [50]. The majority of polyps are small, non-neoplasitc lesions that are found during colonoscopies. The polyp type decides the potential of the polyp being malignant. If a polyp is found with the potential to turn neoplastic, they are removed through a colonoscopic biopsy. In the procedure of removal, the polyp is dyed and lifed for visibility. Then, the polyp is resected. It is important to remove all of the polyp, or the chances are still there of the polyp turning malignant [12,27].

A colonoscopy generates a lot of data in the form of videos and images. This data is often stored and unused. Riegler [111] quoted a doctor in the Vestre Viken Hospital in Norway, September 2015 as saying; *"I have a lot of data lying around. Like for example images, videos, sensor logs, patient records and so on. Unfortunately, I am not able to use all the different types of data like I would like to do. They are just stored on a computer somewhere. I even don't know where, and I don't think the IT support really know either... Sadly, we are collecting a lot of data, but we do not benefit from it at all."* [110]. The problem is that most use cases for the data requires manual labeling and classification. Without it, the data is useless. This is a waste of resources, so researchers are trying to design viable solutions.

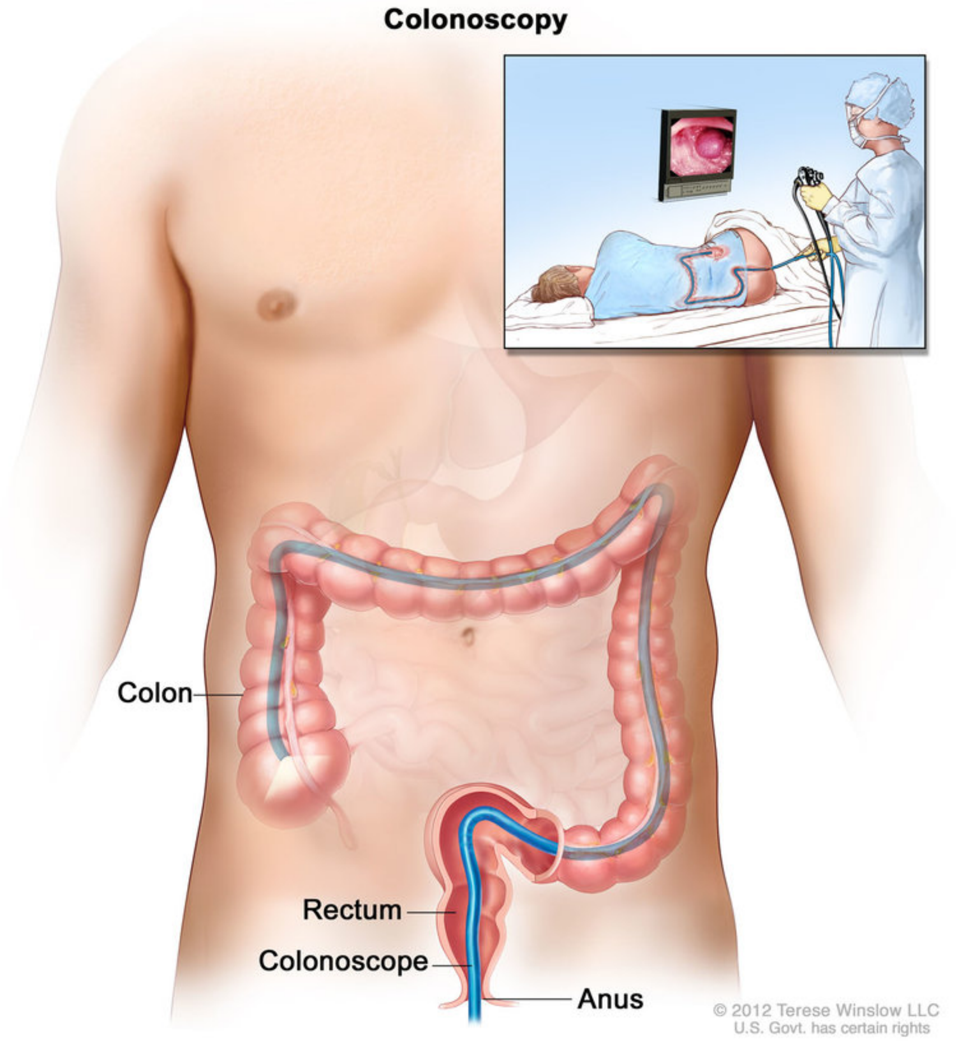


Figure 2.3: Illustration of a colonoscopy procedure. The doctor operates a flexible tune with a camera attached. The tube is then passed through the anus and provides a video of the insides of the colon.

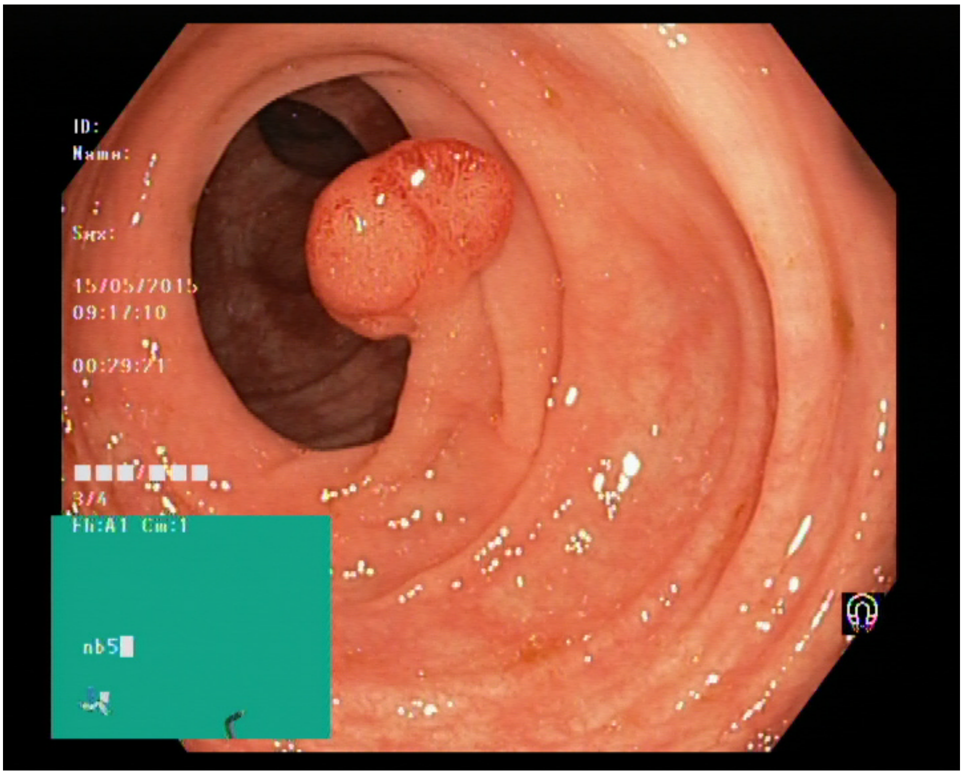


Figure 2.4: Image of a polyp from the Kvasir dataset [98].

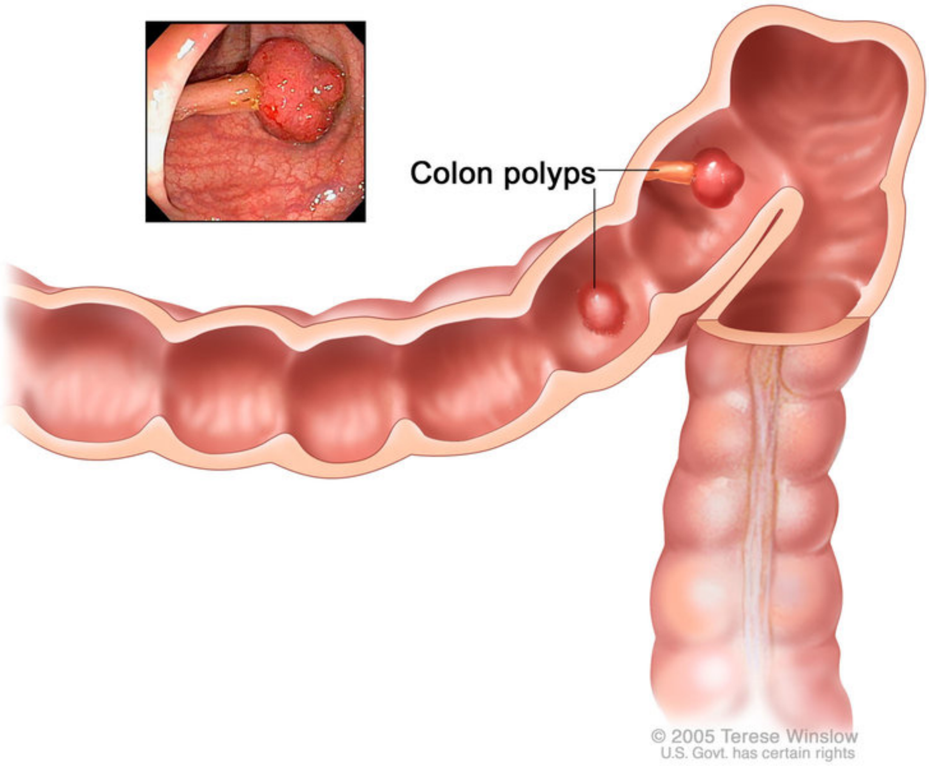


Figure 2.5: Illustration of the colon with polyps growing inside.

2.1.3 Medical Multimedia Systems using Handcrafted Image Features

Several medical multimedia systems have been proposed to remedy the situation and make use of the data. These systems are designed to help annotate, analyze and visualize data. By using image recognition techniques like handcrafted image features and machine learning, they can detect objects in the available data. Some also have systems to help annotate data and support the doctors in visualizing and understanding what caused the system to react. The following systems are such systems, but they do not use machine learning. Nevertheless, they are relevant as they were the precursors to machine learning in the medical domain.

Polyp-Alert

"Polyp-Alert" is a software system created to assist the endoscopist in finding polyps by providing visual feedback during colonoscopy [139]. Polyp-Alert uses edge-cross-section visual features and a rule-based classifier to detect edges along the contour of a polyp. It works by tracking detected polyp edges to group a sequence of images covering the same polyp as one polyp shot. From the experiments done by Wang et al. [139] the software correctly detected 97.7% (42 of 43) of polyp shots on 53 randomly selected video files of entire colonoscopy procedures. Additionally, Polyp-Alert incorrectly marked only 4.3% of a full-length colonoscopy procedure as showing a polyp when they do not.

Polyp-Alert was one of the best options in 2015 for polyp detection. One of the significant drawbacks, however, is the how the system is not able to perform more than ten feedbacks per second. This performance is not real time and would slow down a live colonoscopy. The system is also only able to detect polyps. For such a system to be useful in a live setting, it needs to be at least able to perform in 25+ frames per second to be considered real-time. Nevertheless, Polyp-Alert is not using machine learning for detection. Not using machine learning is limiting as it is more difficult to expand to other diseases and abnormalities. Currently, the system can only detect polyps. However, it has good accuracy, so new systems utilizing machine learning should be able to have the same or higher accuracy.

Basic EIR

EIR, named after the Norwegian mythological goddess with medical skills, is an efficient and scalable information retrieval system for medical videos. The goal of the system is to assist gastroenterologists by automatically detecting diseases in the GI tract from videos or images. It aims to do this in real-time, something which is not usual for these type of systems but is essential for these type of systems to have real-world application. Further, the system is designed in a modular way so it can combine filters using machine learning, image recognition and extraction of global and local image features. So far the system is focused on the detection and classification of polyps but will be extended for other diseases and abnormalities as well. Additionally, a goal of the system is to be able to be used in combination with video from camera pills [113].

EIR includes the whole pipeline from data collection to analysis of the data to visualization and thus consists of these three subsystems:

Annotation subsystem: The annotation system subsystem is the entry point of the whole EIR system. It is to collect efficient training date for the detection and automatic analysis subsystem. Without this part the rest of the system is hard to realize as it relies on the data collected. The system is necessary as labeled data is hard to come by. This is because a physician is needed to label it and there are legal issues involving patient consent. In addition, as discussed earlier in section 2.1.1, the quality of the annotations is dependent on the experience and concentration of the physicians [44].

By utilizing automatic methods that extend the provided data combined with the manual annotations of the physician, the developers of EIR has designed a semi-automatic annotation system. It allows the physicians to only provide annotations in a single frame of the video or image series. The system uses this information to automatically track the regions of interest on previous and subsequent frames. The annotated data collected can, in addition to the EIR system, be used for surgical documentation or teaching purposes.

Detection and automatic analysis subsystem: In EIR, the detection and automatic analysis subsystem is divided into two parts. There is the subsystem that detects irregularities in frames and the localization subsystem that localizes where the detected polyps are in the body. The detection alone can not determine the location, so the localization subsystem exists. The localization subsystem uses the output of the detection as input. The polyp's physical shape is used to find the exact position in the frame. Handcrafted filters are applied to the image to create an energy map. The coordinates of one or more polyps in the frame are chosen from the maxima of the energy map.

Detection subsystem: The detection subsystem is designed to only detect abnormalities like bleeding or polyps in the given video frame. The detection subsystem is designed to only detect if a frame has a polyp. The location of the polyp in the frame is determined by the localization subsystem which uses the detected frames as input. The detection subsystem is built in a modular way allowing it to be extended with new image models or submodels. For example, if there are images of dyed and lifted polyps, this could be used as a submodel of polyps. The detection subsystem in EIR started out as a single-class handcrafted global-feature-based detector able to recognize the abnormalities in a given video frame [100, 111, 113]. Handcrafted image features were used because they were, as Pogorelov et al. put it, "easy and fast to calculate". In addition, the exact positions of the abnormalities were not needed for detection using these. The method was built on a search based method for image classification. Indexes were created from visual features extracted from video frames or images and used by a classifier to then search the index for the frames that are most similar to a given input frame [113]. The detection subsystem then decides, based on the classification results, in which frame the abnormality belongs to.

Localization subsystem: The localization subsystem takes the positive frames list generated by the detection subsystem. The processing of the images is implemented as a sequence of intraframe pre- and mail-filters. These can be extended to localize different diseases. Differently, from the detection subsystem, the localization subsystem uses local image features to be able to find the exact positions of objects inside the frames. Challenges like partially covered objects, ambiance and different medical equipment, which makes the objects hard to find, are handled by applying pre-filters.

Visualization and Computer aided diagnosis subsystem: The visualization and computer-aided diagnosis subsystem have two primary purposes. First, it should help evaluate the system performance of EIR and get better insights into why things work well or not. Second, it can be used as a computer-aided diagnosis system for medical experts. A web-based visualization tool has been developed to support the medical experts in understanding what in the detection caused it to classify the way it did. Additionally, it can help visualize the localization of polyps in a frame.

2.1.4 Related work on Deep Convolutional Neural Networks and Transfer Learning

Much of the rapid development around image classification and analysis using machine learning can be attributed to the development of effective models using deep convolutional networks trained on large datasets [51,57, 124,133]. However, while useful on large datasets, these models lose their generality on smaller datasets. Besides they require expensive hardware to train and training can take several weeks even on good hardware. Transfer learning could be the answer, and the following research papers explore this method for small medical image datasets.

Convolutional Neural Networks for Medical Image Analysis

Tajbakhsh et al. [135] tried to answer in their paper the following central question in the context of medical image analysis: *"Can the use of pre-trained deep CNNs with sufficient fine-tuning eliminate the need for training a deep CNN from scratch?"* This question is central to whether transfer learning is useful for the medical domain. If transfer learning is unable to achieve the same result as training a deep CNN from scratch, other solutions need to be explored.

To answer the question, the researchers carried out an extensive set of experiments for four medical imaging applications: 1) polyp detection in colonoscopy videos, 2) image quality assessment in colonoscopy videos, 3) pulmonary embolism detection in computed tomography (CT) images, and 4) intima-media boundary segmentation in ultrasonographic images. These applications were chosen to represent the most common clinically used imaging modality systems and the most common medical image analysis tasks. The performance of the pre-trained CNNs using transfer learning was compared with the CNNs trained from scratch for each application. The data trained on was entirely medical imaging data. Moreover, the performance of the CNN-based system's corresponding handcrafted counterparts were compared.

The results of the experiments consistently demonstrated that pre-trained CNNs with fine-tuning performed better or, in the worst case, performed as well as CNNs trained from scratch. This was an important find which proves that transfer learning can be a solution for medical image analysis and classification. Besides showing pre-trained CNNs can perform better, fine-tuned CNNs were also more robust to the size of training sets than CNNs trained from scratch. This was not surprising as it is the point of transfer learning to be able to perform well on smaller datasets, but this paper proves this. In context to our thesis, the most relevant result, however, is that the experiments showed that neither shallow tuning nor deep tuning was the optimal choice for a particular application. Hyperparameter optimization could, therefore, find the optimal choice for the depth of fine-tuning. Finally, the experiments confirmed the potential of CNNs for medical imaging applications as both CNN variations outperformed the corresponding handcrafted alternatives [135].

Exploring Deep Learning and Transfer Learning for Colonic Polyp Classification

In work by Ribeiro et al. [108] they explore Deep Learning for the automated classification of colonic polyps. They achieve this by running experiments with different configurations for training CNNs from scratch and distinct architectures of pre-trained CNNs. The experiments are tested on eight high definition endoscopic image databases acquired using different modalities and then compared. Data augmentation with image patches to extend the size of the training database were used to train the CNN from scratch. This work is comparable to the work done by Tajbaksh et al. [135] in that they both explore whether deep CNNs trained with transfer learning work better than CNNs trained from scratch. The difference, nevertheless, is that the focus of this paper is solely on colonic polyps. Additionally, the datasets used are close-up pictures focusing on the properties of the polyp, like color and texture. Since the dataset contained HD images, the texture of the polyp was something that could be detected. As such, the focus of the classification was to classify the polyps into one of three different categories: hyperplastic, adenomatous, and malignant.

In the work, the researchers trained a CNN from scratch and a CNN with transfer learning. The transfer learning procedure was different from ours in that instead of choosing a layer to split the knowledge all the layers were trained from a pre-trained network. Additionally, the focus was also on resizing and cropping the images to increase the results. However, that is not our focus in this thesis. In the paper, they do several experiments with different pre-trained models. The hyperparameters are not mentioned, so there is room to improve with optimization.

The results were compared with some commonly used features for colonic polyp classification. The results were deemed sound and suggested that features learned by CNNs trained from scratch and the pre-trained CNNs features can be highly relevant for automated classification of colonic polyps. Furthermore, the paper shows that the combination of classical features and pre-trained CNNs features can be a viable approach to improve the results further. Finally, the paper concludes that Deep Learning using CNNs is a viable option for colonic polyp classification [108]. The paper builds on the consensus that CNNs with transfer learning is successful at medical image classification, but does not do any automatic hyperparameter optimization to increase the results of the tuned CNN.

Transfer Learning with Deep Networks for Saliency Prediction in Natural Video

From the paper by Chaabouni et al. [20] they propose a deep CNN to predict salient areas in video content. While the use case is not medical videos, but more general, natural videos, it is still highly relevant as polyps are salient in the colon. The deep CNN is trained using transfer learning, which makes it relevant to this thesis and also to the medical multimedia systems that are exploring the use of transfer learning. The CNN in the paper is custom built and is trained on the HOLLYWOOD2 dataset [83], which is a big dataset. This pre-trained CNN is then fine-tuned on two smaller datasets. The results show that transfer learning for the task of saliency in natural videos allows solving the problem of an insufficient dataset. However, even though the results with the transfer learning were better, the gain was not strong [20]. The results showed that transfer learning could improve the performance of a pre-trained CNN on two small publicly available video datasets. For our work, it is relevant because it validates the usefulness of transfer learning on small datasets. However, as with the paper discussed in section 2.1.4, this paper uses a transfer learning method where they do not consider a delimiting layer, but just loads pre-trained weights trained on a larger dataset and trains all layers normally. In our work, we hope to show that by considering a delimiting layer and doing optimization, we achieve better results than baselines provided by the datasets we are using.

2.1.5 Medical Multimedia Systems using Deep Convolutional Neural Networks and Transfer Learning

Researchers of medical multimedia systems paid attention as several different groups of researchers had found good results using deep CNNs with transfer learning for image object detection. The previous work promised a solution for the problem of insufficient amounts of available training data.

Deep EIR

Pogorelov et al., the designers of EIR, took notice of the success other researchers had using deep learning and transfer learning in similar efforts [20, 108, 135], and decided to explore this solution for detection in EIR. They made a neural network version of EIR, called Deep-EIR [94], based on pre-trained convolutional network architectures and transfer learning. The researchers chose transfer learning as the available dataset had insufficient amounts of available training data to conduct adequate training.

The first part of the transfer learning was done the same way we do it in this thesis. A CNN model pre-trained on the ImageNet dataset, in this case, InceptionV3 [134] was used. In our thesis, we remove the classification block instead of just the final layer of the model, which is done in the paper. A block is a bundle of layers which together perform a function for the machine learning model. In the paper, the final layer is replaced with a classification layer fitting the number of classes on the used dataset. Only the final layer is changed during training. However, the similarities between the transfer learning methods end here. In our work, we continue with a fine-tuning step, but Deep EIR decides to use the model trained from re-training the classification layer as the final classifier.

By expanding the transfer learning with a fine-tuning step and performing automatic detection of the hyperparameters, we hope to show that future work into systems like Deep EIR can include these methods and thereby increase the accuracy of their detection systems.

2.1.6 Summary

In summary, the technological advancements in medicine generate substantial data such as video, images, and sensor data. Doctors use this data for diagnosis. It is then archived or deleted. However, medical multimedia systems and computer-aided detection systems try to use this data to assist doctors in the diagnosis. Researchers have proposed many approaches to detect polyps and diseases, and there exist many systems for general image or frame recognition where researchers have spent a considerable amount of time to develop algorithms for detection. Recent medical multimedia systems now use deep convolutional neural networks as they have shown excellent results in image classification. Unfortunately, these networks are heavily dependent on large and varied datasets for training. Without, neural networks become good at classifying the images they are training on, but terrible at classifying other images, even those that are similar. This lack of generality goes against the whole point of deep learning as one wants the trained network to be able to classify similar, never before seen images correctly. However, papers exploring the use of transfer learning have promised a solution for insufficient datasets. Most results in the field of medical multimedia systems are, as a consequence, focuses on pre-trained deep CNN models trained on large general datasets. These are then fine-tuned on the medical dataset. The results have been solid, but there are ways to improve these results through techniques such as hyperparameter optimization and image augmentation. In this thesis, we will try to improve the results through hyperparameter optimization, and then with a particular focus on the frozen layer. None of the previous papers does optimization of this hyperparameter, only manual tuning.

2.2 Machine Learning

Today few people have not heard of machine learning. Machine learning is a field of computer science where the aim is to develop algorithms that can learn a task without being explicitly trained. The advantage is that these algorithms can learn complex patterns, features, and tasks that would otherwise be practically impossible for a programmer to solve with an explicitly written algorithm. Machine learning can be divided into three main categories: Supervised learning, unsupervised learning, and semi-supervised learning. In supervised learning, the model is given a labeled dataset to train on. The goal is to learn a generalized rule that maps similar input to the correct labeled output. In unsupervised learning, the model is given a dataset without labels. The goal is for the model to find a structure in the data or discover hidden features. Semi-supervised learning is a form of supervised learning where not all the target outputs have data to learn from in the training dataset. It can also be in the form of active learning or reinforcement learning. Active learning can be used when there are unlabeled instances in a dataset. A user can then be queried for labeling. Reinforcement learning is when a model is trained through rewards for actions performed in a dynamic environment [117]. In this thesis, we will be using supervised learning.

The popularity of machine learning is sky-rocketing, and massive investments are being made into development and implementations of machine learning solutions. Everyone using the Internet is being exposed to machine learning, and many foresee a future where machine learning is everywhere. From social media, search engines, and self-driving cars to personal artificial intelligence assistants, Internet of Things and most of today's jobs being done by artificial intelligence.

As the popularity of machine learning is something that has exploded over the last decade, many would mistakenly assume that the research field originated around that time ago. This is not the case. Machine learning was coined already in 1959 by A. Samuel [120] and have been researched ever since. The reason for the sudden focus and developments of real-life applications in large-scale happening first this decade is because the algorithms created then were not effective enough. Equally important is that the hardware was very limiting. Even though breakthroughs were made, like the invention of artificial neural networks, the requirements of effective algorithms and powerful hardware met in 2012 when the image database ImageNet [60] held its annual ImageNet Large Scale Visual Recognition Challenge [116]. The winner of the 2012 challenge, the deep convolutional neural network, AlexNet, is considered as the real catalyst for today's AI boom [43, 72]. This was the turning point for large-scale object recognition when large-scale deep neural networks entered the scene [116].

Although the popularity we see today originated from the ImageNet challenge, image classification is not the only use case for machine learning. Different fields utilize machine learning such as Image Processing, Computer Vision, Speech Recognition, Machine Translation, Art, Medical imaging, Medical information processing, Robotics and control, Bio-informatics, Natural Language Processing (NLP), Cybersecurity, and many more [5]. However, for this thesis, we will focus on machine learning for image classification and analysis. One of the most common methods for image classification and analysis is using deep convolutional neural networks [103].

2.2.1 Deep Convolutional Neural Networks

Artificial neural networks (ANN) are machine learning computational models that draw inspiration from the biological neural networks of the brain. ANNs can adapt or learn to generalize or to cluster or organize data. ANN are built of simple processing units reflecting the neurons in the brain and directed, weighted connections between those neurons. The weights are multiplied with the input of the connection and summarized. An activation function decides if the neuron should activate or not. This means that if the result is above the threshold, the output is the result. The result is zero if the result is below the threshold. The network forms a directed, weighted acyclic graph as the output of certain neurons are connecting to the input of other neurons [47].

The typical neural network and the one most relevant for us is the multilayer perceptron (MLP). MLP is a class of feedforward ANN. In these networks, neurons are organized in layers, where neurons from one layer are connected to the neurons of the next layer. For example, three functions $f(1)$, $f(2)$, and $f(3)$ can be connected in a chain to form $f(x) = f(3)(f(2)(f(1)(x)))$. In this case, $f(1)$ is the first layer of the network, $f(2)$ is the second and so on. The layers do not need to be connected in a chain. Many architectures build a main chain and attach to it extra architectural features such as skip connections going from layer i to layer $i+2$ or higher. Data is being fed through the network through these layers until the final layer yielding an output [47]. This process is what is referenced to as "feedforward". However, to be able to learn, the weights of each layer is adjusted by a backpropagation algorithm [47]. The backpropagation algorithm calculates how much each layer needs to change in order to reach the desired output. The algorithm does this by calculating the gradient for each layer, which is then used for adjusting the weights of that layer based on a gradient descent optimizing function. We will use this optimizing function as a hyperparameter in the thesis. How much the weights are changed for each iteration is dependent on the optimizing function, but they all have a learning rate in common. Some functions can adapt this learning rate to the training. We also use the learning rate as a hyperparameter in this thesis. This kind of neural network is called a multilayer perceptron [49,71,125]. The layers between the input and output layer of an MLP are called "hidden" layers. The classical MLP has one or a few more of these. These networks are far from as good as the deep learning networks of today.

Deep learning has several definitions described in work by Deng et al. [33]. Common among them are two key aspects: (1) models consisting of multiple layers or stages of nonlinear information processing; and (2) methods for supervised or unsupervised learning of feature representation at successively higher, more abstract layers. As such, adding many hidden layers to an MLP makes it a deep neural network (DNN). A deep neural network is not fundamentally different from the old standard ANN. This is due to the universal approximation theorem, proven by George Cybenko [31] that states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate any function. A shallow, 1-layer, ANN can in principle learn anything [75], but the layer may be infeasibly large and may fail to learn and generalize correctly. For now, DNNs work better as they can reduce the number of units required to represent the desired function and can reduce the amount of generalization error [47].

The type of DNN we will use in this thesis is the deep convolutional neural network (CNN). CNNs [74] are a special type of neural network where the core building block is a convolutional layer. These neural networks are used for processing grid-like topologies such as 1-D grids or 2-D grid such as pixels in an image. Goodfellow et al. [47] describe neural networks as being *"simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers."* Convolution is a specialized kind of linear operation [47] where the aim is to create a feature filter. One can think of CNNs as a way to imitate the animal visual cortex in the way it is organized [85]. CNNs create image feature convolutional filters that resemble the layers of the visual cortex [93]. As such, CNNs are the quintessential machine learning method for image classification, analysis, and object detection [51,57,72,103,108,124,135].

CNNs are feedforward networks and generally have an architecture of blocks of convolutional and pooling layers. As in standard feedforward networks, either one or more fully connected layers follows the blocks. An image is input directly to the network, which is put through the layers of convolution and pooling. The image features gathered from these operations are then fed into one or more fully connected layers. The final, fully connected layer, outputs the class label. However, several models exist that change this architecture [103]. CNNs require large datasets to avoid the problem of overfitting. When training any CNN model, we want to strike a balance between the model being able to classify as much as possible in a dataset correctly, and being able to classify other, similar images, which are outside of the dataset, correctly as well. Early stopping is a technique for achieving the balance by testing the classification against a different, similar dataset, called the validation set. The early stopping checks if the accuracy or loss from the classification on the validation set is growing over time. The implementation of early stopping usually takes a metric to monitor and the number of epochs the training is allowed before being required to pass the previous best. If the accuracy or loss on the validation set declines, but the accuracy or loss on the training set increases or stay the same, our model is being overfitted. Small datasets are problematic for reaching good classification rates without overfitting as there are not enough diverse samples for the algorithm to be general. However, transfer learning is a learning technique that helps solve this problem.

2.2.2 Transfer learning

Transfer learning is a technique for training machine learning models. The technique aims to transfer the knowledge obtained by one task in one domain to another task in another domain [90, 140]. There are several techniques for transfer learning, such as the works of Razavian et al. [104] and Oquab et al. [89] which are suggesting that intermediate layers of a CNN can be used as input features to train other classifiers, such as support vector machines, for other, different applications obtaining good performance [108]. However, we will focus on one method for transfer learning, suitable for our use case. In this method, we remove the classification block of a pre-trained CNN model. The pre-trained model will be used as a hyperparameter. We replace the removed classification block with our own block, which is fitted to the number of classes on the dataset we are currently using. Then, only the classification block is trained, while the other layers in the model stay the same. The classification block is trained until it can classify images based on the previous knowledge. The next step is to fine-tune the model. We want to train some of the layers, but not all. Some transfer learning methods will train all the layers, such as the method used in the paper described in section 2.1.4, but we, instead, want to explore using a transfer learning method where only train some of the layers. Each block in a CNN model creates image features which are used for the classification. These image features become increasingly abstract as we progress through the network. For example, the earlier layers can show a line, or a color, while later layers can show the outlines of an object. The point of transfer learning for CNNs is to keep the lower-level image features that are learned on very large image sets and tune the higher-level image features on the new dataset.

Our approach is, therefore, to select a layer which splits the model's layers into two groups: The layers that are training and the layers that are left with the weights from the pre-training. All layers which come before the selected layer are left with weights from the pre-training, and all layers after the selected layer are tuned on the dataset we target. We call the splitting layer, the delimiting layer, and use it as a hyperparameter in this thesis.

The advantage of and the primary reason for using transfer learning is related to generalization and the size of the datasets used for training. The training technique is useful for the problem of having small datasets [62]. When training deep learning models, the difficulty lies in achieving good results, while at the same time being able to generalize over different, but similar input. In this context, it means that a neural network should be able to reach good results on similar images even though it has never seen those images before. For example, a CNN using a dataset of cars to train should after training be able to detect other, similar images of cars and not only the cars in the training dataset. If the CNN can detect cars from its training dataset at a much greater rate than similar images it has never seen before, we say the CNN has overfitted on the data. Likewise, the training aims to go from a state of undergeneralization to a state of generalization. The issue of a small dataset for training is that there are too few samples for a DNN or CNN to be able to reach a good rate of generalization, and the network tends to overfit [20, 108, 140].

Another advantage of using transfer learning is that training a CNN from scratch is very costly. Not only does it require hardware to keep it in memory, but even with powerful hardware, the training can take several days [41]. Using transfer learning on the other hand only takes hours. One would think that transfer learning would come at the disadvantage that transferring knowledge from a different, but similar dataset would produce lesser accuracy than those CNNs that were trained from scratch. However, research on transfer learning for CNNs have shown that transfer-learned CNNs can produce the same or even better results than those trained from scratch [20, 94, 108, 135]. This applies only to those domains where very large datasets are unavailable. Therefore, in this thesis, we focus on transfer learning for pre-trained CNNs.

2.2.3 Hyperparameter optimization

In any machine learning models, there are certain parameters that can be changed. These parameters greatly affect the outcome of the training. In the context of a CNN, the parameters can be such as which optimizing function, what learning rate, and what batch size to use when training. These parameters are very different from the parameters, or weights, the model is adjusting during training, and as such is called ”hyperparameters”. There are many hyperparameters to choose from, and as long as these are parameters that don’t change the structure of the overlying training, we will refer to them as hyperparameters. This means that we consider the choice of pre-trained CNN model as a hyperparameter in this thesis. This is because changing between different pre-trained CNN models does not change the transfer learning technique used, nor the dataset or the task, only the result.

Considering that the number of hyperparameters can be large, tuning them is not a trivial task. They have a huge impact on the success of the training. For example, the learning rate helps determine how much the weights are adjusted each iteration. Having a too low learning rate will cause the training to take too long to reach good results. Having a too high learning rate will cause the training never to reach the good results as the weights are adjusted past the optimal value each iteration. There exist optimal hyperparameters for each task, so to improve the results of a model, finding these optimal hyperparameters is important. Manually tuning the hyperparameters is possible, and has in fact been the procedure for many machine learning tasks, but it is very ineffective and usually relies on good starting parameters from other previous research literature. Automatically detecting the optimal hyperparameter configuration seems to be the best way.

In transfer learning for CNN models, there is one special hyperparameter. Choosing which layers to tune on the dataset and which layers to keep the pre-trained weights is a hyperparameter and essential for the effectiveness of the transfer learning. There is little to no research on the topic. Most research into fine-tuning pre-trained CNN models uses a manually chosen delimiter. They split the layers block-wise. For example, many papers choose only to fine-tune the last convolutional block of a CNN. The reasoning being that lower-level image features wants to be kept general, and only the top-level features, where the high-level abstraction happens, needs to be tuned. This sounds reasonable, but there is no way to be sure this is true in practice without trying delimiting on other layers. Automatic hyperparameter optimization might give better and unexpected results.

Optimization of hyperparameters relates to the problem of optimizing "black-box" functions. A black-box function is a function where all we can do to affect the outcome is to alter the input. We consider a neural network a black-box as we do not have an expression of the neural net we can analyze, and we do not know its derivatives. We also consider the output of the neural network noisy, as small changes to the hyperparameters make the results swing unpredictably. There are several methods for hyperparameter optimization, and new ones are emerging with time. Typical examples of optimization algorithms are grid search, random search, and Bayesian optimization. Grid-search is a method where an exhaustive search is conducted on a manually specified subset of hyperparameter space. This method has been proven to be less effective than random search, where the method is to select random hyperparameters from a specified search space [11]. Regardless, there are better methods. Bayesian optimization and related sequential model-based optimization techniques have proven effective [10,26,61,76,127,131], and hyperparameter optimization systems like Google Vizier implements it [46]. There exist other optimization methods, but Bayesian optimization is commonly used because of its good results. As such we will use Bayesian optimization in this thesis. Nevertheless, there are Bayesian optimization methods that have shown better results [84] than the one we will use as well as other methods such as Hyperband [77]. These could be explored in further work.

Bayesian optimization is a strategy for global optimization of noisy, expensive black-box functions [127]. The efficiency of Bayesian optimization stems from the ability to incorporate prior belief about the problem to help direct the sampling and to trade off exploration and exploitation of the search space [14]. It is called Bayesian because the optimization strategy uses the Bayes' theorem. The Bayes' theorem describes the probability of an event based on prior knowledge of conditions that might be related to the event [9]. This is used to construct a probabilistic model that defines a distribution over objective functions from the input space to the objective.

In this thesis, we use Gaussian Process [37] as the probabilistic model. The model becomes a surrogate to the real machine learning model but is significantly cheaper to optimize. Gaussian Process is a model that generates data located throughout some domain. The model is fitted through observations in that domain to the model it surrogates. The Gaussian Process aims to output the same as the surrogated model for the same inputs. For Bayesian optimization, we use Gaussian Process as the surrogate model and the hyperparameters as the domain. Gaussian Process can work in more than one domain, which means we can use more than one hyperparameter at a time. The Gaussian Process needs to make observations in the search space to map out the surrogated model. The Bayesian optimization selects observations from the domain sequentially. An acquisition function called Expected Improvement [101] decide from where the next observations are taken in the search space. The acquisition function tries to balance exploring and exploiting. Exploring means to get observations from areas in the search space where there are few observations. Exploiting means to get observations from areas in the search space where good results have been found. The point of exploiting is that we rarely hit the best point immediately, and adjusting the point of observation slightly might return better results. In this thesis, we use Bayesian optimization with Gaussian Process as the surrogate model and Expected Improvement as acquisition function. The domain for the Gaussian Process is the bounds of the hyperparameter values we set. The result which is evaluated is the validation accuracy from an iteration with hyperparameter set selected by the acquisition function.

2.3 Summary

In summary, machine learning exceeds at several tasks where developing explicit algorithms are unfeasible. For image detection, classification, and analysis, deep convolutional neural networks have achieved superhuman results. These networks are trained on large datasets with many features using supervised learning. The training can take many weeks and relies on hyperparameters that yield a good result. An apparent drawback to using CNNs is its reliance on large datasets. When training on smaller datasets, the problem of overfitting makes it very difficult. Transfer learning alleviates this problem. It is a training method where the aim is to transfer knowledge from one pre-trained machine learning model onto another. This helps both generalization and training time. In the case of CNNs, the classification block of the pre-trained CNN is adapted for the new dataset and then trained until the classifier can classify based on the existing features of the pre-trained CNN. The next step is to fine-tune the pre-trained model by choosing how many layers to tune to the new dataset. Layers in a CNN contain different levels of feature abstraction. Early layers contain low-level features like lines and curves, while later layers contain high-level features like objects. When fine-tuning a layer is chosen as a delimiter, and all later layers are trained on the new dataset while earlier layers keep their pre-trained weight values. Therefore, choosing the layer is a hyperparameter. Hyperparameter optimization can be done in several ways. For CNNs, there are many methods created for hyperparameter optimization, and the results have been good. However, for transfer learning CNNs, there is little hyperparameter optimization done on the delimiting layer. Automatic hyperparameter optimization could achieve better results on already existing fine-tuned models. Several methods for automatic hyperparameter optimization exists, but the most common one is Bayesian optimization. Bayesian optimization is well researched. We have therefore chosen to use standard Bayesian optimization in our experiments.

Many researchers have found transfer learning to be effective for training CNNs on small datasets insufficient for training from scratch. Hyperparameter optimization is used for the pre-training, but not for the hyperparameter of the fine-tuning in transfer learning, which is the delimiting layer. This layer has a substantial impact on the efficiency of the fine-tuning. Therefore, we would like the use automatic hyperparameter optimization on this hyperparameter to test this. There are several ways to conduct automatic hyperparameter optimization, but we have landed on the use of Bayesian optimization as it delivers excellent results and its use is widespread by the research community.

Chapter 3

Methodology

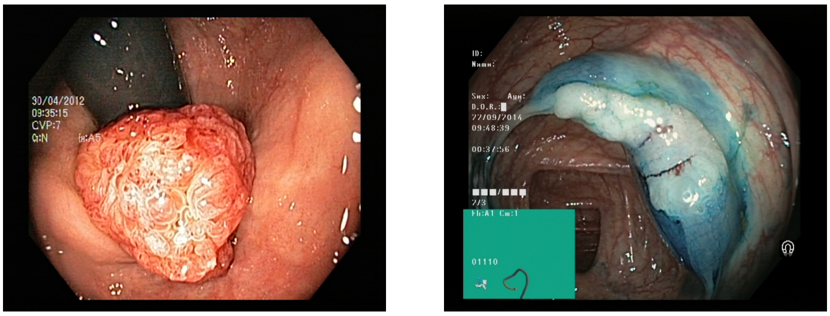
To achieve our goal of testing automatic hyperparameter optimization for transfer learning, we have to create a system for doing so. From related work, we found that there is room to improve models trained through transfer learning by adjusting the number of layers we train during fine-tuning. Additionally, we would like to test other hyperparameters. In this chapter, we will detail how we designed and implemented a system for automatically optimizing a given set of hyperparameters. Additionally, we will discuss our design decisions and introduce our choice of libraries, metrics, and hyperparameters.

3.1 Datasets

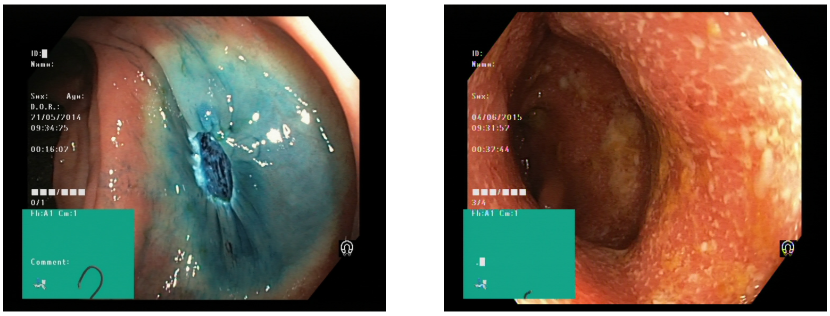
In this thesis, we chose to focus on the use case of medical images. The reason for this is that our research group at Simula Research Laboratories, is currently researching a medical multimedia system for automatic detection of diseases in the GI tract [95, 97–100, 110, 111, 121]. There is much value to society in improving algorithms helping doctors diagnosing diseases from medical images, and as such this is the ultimate goal of this thesis and the medical multimedia system being researched. From the previous research and work conducted by members of our group, there currently exists two publicly available medical datasets which we will use for our use case. As discussed in the background chapter, the availability of annotated medical images makes these datasets insufficient for deep learning from scratch, but is perfect for our use case of transfer learning.

3.1.1 Kvasir

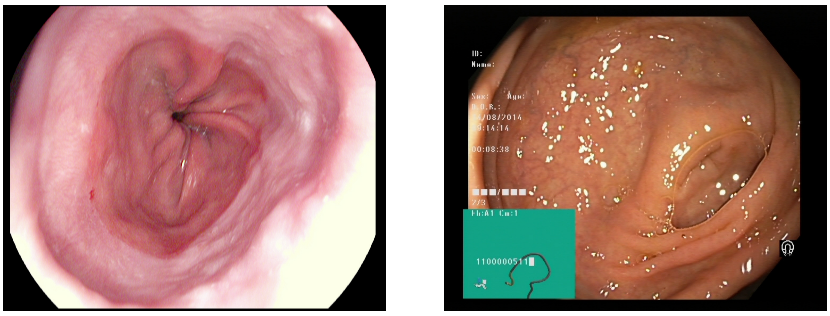
Kvasir is an eight-class dataset of images from the lower human GI tract. It is a dataset made available to researchers, educators, and students. The collection of images are classified into three important anatomical landmarks and three clinically significant findings. Furthermore, it contains two categories of images related to endoscopic polyp removal. The goal of the dataset is to help researchers develop systems that improve the health-care system in the context of disease detection in videos of the GI



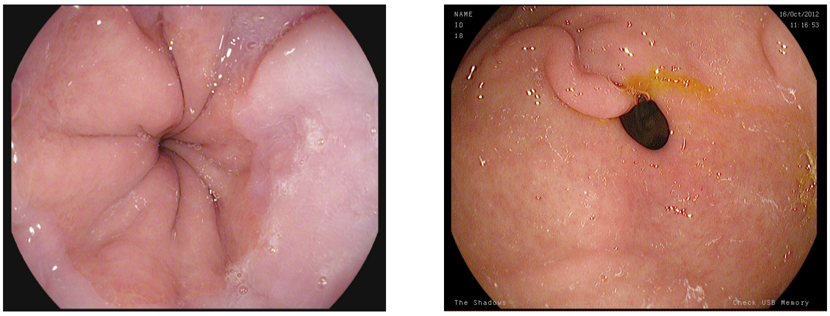
(a) Polyps
(b) Dyed and Lifted Polyps



(c) Dyed Resection Margins
(d) Ulcerative Colitis



(e) Esophagitis
(f) Normal cecum



(g) Normal z-line
(h) Normal pylorus

Figure 3.1: Images from each class in the Kvasir dataset [98]. The classes are from the categories anatomical landmarks, pathological findings, and endoscopic procedures. The images are taken from different endoscopies and are images of diseases and abnormalities are taken from different locations in the GI tract.

tract [98]. The dataset is a response to the lack of publicly available datasets of the GI tract. Additionally, many publications show results that are hard to reproduce because they use different and non-public data. Therefore, Pogorelov et al. in collaboration with experiences endoscopists from Vestre Viken Hospital Trust (VV) in Norway, responsible for the annotation of the dataset, went together and released Kvasir: A Multi-Class Image Dataset for Computer Aided Gastrointestinal Disease Detection [98].

Endoscopic equipment at VV captures the medical images which one or more medical experts from VV and the Cancer Registry of Norway (CRN) carefully annotates. VV consists of 4 hospitals and provides health care to 480,000 people in 26 municipalities in Norway [137]. One of the hospitals has a large gastroenterology department from where Kvasir has collected the dataset. This department will continue collecting data, so Kvasir will continue to grow over time. CRN conducts cancer research and is responsible for the national cancer screening programmes in Norway with the goal to prevent cancer death by discovering cancer or signs of cancer early.

There are currently two versions of Kvasir. Version one is the first version of Kvasir and consists of 4,000 images in 8 classes showing anatomical landmarks, pathological findings or endoscopic procedures in the GI tract. There are 500 images for each class in version one. Anatomical landmarks are features within the GI tract that are easily recognizable through an endoscope. They are used for navigating the GI tract and can be used to describe the location of a given finding. The anatomical landmarks are Z-line, pylorus, and cecum. Pathological findings are, in the context of Kvasir, abnormal features of the GI tract. It is visible as damage on or changes in the normal mucosa. The findings may show an ongoing disease or a precursor to for example cancer. The pathological findings available in Kvasir are esophagitis, polyps, and ulcerative colitis. Endoscopic procedures refer in this case to procedures related to the removal of polyps. The endoscopic procedures depicted in Kvasir are dyed and lifted polyps and dyed resection margins. The procedure of dyeing and lifting polyps is done to minimize the risk of mechanical or electrocautery damage to the deeper layers of the GI wall. Additionally, the color helps facilitate accurate identification of the polyp margins. After the polyp is dyed and lifted, it is removed by use of a snare. Evaluation of the resection margins is important to make sure the polyp is removed entirely. The images are taken from different colonoscopic procedures.

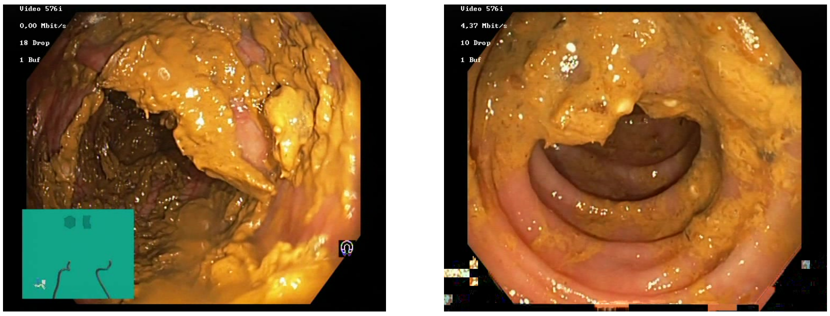
In our thesis, we use version two of Kvasir. Version two extends the first version and consists of 8,000 images. It has the same eight classes as version one, and as such there are 1,000 images for each class. Figure 3.1 shows an example image from each class in the Kvasir dataset. Kvasir is the most extensive dataset we have available and is sufficient for transfer learning. However, the images contain several artifacts, such as text and the blue box in the left bottom corner as can be seen in several images in figure 3.1 and in figure 2.4. The problem with these artifacts is that they can train the CNN to activate on them, which means they lose their generality. For example, the model can learn that polyps must have a blue box and text in the image. Any image we server to the model after training that does not contain such a blue box, but indeed a polyp might be deemed not a polyp by the model. Nevertheless, this is the best we have available at the moment, and it will be sufficient for our experiments as our approach can be transferred to better datasets in the future.

3.1.2 Nerthus

Contrary to Kvasir, where there are different images from different colonoscopic procedures, Nerthus' images are from videos of different parts of the lower GI tract. Nerthus is a four-class image dataset labeled with different bowel preparation quality scores according to the Boston Bowel Preparation Scale (BBPS) [73, 97]. As with Kvasir, Nerthus was created by Pogorelov et al. in collaboration with VV and CRN [97]. However, the dataset is focused differently. Pogorelov et al. identified the challenge for doctors of assessing the bowel preparation quality before a colonoscopy. It is crucial for a successful colonoscopy to have clean bowels for detecting diseases since uncleansed bowels can influence decisions on screening and follow-up examination intervals. There exists reliable and validated bowel preparation scales, but grading still varies between doctors. These inequalities could be reduced by an objective and automated assessment of bowel cleansing. By allowing researchers and academics to use Nerthus for their research, Pogorelov et al. want researchers to contribute in the medical field by making systems that automatically evaluates the quality of bowel cleansing for colonoscopies.

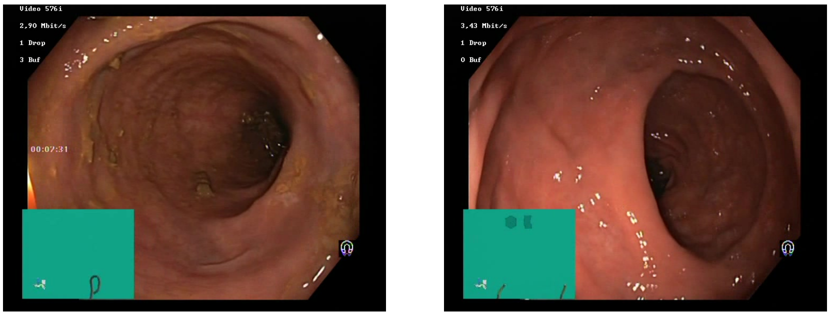
State-of-the-art bowel preparation scales include the BBPS [73] and the Ottawa Bowel Preparation Scale [114]. Both are reliable and validated, but for Nerthus, the dataset is labeled after the BBPS score. The reason for this is that the creators of Nerthus found it to be the best validated and most frequently used scoring system in both routine clinic and screening settings today. BBPS uses a four-point scoring system ranging from 0 to 3, where 0 is the worse and 3 is the best quality of the bowel preparation. Examples from the system can be seen in figure 3.2. BBPS divides the bowel into three sections; right, middle, and left. The four-point score is then given to each section and summarized. However, in Nerthus, all the videos are recorded in the left part of the bowel, and as such, all the scores are between 0 and 3.

Nerthus has only one version so far. It consists of 21 videos with a total number of 5,525 frames. The frames are annotated and verified by experienced endoscopists from VV and CRN. Furthermore, there are plans to involve medical experts from several different countries through a web-based test to get a dataset with higher quality regarding bowel preparation assessment. However, this will become an extension of Nerthus at a later date. For now, Nerthus contains four classes showing four-score BBPS-defined bowel-preparation quality videos. There are between 1 and 10 videos per class, with the number of frames per class varying from 500 to 2,700. Nerthus is smaller than Kvasir, both in regards to the number of labels and in regards to the number of images. However, it is still sufficient for transfer learning. Nevertheless, it suffers from the same problem of



(a) BBPS 0 (from splenic flexure)

(b) BBPS 1 (from descending colon)



(c) BBPS 2 (from sigmoid colon)

(d) BBPS 3 (from rectum)

Figure 3.2: Images from each class of the Nerthus dataset [97]. The images are frames from videos from colonoscopies. BBPS is a scale for bowel cleanliness. A score of 0 is most unclean, while 3 is clean. The videos are taken from different parts of the GI Tract, which is problematic for model generality.

blue boxes and text artifacts as Kvasir. Additionally, each class contains videos from a different part of the GI tract. The problem with this is that images from different parts of the GI tract look different without focusing on the cleanliness, which could mean that the trained CNN model is not classifying based on cleanliness level alone, but also with regards to the location of the image. Even though Nerthus can be considered somewhat flawed because of this, it is still interesting for our use case of transfer learning even though we might end up with overinflated results.

3.2 Metrics

When optimizing, the Bayesian optimization algorithm needs a metric to evaluate and minimize or maximize. For transfer learning, this metric can be one of two: (i) Validation accuracy and (ii) validation loss.

Validation accuracy is a value from 0 to 1, where 1 is a perfect classification, and 0 is all wrong in the classification of the validation set images. Validation accuracy is the calculated percentage of correct classifications, and as such, we want to maximize this value when optimizing. Validation loss is different from validation accuracy in that we want to minimize this value as it is a summation of the errors made for each example in the validation set. In this thesis, we have chosen to focus on validation accuracy for both optimization and the model's performance evaluation. The reason for this is that validation accuracy is what we care about most. We want to reach the highest classification rate possible without losing data generalization. However, using validation loss for optimization would do the job as well. Therefore, it is not clear which one to choose when optimizing. In the first iterations of the experimentation, we used validation loss. It worked fine, but we noticed in some runs that the validation loss was different between two test iterations with the same validation accuracy. We, therefore, changed to validation accuracy to test that out as well. We did not notice any difference between the two and decided to keep using validation accuracy because of it.

Calculation of accuracy and loss can be done the same way for the training set. However, we only care about the validation set. The reason for this is that the accuracy calculated for the training set does not account for "overfitting". Overfitting is when a neural network is overgeneralizing when training. It means that the neural network fits too much on the dataset it uses for training, and the result is that, for example, a CNN model will be good at classifying data from the training dataset, but will be worse at classifying data from any other similar datasets. The CNN model learns the training examples and becomes ineffective for the validation set.

The goal of any CNN training is to train a model that is good at classification, but at the same time good at generalizing. For this reason, we split image datasets into a training set and a validation set. We use the training set for training and the validation set for testing the generalization. Having another dataset is an alternative. The background for this is that we also use the validation dataset for checking when the model is good enough to stop training. Using the validation dataset for this could in some cases mean that the model is overfitted on the validation dataset instead of the training dataset. However, it is unlikely since the weights change from input from the training dataset. Nevertheless, to be sure, some choose to add another dataset which has nothing to do with the training or determining when to stop to be sure the dataset is generalized. In this thesis, we omit using another dataset as we have little data and we assess that we should not have problems with overfitting to the validation dataset as we find it highly unlikely. However, future work could look into this aspect.

Besides the previously mentioned metrics, there are also other metrics mentioned in related work. Although metrics mentioned in related work do not affect the optimization, they are still important to get a full picture of the model's performance. Additionally, it gives a more precise view of the results from the optimized models. For both the Kvasir and Nerthus datasets, the authors suggest a set of metrics they deem to be important in the field of image detection. We include these metrics to understand the performance of our automatically hyperparameter optimized CNN models better. It also gives easier comparisons with related work that use these metrics.

We will use the following metrics, suggested by Pogorelov et al. in the Kvasir paper [98]:

- **True positive (TP).** The number of images containing an endoscopic finding that were classified correctly as an endoscopic finding.

- **True negative (TN).** The number of images without endoscopic findings that were classified correctly as without endoscopic findings.

- **False positive (FP).** The number of images without an endoscopic finding which was wrongly classified as an endoscopic finding.

- **False negative (FN).** The number of images with an endoscopic finding which was wrongly classified as not an endoscopic finding.

- **Recall (REC).** The ratio of samples that are correctly identified as positive among all existing positive samples. The number of correct classifications divided by the number of classifications that should have been correct. The function is $REC = TP/P = TP/(TP + FN)$, where P is the sum of all positives.

- **Precision (PREC).** The ratio of samples that are correctly identified as positive among the returned samples. The number of correct positive classifications divided by the sum of all positives. Precision is calculated by the following: $PREC = TP/(TP + FP)$

- **Specificity (SPEC).** The ratio of samples that are correctly identified as negative among the returned samples. The number of correct negative classifications divided by the sum of all negatives. The function is $SPEC = TN/N = TN/(TN + FP)$, where N is the sum of all negatives.

- **Accuracy (ACC).** The percentage of correctly classified images. It is calculated by the following: $ACC = (TP + TN)/(TP + FP + FN + TN)$
- **Matthews correlation coefficient (MCC).** MCC takes into account true and false positives and negatives and is a balanced measure even if the classes are of very different sizes. MCC is given by the following function: $MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$
- **F1 score (F1).** A measure of a test's accuracy by calculating the harmonic mean of the precision and recall. The function is the following: $F1 = 2TP/(2TP + FP + FN)$

3.3 Hyperparameters

The choice of hyperparameters is essential for the scope of the thesis and the complexity of the optimization. In deep learning, there are several hyperparameters, and we had to limit the ones we focus on to be able to achieve results. The problem with optimizing too many hyperparameters is that the search space for the optimization algorithm increases in dimension for each hyperparameter added. The added dimensions mean we get exponential growth in search space making it very difficult and unlikely that the optimization algorithm can find the global minima, or even approach it. As a result, we have chosen to limit the hyperparameter search space to four hyperparameters. Additionally, hyperparameters affecting the amount of resources the model requires, such as batch size, was not included as our resources was limited, and we were restricted to a batch size of 16 to have enough memory for all of the pre-trained models.

3.3.1 Pre-trained Model

There are several pre-trained CNNs available in the Keras library. These are well-known state-of-the-art CNN models for image detection, classification, and analysis. For many years, ImageNet [60] has been a benchmark and dataset for CNN models. Researchers try their best to tune their new models to this dataset, and much of the competition between the models is about the attained accuracy on the ImageNet validation dataset. As such,

Table 3.1: Table of optimized hyperparameters and their type and bounds

Hyperparameter	**Type**	**Domain**
Pre-trained Model	Discrete	Pick from a list of models
Model Optimizer	Discrete	Pick from a list of optimizers
Learning Rate	Continuous	From 1 to 10^{-4}
Delimiting layer	Discrete	From the first layer to the last layer of the model

Table 3.2: Table of pre-trained CNN models used as hyperparameters in this thesis and their overlying performance and properties. Size refers to how much memory in RAM the model consumes, Top-1 is the top-1 accuracy the model achieved on ImageNet. Top-5 is the top-5 accuracy on ImageNet. Parameters is the number of weights in total in the model, and the depth is the number of blocks in the model. The table is taken from the official Keras website [25].

Model	Size	Top-1	Top-5	Parameters	Depth	Layers
Xception [23]	88 MB	0.790	0.945	22,910,480	126	134
VGG16 [123]	528 MB	0.715	0.901	138,357,544	23	21
VGG19 [123]	549 MB	0.727	0.910	143,667,240	26	24
ResNet50 [52]	99 MB	0.759	0.929	25,636,712	168	177
InceptionV3 [134]	92 MB	0.788	0.944	23,851,784	159	313
InceptionResNetV2 [132]	215 MB	0.804	0.953	55,873,736	572	782
DenseNet121 [58]	33 MB	0.745	0.918	8,062,504	121	428
DenseNet169 [58]	57 MB	0.759	0.928	14,307,880	169	596
DenseNet201 [58]	80 MB	0.770	0.933	20,242,984	201	708

Keras has implemented a select number of the highest yielding accuracy CNN models into their application module. These are available for anyone to use and come with the best-achieved weights pre-trained on ImageNet. The models are therefore easy to use for transfer-learning. As the primary focus of the thesis is to do hyperparameter optimization for transfer-learning, we found it interesting to also find the best pre-trained model for the best results as this would presumably have a significant impact on optimization performance and, therefore, the outcome of the optimization. For outsiders, the models seem like black-boxes where we only see their performance. However, even though the models are using the same principle of convolutional neural networks, differences could impact how fast the optimization performed, the efficiency of the hyperparameter tuning, and whether the optimization can find local minima, let alone the global minima.

3.3.2 Model Optimization Function

It is important to note that we here talk about gradient descent optimization algorithms and not Bayesian optimization. There are a select number of optimization functions available in Keras. We use all of the optimization functions available in Keras, and we list them in table 3.8. The optimizer is one of the two arguments required for compiling a Keras model. All of the optimization functions are based on gradient descent. Gradient descent is the most popular algorithm for optimization of neural networks [115]. There are various algorithms to optimize gradient descent available

Table 3.3: Table of the optimizers used as hyperparameters in this thesis and their default learning rate.

Optimizer	**Default LR**
SGD [13]	0.01
RMSprop [54]	0.001
Adagrad [36]	0.01
Adadelta [143]	1.0
Adam [68, 105]	0.001
Adamax [68]	0.002
Nadam [35, 130]	0.002

in Keras, and we use all of them as hyperparameter bounds for the Bayesian optimization function. The gradient descent optimizers can be configured with parameters such as learning rate, decay, and parameters unique to each optimization such as momentum for the stochastic gradient descent optimizer (SGD). However, if these parameters are not specified, the optimization uses default values. In our work, the only parameter we change is the learning rate.

We chose to include the optimization function as a hyperparameter since selecting an optimizer for the training is not trivial. We regard the optimization functions as black-box optimizers. We have this perception because we are mostly unable to predict the optimizers effect on training. Additionally, finding strengths and weaknesses between the optimizers requires deep knowledge and experience in using them. To someone unfamiliar with the different optimizers, it is difficult to select between them. Aside from the difficulty of comparing them, there is also the nontrivial task of combining them with a pre-trained model. Without trying out every combination of optimizer and model, knowing which optimizer works best with which model seems to us impossible. By using Bayesian optimization for automatic hyperparameter optimization of the gradient descent optimization algorithm, we eliminate the need for treating the optimizer as anything else but a black-box, which significantly reduces the complexity of transfer learning.

3.3.3 Learning Rate

We decided to include the learning rate of the gradient descent optimizer as a hyperparameter. The reason for including this hyperparameter is that it is common for all gradient descent optimizers. Additionally, the effect of the learning rate is hard to predict before training and behaves differently between the different training runs. By different training runs, we mean between the classification block run and fine-tuning run. Intuitively, the classification block run should require a higher learning rate than the fine-tuning run as the point is to reach a sufficient classification ratio fast. For the fine-tuning, we are interested in reaching the best result and considering we start the training with already attained knowledge, we, therefore, use a smaller learning rate. A large learning rate would in a way contradict the idea of fine-tuning as the weights being trained would take large steps in the gradient descent direction, which would make it harder to reach the optimal results.

However, there are also arguments to be made about excluding this hyperparameter. Not only does it increase the dimensionality, it is also hard for the optimizer to optimize. The learning rate hyperparameter bound is continuous and the combination of learning rate and gradient descent optimizer is very different for each optimizer. In table 3.8 each of the optimizers we use is listed together with their default learning rate. As they are different for each optimizer, it means that the optimal hyperparameter changes everytime the optimizer changes. This volatility will make it harder for the Bayesian optimization to find the optimal hyperparameters as there will be a high density of local extrema.

Moreover, we have to consider if changing the learning rate is the right thing to do considering default parameters exists. Firstly, the creators of the optimizers have given default values based on their research. By asserting these to be the correct values for the learning rates would make changing them pointless. We believe the only way to find out if the provided learning rates are the optimal ones is to include the learning rate as a hyperparameter. Secondly, some of the gradient descent optimizers are implemented to automatically adjust the learning rate while training. Gradient descent optimizers such as Adagrad, Adadelta, and RMSprop adapts the learning rate to the parameters, performing larger updates for infrequent parameters and smaller updates for frequent parameters [115]. Changing the learning rate from the default value might work worse for these optimizers. In the documentation of the optimizer available in Keras, the user is advised to not change the learning rate. Nevertheless, we argue that if the default value given by the researchers is the best one, then our Bayesian optimization algorithm will reveal this, so we include learning rate as a hyperparameter.

3.3.4 Delimiting Layer

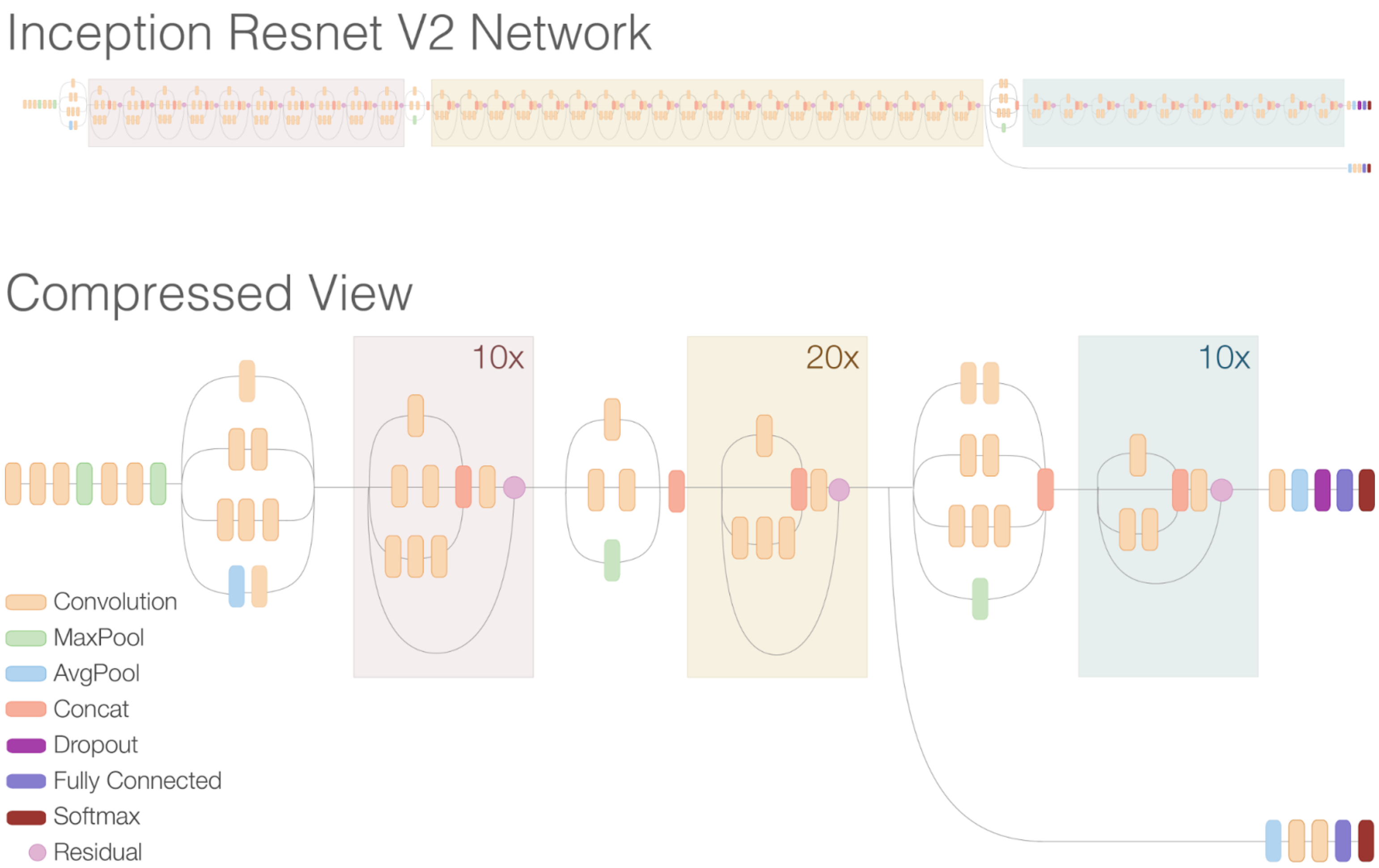


Figure 3.3: Schematic diagram of Inception-ResNet-v2 taken from the Google AI blog [4]. Each colored rounded rectangle represents a layer. The layers are structured from left to right, with left being the input and right being the output of the model. The lines represents the paths the data can take.

For our way of doing transfer learning, fine-tuning a pre-trained CNN model requires one to know which layers to train and which layers to keep their pre-trained weights before fine-tuning. CNN models are complex. From figure 3.2, we can see that each model has different amounts of blocks. Most have more than a hundred blocks, and each block contains several layers. In figure 3.3 we show the schematics of InceptionResNetV2, which is a model containing many layers. From the schematics, we can see that knowing where to draw the line separating the layers that should be trained from the layers we want to keep is not trivial. Intuitively, the separation should be at the connection between two convolutional blocks, as each block possess image features unique to the block. Splitting in the middle of a block might break these image features. Nevertheless, we will find out through automatic hyperparameter optimization.

3.4 Proposed System

To be able to test the effects of automatic hyperparameter optimization, we had to build experiments running optimization instances. In the following section, we will describe how we designed and implemented the experiments. We will also go into detail on libraries used and discuss design decisions.

3.4.1 Libraries

We relied heavly on libraries to create our tests. We used state of the art libraries for machine learning, recognized by corporations and researchers around the world. Additionally we used a library for the Bayesian optimization. These libraries are described in the sections below and listed in table 3.4.

TensorFlow

TensorFlow is an open source library for numerical computation using data flow graphs and is available on Windows, Linux, Mac OS and Android [30]. The graph nodes represent mathematical operations, while the graph edges represent the multidimensional arrays that flow between them. These multidimensional arrays are called tensors, which is where TensorFlow got its name. Tensorflow is designed for large-scale distributed training and inference and is flexible enough to support experimentation with new machine learning models and system-level optimizations [1, 136]. For example, it can efficiently use hundreds of powerful servers for fast training and at the same time run locally on mobile devices [2]. TensorFlow is created and maintained by Google, and is used in many Google products.

The architecture of the TensorFlow platform is illustrated in figure 3.4. It is built on layers of abstraction, where the most notable is the C API. The C API separates the user-level code from the core runtime, and allows for implementations in different languages. The most notable one is the

Table 3.4: Overview over important libraries. The table is not exhaustive, but lists the main libraries used to create the experiments.

Main Libraries			Other Libraries		
Name	Version	Language	Name	Version	Language
TensorFlow [1]	1.4.1	Python	Cuda [87]	8.0.61	C
Keras [25]	2.1.4	Python	cuDNN [22]	6.0.21	C
GPyOpt [6]	1.2.1	Python	GPy [48]	1.9.2	Python

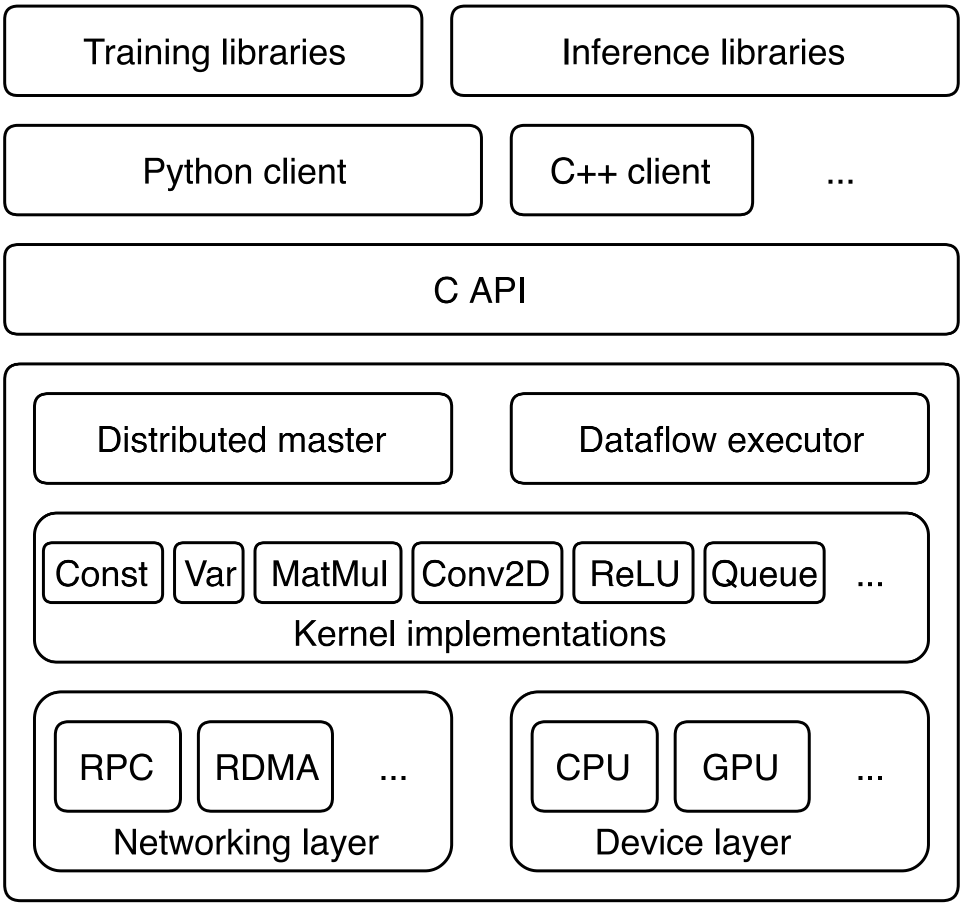


Figure 3.4: Illustration of the general architecture of the TensorFlow runtime library. The top layers of the illustration shows higher levels of software abstraction and the bottom layers show the lowest levels of abstraction. This is a remade version of the illustration available on the official web page for TensorFlow and in the paper by Abadi et al. [2, 136].

Python implementation, but there are also implementations in C++, Java, Javascript, Go and Swift. In addition, bindings for other languages such as Rust, Ruby and Scala are in the works [136]. As shown in the illustration, the architecture resembles that of an operating system, and as such the heart of the platform is the kernel. The kernel contains over 200 standard operations, inluding arithmetic, array manipulation, control flow, and state management operations. These are implemented to use efficient parallel code for both CPUs and GPUs [2]. In the illustration in figure 3.4, the device layer mentions GPU. In this thesis, we will use multiple GPUs, and as such, TensorFlow utilizes Cuda [87] and cuDNN [22], which are Nvidia's general GPU API and deep learning GPU API, respectively. Even though we do not use TensorFlow, Cuda or cuDNN directly, they are important for the overlying libraries that we use directly. We could have run with other underlying libraries, but these are the ones best serving our goal, so we have listed them in table 3.4.

TensorFlow is well-known for its use on machine learning problems and its great performance and resource utilization. Tensorflow is adopted by many projects and has an extensive community of collaborators, researchers and entusiasts [2]. As a result, there exists several resources and unofficial extensions to TensorFlow either extending functionality or making it easier to use by abstracting some parts of the platform. One such extension is the Keras library [25]. It started as a wrapper library, creating a higher abstraction layer over several neural network libraries, but has since become part of the TensorFlow distribution. Because of the performance on GPUs and its support in Keras, TensorFlow was chosen as the platform for the machine learning models and the training in the thesis. However, every call to the library is done through the Keras API. This is due to the nature of our experiments. We don't make the models ourselves as they are available pre-trained from Keras, so we only need the functionality offered by the Keras API. By using TensorFlow as Keras' tensor manipulation library, we also have access to TensorBoard [29]. TensorBoard is a visualization tool to visualize scalars such as validation accuracy and loss. Moreover, it can visualize a graph of the trained model. We use this tool to display some of our results later in this thesis.

Keras

Keras is a high-level neural networks API running on top of either TensorFlow [1], Microsoft Cognitive Toolkit (CNTK) [122], or Theano [3]. Chollet, the creator of Keras [25], recognized the lack of options for higher leveled machine learning libraries. Keras aims to allow for fast experimentation and to be able to go from idea to result with the least possible delay. The library follows four guiding principles: (1) User friendliness, meaning the Keras API focuses on user experience and accessibility. (2) Modularity. Creating models is combining components such as neural layers, cost functions, optimizers and initialization schemes. (3) Easy extensibility, which means that adding new classes and functions are easy if needed. (4) Work with Python.

Keras creates a standard way to use functionality across machine learning libraries, and, the exception being some functionality unique to TensorFlow, changing the underlying library is as easy as changing a setting. The primary reason for TensorFlow getting special treatment is due to Keras being implemented directly into the TensorFlow library. The core data structure of Keras is a model, which is a way to organize machine learning layers. There are two ways to make models in Keras: (1) Using the sequential model, which is a linear stack of layers used for simple models and (2) the functional model API, which allows for building arbitrary graphs of layers used for more complex architectures [25].

We used Keras in this thesis because of the simplicity and efficiency provided. Keras is easy to set up and supports convolutional neural networks. Additionally, it has several built-in optimizers and contains an application module including state-of-the-art CNN models pre-trained on the ImageNet dataset. We have listed these pre-trained models and their overlying properties in table 3.2. It is a Python library, but using the TensorFlow library, the performance is impeccable. It would be possible to use TensorFlow directly, but considering the convenience of Keras and the available pre-trained CNN models, our choice landed on Keras. In this thesis, we use Keras by taking pre-trained CNN models, which are Keras model instances, and use functions available by the model to replace the classification block. From there we use different functions available in the model instance to train and later fine-tune the model.

GPyOpt

GPyOpt [6] is a Python open-source library for Bayesian Optimization. It is based on GPy [48], which is a Gaussian processes framework in Python. Groups from the University of Sheffield developed both frameworks. GPyOpt performs global optimization of black-box functions using Gaussian processes. A black-box function is a function where we don't see what happens internally, and can only affect the output by changing the input. GPyOpt can optimize with different acquisition functions. It can be used for both the automatic configuration of models and machine learning algorithms. Additionally, it can be used to optimize physical experiments both sequentially or in batches. It is also able to handle large datasets via sparse Gaussian process models.

In the thesis, we use GPyOpt for its capability of running Bayesian optimization trivially. GPyOpt solves the issue of having to implement and configure a Bayesian optimization algorithm. The library can run Bayesian optimization without any configuration, which makes it use default values, or it can run with different user-specified configurations. Table 3.6 provides a table of selected relevant parameters available to configure the Bayesian optimization and their default values. The parameters listed are not all the possible parameters but are the ones most relevant to our use case.

There are several libraries for Bayesian optimization available online that we could have used, but we landed on GPyOpt. The reason for this is that GPyOpt is that the Machine learning group at the University of

Sheffield is still maintaining the repository, it is trivial to use, and the quality is high. It has much more functionality then what we use, but future research can make use of this functionality to improve the results. We use the GPyOpt library for instancing a Bayesian optimization object. This object takes several parameters. However, we only use some, and the rest of the parameters we leave to their default values. Table 3.6 provides a list of the most relevant parameters and their default values. In addition to these, there are the required parameters. These are the function we want to optimize and the domain from where to extract hyperparameters. Figure 3.7 shows how the Bayesian optimization object wraps the function we want to optimize. The optimized function takes one parameter, which is a NumPy array. This array contains the hyperparameter values for the current optimization iteration. The values are taken from the domain specified. Furthermore, the optimized function returns a value, which the Bayesian optimization object use to evaluate the optimization and select new hyperparameters for the next iteration. The optimization run is started from a function in the Bayesian optimization object. We supply this function with the number of iterations it should run and wait for the result. At the end of the run, the function saves a report on the model and evaluation values for each iteration and the summary of the optimization run. Additionally, it saves a plot for the acquisition function and the optimization convergence.

3.4.2 System Description

The system for running experiments and the experiments themselves are written in Python. Python is a versatile language, but the primary reason for the utilization of Python is that the libraries we use are written in this language. The system is designed for easy interchangability with other datasets and hyperparameter bounds. The user experience of the system is made with a researcher with programming skills in mind as running other optimizations than the optimizations we run may require changes to some functions. However, the structure and majority of the system can be left as is. The system as we use it is illustrated in figure 3.5. The figure gives an overview of the system flow. From the illustration one can see that datasets and defined hyperparameter bounds are fed to the test-suite, which runs one or all of the optimization strategies. After one or all of the optimization strategies have finished their run, the results are written to disk. The results describe the optimization and gives the best hyperparameters of the best trained model. The system running the experiments is structured as follows:

Arguments decide dataset and experiment settings Arguments such as name of experiment, type of run and dataset directory are parameters that decide how the system will run. Which hyperparameters to test and other parameters impacting the optimization at a lower level are not included here.

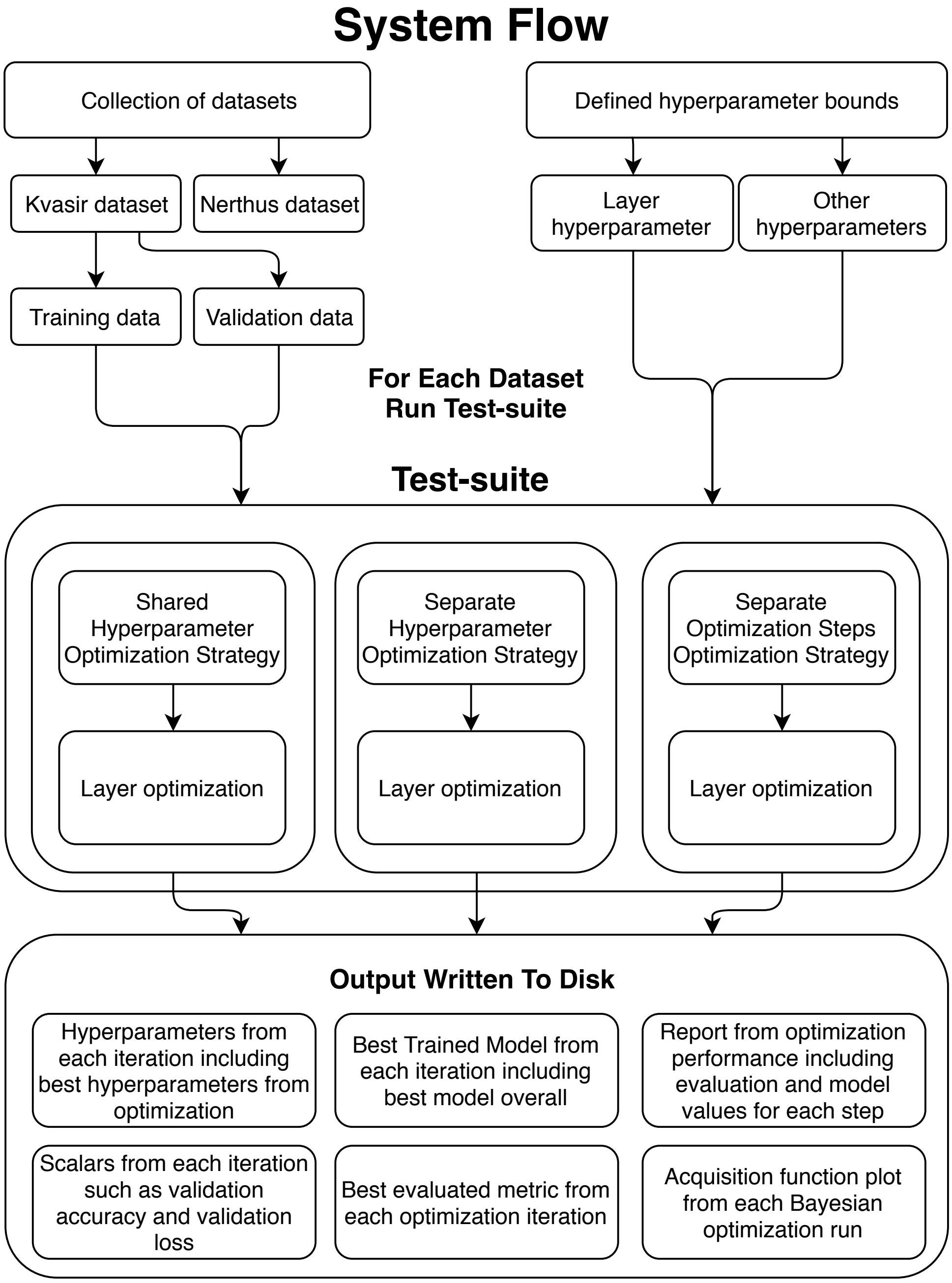

System Flow
Collection of datasets
Kvasir dataset
Nerthus dataset
Training data
Validation data
Defined hyperparameter bounds
Layer hyperparameter
Other hyperparameters
For Each Dataset Run Test-suite
Test-suite
Shared Hyperparameter Optimization Strategy
Separate Hyperparameter Optimization Strategy
Separate Optimization Steps Optimization Strategy
Layer optimization
Layer optimization
Layer optimization
Output Written To Disk
Hyperparameters from each iteration including best hyperparameters from optimization
Best Trained Model from each iteration including best model overall
Report from optimization performance including evaluation and model values for each step
Scalars from each iteration such as validation accuracy and validation loss
Best evaluated metric from each optimization iteration
Acquisition function plot from each Bayesian optimization run

Figure 3.5: Overview of the proposed system flow. Datasets are fed along with defined hyperparameter bounds to the test-suite. For each dataset, the test-suite runs one or all of the optimization strategies, depending on the program arguments. Each optimization strategy ends with a layer optimization, which writes results from the training together with optimization statistics and plots to disk.

Three different optimization strategies per dataset There are three different optimization strategies to run. For each dataset, we run all or one of them, depending on the argument given to the program.

Bayesian optimization chooses hyperparameters We utilize Bayesian optimization for each optimization. It is a wrapper around a function running one test instance. The test instance function gets hyperparameters from the Bayesian optimization algorithm each time it is called. The hyperparameters are chosen from a dictionary of hyperparameter bounds decided by us.

Pre-trained model is created The pre-trained model is a hyperparameter. We remove the classification block and replace it with a pooling layer and a classification layer with the same amount of outputs as there are labeled categories.

Classification layer is tuned We tune the classification and pool layer only. All pre-trained layers are not trained. Hyperparameters are given by the Bayesian optimization.

Fine-tuning of layers separated by default 2/3rd layer After the classifier is trained, we use hyperparameters from the Bayesian optimization to fine-tune the model. The delimiting layer is a default value of 2/3rd the length of the model. We don't care about the delimiting layer in this step, and only want to find the best hyperparameters among the other hyperparameters.

Fine-tuning with best hyperparameters for finding best delimiting layer After finding the best hyperparameters, we want to find the best delimiting layer. We fine-tune the network again, but this time use the best hyperparameters from the previous step and get the delimiting layer from the Bayesian optimization.

Best validation accuracy is used for optimization The Bayesian optimization chooses hyperparameters based on the optimization instance that yields the best validation accuracy.

Following is a detailed description of the system:

3.4.3 Test Suite

The test suite is designed to run the tests sequentially from start to finish. It automatically creates a log directory if not already existing. It also creates a new test folder for storage of each test result. For test-suite configuration, the program takes arguments from the user. A configuration file, which is typically used for configuration, is not needed as the experiment is the same for each optimization strategy. There are five arguments available:

1. **The name of the test-file for the run.** The folder can't already exist. Naming the test-file is an alternative to having the test-suite generating the name.

2. **The type of test run.** This refers to which optimization strategy to use. The options are all or one of them.
3. **How many GPUs to use.** If there is more than one GPU available to the system, there is the alternative to use them. The default value is one GPU.
4. **Disable nonconvergence filtering.** The default value is that the nonconvergence filtering callback is enabled. Nonconvergence filtering is canceling a training early because the training failed to reach a given threshold in a given number of epochs. We introduce nonconvergence filtering in detail in section 3.5.
5. **Resume running tests in a given folder.** The folder must already exist. The tests have a checkpoint system in case they shut down early, and the resume option is an extension to this. Option 5 is mutually exclusive with option 1.

There are some global constants available to the test-suite. These are the directory of the datasets and a list of models and optimizers that the hyperparameter optimizer selects from when optimizing. The test-suite iterates through each dataset and sets up the folder structure of each test when running them. For each dataset, it runs each optimization strategy sequentially unless the arguments specify a specific strategy. The test-suite times the whole run, and logs metadata such as the arguments and constants.

3.4.4 Optimization Strategies

There are three optimization strategies available to us:

1. Share one set of hyperparameters on both the training of the classification block of the pre-trained CNN and the fine-tuning of the pre-trained CNN onto the dataset.
2. Have two separate sets of hyperparameters, one for each type of training.
3. Do the optimization in two steps:
    (a) Optimize the training of the classification block until you get the hyperparameters yielding the highest validation accuracy. The optimization will use one set of hyperparameters for this step.
    (b) Use the best model from the previous step and optimize the fine-tuning. This step will also have its own set of hyperparameters. At the end of each strategy, we use the same function for layer optimization.

The system implements the optimization strategies through two functions. The first function performs both the shared hyperparameter and separate hyperparameter optimization strategies. The user can decide the optimization strategy by passing a boolean to the function. The similarity for

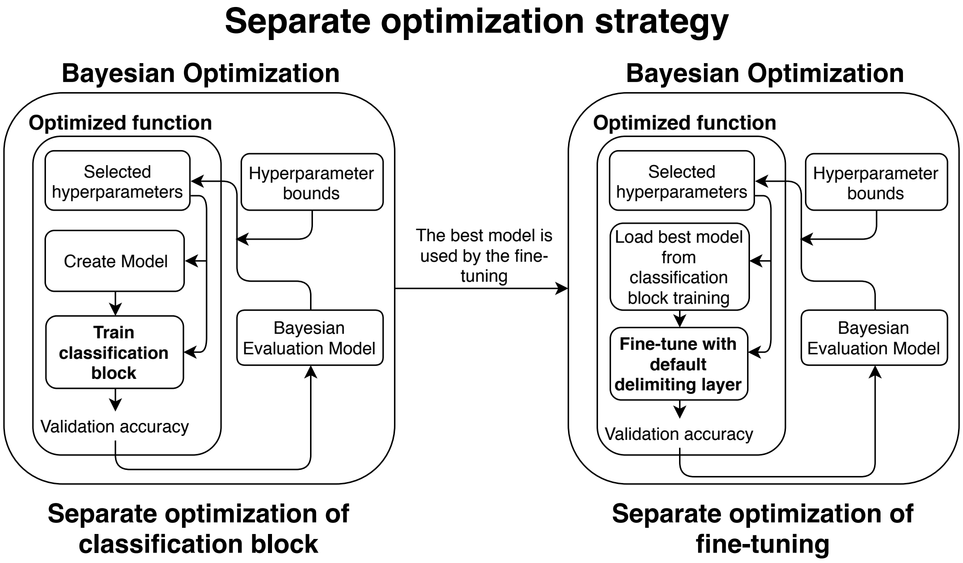

Separate optimization strategy
Bayesian Optimization
Optimized function
Selected hyperparameters
Hyperparameter bounds
Create Model
Train classification block
Bayesian Evaluation Model
Validation accuracy
Separate optimization of classification block
The best model is used by the fine-tuning
Bayesian Optimization
Optimized function
Selected hyperparameters
Hyperparameter bounds
Load best model from classification block training
Fine-tune with default delimiting layer
Bayesian Evaluation Model
Validation accuracy
Separate optimization of fine-tuning

Figure 3.6: Overview over the separate optimization stategy. The figure shows two instances of Bayesian optimization running sequentially. The left box represents the first optimization run. Here the goal is to train the replaced classification block of the pre-trained layer and optimize until the best results are achieved over a set number of iterations. When the Bayesian optimization algorithm finishes, the model yielding the best results are sent to the next step, represented by the box on the right. Here the goal is to do the fine-tuning with a default limiting layer. The optimization algorithm is finished when it reaches the set number of iterations or it can no longer improve the results.

Table 3.5: Hyperparameter sets for each optimization strategy.

Optimization strategy	**Optimization step**	**Hyperparameter set**
Shared Hyperparameters		Model Type Shared Optimizer Type Shared Learning Rate
Separate Hyperparameters		Model Type Classification Optimizer Classification Learning Rate Fine-tuning Optimizer Fine-tuning Learning Rate
Separate Optimizations	Classification block optimization	Model Type Optimization Type Learning Rate
	Fine-tuning optimization	Optimization Type Learning Rate

both optimization strategies is that the function creates a function to pass to the Bayesian optimization library. The function passed is what is optimized, and as such contains the training procedure. We call this function the optimization function, and it takes only one parameter; a NumPy [88] array containing the hyperparameters selected by the Bayesian optimization algorithm for a specific iteration. The optimization function unpacks the hyperparameters from the NumPy array and sends them to the next step where they are used for training. The only difference between running the function in share hyperparameter mode versus separate hyperparameter mode is the number of hyperparameters. In shared hyperparameter mode, we have a set of three hyperparameters: The model, the optimizer, and the learning rate. We then copy the optimizer and learning rate to get two sets of hyperparameters. In separate hyperparameter mode, we have a set of five hyperparameters: The model, the optimizer and learning rate for the classification block training, and the optimizer and learning rate for the fine-tuning. Figure 3.7 shows an illustrated overview of the optimization function.

3.4.5 Bayesian Optimization

The Bayesian optimization class running the test instances and evaluating the results is a class available from the GPyOpt library [6]. This class takes

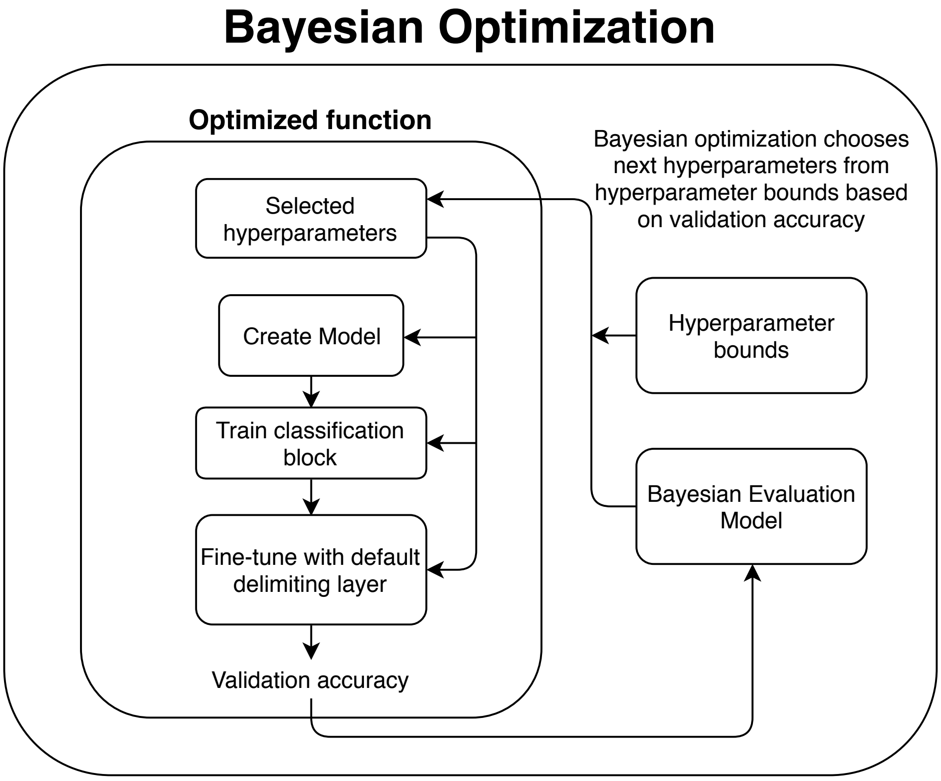

Bayesian Optimization
Optimized function
Selected hyperparameters
Create Model
Train classification block
Fine-tune with default delimiting layer
Validation accuracy
Bayesian optimization chooses next hyperparameters from hyperparameter bounds based on validation accuracy
Hyperparameter bounds
Bayesian Evaluation Model

Figure 3.7: Overview over how Bayesian optimization is implemented in the system. The optimized function is called for every iteration by the Bayesian optimization function. The hyperparameters for the optimized function is chosen from hyperparameter bounds and by the Bayesian optimization. It is therefore highly probable, almost guaranteed, that the hyperparameters are different for each iteration. This particular illustration is of a shared hyperparameter or separate hyperparameter optimization strategy. In this case the model hyperparameter is sent to the create model part of the system, while the optimizer and learning rate is distributet among the next parts of the system. Finding the optimal delimiting layer is done in a separate optimization.

Table 3.6: List of selected relevant default parameters given to the Bayesian optimization function available in the GPyOpt library [6].

Parameter Name	**Description**	**Default Value**
model_type	Type of model to use as surrogate	'GP' (Standard Gaussian process)
acquisition_type	Type of acquisition function to use	'EI' (Expected improvement)
acquisition_optimizer_type	Type of acquisition optimzer to use	'lbfgs' (L-BFGS)
initial_design_type	Type of initial design	'random' (To collect points in random locations)
initial_design_numdata	Number of initial points that are collected jointly before start running the optimization	5
model_update_interval	Interval of collected overservations after which the model is updated	1
normalize_Y	Whether to normalize the outputs before performing any optimization	True
batch_size	Size of the batch in which the objective is evaluated	1

several parameters affecting the optimization such as the type of model to use as a surrogate, the type of acquisition function to use, the type of acquisition optimizer, and more. However, to limit the scope of the thesis, default values were used for all of these parameters. It would be interesting for future work to adjust these values for perhaps even better results. The options for each parameter can be found in the documentation for the Bayesian optimization module in the GPyOpt library [6]. The relevant default parameters we used are listed in table 3.6. Nevertheless, the parameters that we had to supply was the function to optimize and the domain, namely the hyperparameters and their boundaries.

The Bayesian optimization class acts as a wrapper for the function to optimize. Figure 3.7 illustrates Bayesian optimization. Any calls to the training function are done through this class. After creating an instance of the class with the given parameter configuration, which in our case is the default one, a call to the class' *run_optimization* function will start the optimization steps. This function will run until it has reached a given number of iterations after the initial exploration data. When done, the function will plot the acquisition function and convergence from the run as well as report evaluation and model values for each iteration. It will also give a summary of the optimization.

It is the function to optimize that is the transfer learning. Figure 3.7 also illustrates this function for the shared hyperparameter optimization strategy. For every iteration, this function is run from scratch. The only difference between each run is the hyperparameter values it selects. These come through the function's sole parameter. This parameter is a NumPy array containing the name of the hyperparameter and the value chosen by the Bayesian optimization from the hyperparameter bounds defined by us. The format of the hyperparameter value is a number from the bounds, and as such we must convert it to the correct format. The hyperparameter bounds, therefore, need a conversion function along with each hyperparameter. For example, the model hyperparameter will be an index from the defined list of models. The conversion function will then convert the index into the actual class representing the model. The Bayesian optimization uses the return value from the function to optimize to select the hyperparameters for the next iteration. The return value is therefore in our case the validation accuracy from the training.

3.4.6 Model Setup

From figure 3.7 one can get an impression of how we structured the optimized function. After receiving the hyperparameters from the Bayesian optimization, the first step is to create an instance of the pre-trained model and to replace the classification block. In this step, we utilize Keras' application module [24]. The application module has several pre-built CNN models, pre-trained on the ImageNet [60] dataset. These are well-known, state-of-the-art models for image classification. We have listed the ones we use in table 3.2. These are not all of the available models, but we chose to limit the number of models by not including the NASNetMobile and NAS-

NetLarge [145] models which were made available with version 2.1.3 of Keras as these were heavy to run and we did not want increase the search space further. We also did not include the MobileNet [56] model as it required a different image size than the other models and was reported to perform worse on ImageNet in Top-1 accuracy and Top-5 accuracy [25].

When creating the instance of the model, we make sure to remove the top block of the model. This block is the classification block and is by default tuned for 1000 classes in the ImageNet validation dataset. Our datasets are far from that size, so we must replace the block with a classification block tuned for the number of classes in the dataset we want to use for fine-tuning. Additionally, we signal the instance that we want to use weights pre-trained from that model's best training on ImageNet. The alternative would be randomly initialized weights, but that would mean we would train the model from scratch instead of using transfer learning.

The new classification block we put on the model is the same for every model instance. We add a layer doing global max pooling for spatial data and then a dense layer with softmax activation. Adding this structure to the end of a CNN is the solution of most CNN models. The idea is to generate one feature map for each corresponding category of the classification task in the last layer of a convolutional block. The max of each feature map is then added to a vector and fed directly into the softmax layer. Using this structure has the advantage of enforcing correspondences between feature maps and categories, and as such, the feature maps can be easily interpreted as categories confidence maps [79,129].

After creating the instance of the model, we create generators for the validation dataset and the training dataset of the model. For the training dataset, we only flip the images horizontally and vertically. We rescale the images to fit with our models. The image generator is a class available from the Keras library, and its function is to process new data automatically when training and validating. It has a function that creates a flow from a dataset directory. It takes as parameters the directory of the training or validation dataset, the target size of the images, size of the image batch that is loaded at a time, and the class mode. We set image width and height to 299 and batch size to 16. We set the class mode to categorical, meaning we get categories from the directories where the images are loaded.

3.4.7 Classification Block Training

After instancing the pre-trained model and replacing the classification block, it is time to train the classification block. It is necessary to train the classification block before fine-tuning because the weights of the classification block layers are randomly initialized after the replacement. Randomly initialized weights can not classify. For training CNN models from scratch, the weights of all layers are randomized. As it trains, the weights are tuned until it is able to classify most images correctly. However, when using a pre-trained CNN model, we do not want to train every layer again as we then lose knowledge attained on the bigger dataset. The reason for replacing the classification block is to let it classify features into the number of classes that our dataset has. Training this block lets the CNN model classify into the classes in the new dataset using only the pre-trained knowledge. If we were to use the new classification block without training it first, we would probably not be able to achieve good results when later fine-tuning. At the very least, it would take much longer as the balance in the network would be off. With an untrained classification block, each iteration would give bad results because of the classification block and not necessarily because of the image features being bad.

When training the classification block, we mark all layers from the base model as untrainable, so we only change the weights of the block. Then we compile the model with the optimization function and learning rate, given as a hyperparameter by the Bayesian optimization. We create a callback list which we will pass to the training function containing the following callbacks:

1. **Logging.** We use the *CSVLogger* callback available in Keras for logging scalars for each epoch in a CSV file.
2. **Saving the best model.** We use the *ModelCheckpoint* callback available in Keras for saving the best model weights. These model weights are used for the fine-tuning and also for reference and calculation of performance metrics. The best model is the model with the best validation accuracy in this work, but it could also have been based on other scalars.
3. **TensorBoard support.** To enable the use of TensorBoard, we need to generate a TensorBoard events file. This callback generates and updates such a file for every epoch in one training. Doing this makes it possible for us to visualize the training through live-plotting of the scalars.
4. **Early-stopping.** We use the *EarlyStopping* available in Keras. Early stopping is a regularization technique to avoid overfitting neural networks when training [17, 142]. In our case, the callback evaluates the validation accuracy. If the validation accuracy stops increasing over five epochs, we end the training early.
5. **Nonconvergence filtering** When optimizing, the Bayesian optimization will try many different combinations of hyperparameters. Many of these will be useless, and the training will never converge to a good result. Instead, they will stay at low validation accuracies and spend time. A full optimization takes several days, so we needed ways to speed this up. Therefore, we introduced our callback that stops a training run if the monitored scalar fails to reach a certain threshold in a certain number of epochs. We call this callback "*nonconvergence filtering*", and it is explained in detail in section 3.5. In our case, we evaluate validation accuracy. We set the threshold to 0.5 and patience to 5 epochs. Patience is the number of steps before evaluating the scalar against the threshold. There is a program argument to turn this feature off, but the default behavior is to have it enabled.

After creating the callback list we start fitting the classification block on the training dataset. We use the model object's Keras function *fit_generator*, which starts the training. The training runs for ten epochs if not stopped by either early stopping or nonconvergence filtering. The number is somewhat arbitrary. The idea behind is that the training of the classification block should converge quickly as it already has the image features pre-trained. Additionally, the point of training is not the achieve the perfect classification block, but rather to have a block capable of classification enough for fine-tuning to work. The highest validation accuracy reached is returned to be used in the separate optimization strategy. The other optimization strategies discard this value.

3.4.8 Fine-tuning

After training the classification block, the next step is to fine-tune the CNN model to the dataset. In this step, we want to find the best hyperparameters except for the delimiting layer. We will find the delimiting layer in the final step. Technically, it is possible to find the layer in the same optimization step, as we can measure the number of layers from the classification block run. That number will not change. However, we see it as an advantage to be able to decrease the dimensionality and the search space of the hyperparameter domain. To the best of our knowledge, the other hyperparameters are not affected by the delimiting layer and vice versa. Therefore, we can, without consequences to the performance, split the optimization into two steps. The other hyperparameters cannot be split the same way as, for example, the learning rate will be dependent on the model optimization function. Nevertheless, we split the optimization into two steps, but we still need to train certain layers and keep others pre-trained. Therefore, we use a default delimiting layer value. To keep the results of each model moderately consistent, we decided to give a value based on the length of the model, rather than a constant. Somewhat arbitrarily, we decided on having the delimiting layer be the layer two-thirds of the model length. That being $\textit{delimiting layer} = \textit{model length} \times \frac{2}{3}$. Our reasoning is that two-thirds of the model intuitively should hit close to the balance of keeping low-level abstraction image features and training the high-level abstraction image features on our dataset. We can unfortunately not verify this without optimizing the delimiting layer. Nevertheless, for our purpose of being a default value, it is sufficient.

In the fine-tuning step, the first thing we do is to load the model with the top weights from the classification block training. At this point, we have the pre-trained weights of the base model and the trained weights of the classification block. After loading the weights, we set all layers that precede the delimiting layer to untrainable and set all layers including the delimiting layer to trainable. From here we do all the same steps as in the classification block training. We create the same callback list, compile the model with the hyperparameters from the Bayesian optimization, and start the training. See section 3.4.7 for how we did this in the classification block step. The hyperparameters from the run with the best result are what we use for the next step, which is layer optimization.

3.4.9 Layer optimization

The final step of any of the optimization strategies is optimizing the delimiting layer. As discussed in the previous section, section 3.4.8, we split the optimization of the fine-tuning hyperparameters into two Bayesian optimization runs. We do this mainly to reduce the dimensionality of the search space available to the Bayesian optimization. It is possible to do this as we assert that the delimiting layer does not affect the effectiveness of the other hyperparameters. For other the other hyperparameters we use, this is not true, so they need to be run in the same optimization run.

At the beginning of this step, we have found the best hyperparameters, except for the delimiting layer, for both the classification block training and the fine-tuning. We have consequently also found the best weights for these runs. However, we want to increase the performance of the model, and our default delimiting layer is likely far from the optimal. We do this through a separate optimization where we train the fine-tuning again from the start, depicted in figure 3.8. However, this time we use the hyperparameters we found to be training the best performing CNN model instead of getting them from the Bayesian optimization. The only hyperparameter the Bayesian optimization provides is the delimiting layer, which is taken from a hyperparameter bound of the first layer to the last layer of the model. This optimization is one-dimensional with a relatively small bound, which makes the optimization potentially very effective at finding the global minima. The result of the layer optimization is a model with the optimal hyperparameters, including the delimiting layer, and the best performing CNN model, found by the Bayesian optimization algorithm automatically. How well the model performs, is dependent on the random nature of the optimization. Different runs will consequently produce different results.

3.5 Nonconvergence filtering

It is in the nature of automatic optimization to test combinations and values that absolutely do not work. One of the toughest parts of hyperparameter optimization is how volatile the performance output is between test iterations. The relationships between the pre-trained model, the optimizing function, and the learning rate is filled with local minima, and as such, a change in one can make a huge impact on the performance of the CNN model. For example, a combination of the pre-trained Xception model with Nadam might give good results. Changing the optimizer or the pre-trained model to something else might make the results significantly worse. The consequence of this volatility is that we will have many optimization iterations that will produce inferior results and fail to converge to better results over epochs. Running any of the optimization strategies could take up to days, depending on the number of iterations per optimization,

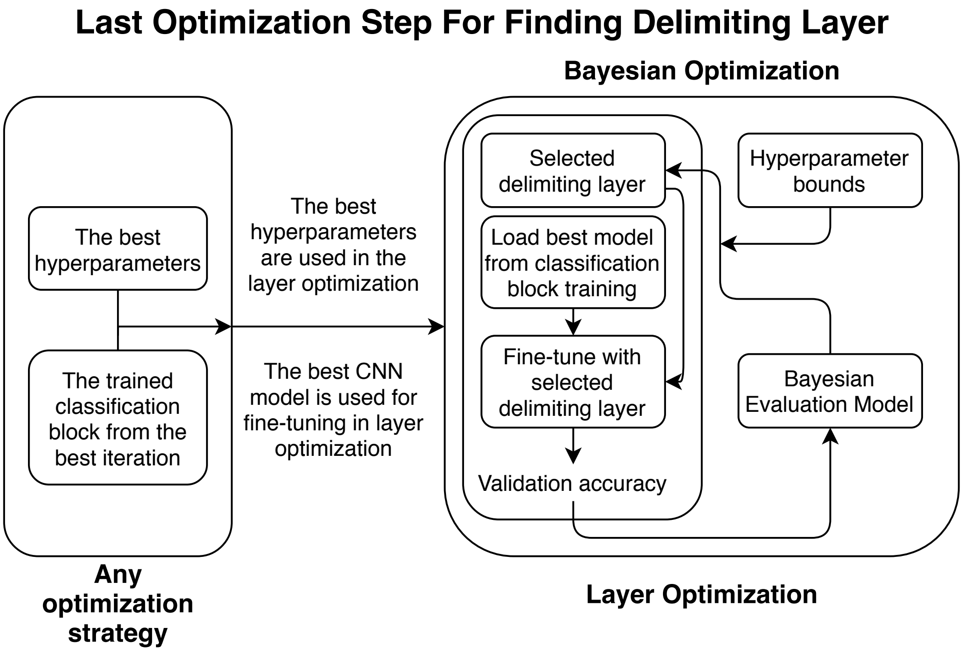

Last Optimization Step For Finding Delimiting Layer
Bayesian Optimization
The best hyperparameters
The trained classification block from the best iteration
Any optimization strategy
The best hyperparameters are used in the layer optimization
The best CNN model is used for fine-tuning in layer optimization
Selected delimiting layer
Load best model from classification block training
Fine-tune with selected delimiting layer
Validation accuracy
Hyperparameter bounds
Bayesian Evaluation Model
Layer Optimization

Figure 3.8: Overview over the last optimization step for finding the delimting layer. The figure shows the final step for all optimization strategies. From any of the optimization strategies we get the best hyperparameters with the expection of the delimiting layer. In addition we get the best classification block model from the best run and use this and the hyperparameters to optimize the delimiting layer. The Bayesian optimization chooses a delimiting layer, which is then used for fine-tuning. The validation accuracy is then measured and used for evaluating the next optimization step.

```python
class NonconvergenceFiltering(Callback):
    def __init__(self, monitor, threshold, patience):
        super(CancelBadResultEarly, self).__init__()
        self.monitor = monitor
        self.threshold = threshold
        self.stopped_epoch = 0
        self.patience = patience
        self.wait = 0

    def on_epoch_end(self, epoch, logs=None):
        current = logs.get(self.monitor)
        if current < self.threshold:
            if self.wait >= self.patience:
                self.stopped_epoch = epoch
                self.model.stop_training = True
            else:
                self.wait += 1
        else:
            self.wait = 0

    def on_train_end(self, logs=None):
        if self.stopped_epoch:
            print('Epoch %05d: stopped by '
                  'nonconvergence filtering' %
                  (self.stopped_epoch + 1))
```

Listing 3.1: Python code of callback for Keras that stops early if the monitored value is below the given threshold in a given number of epochs. We call this technique nonconvergence filtering. The monitor parameter is the value we use for evaluation, the threshold is the threshold the monitored value is required to pass, and the patience is the number of epochs the monitored value is allowed to be under the threshold before being stopped. The presented version is stripped down to only work with accuracies and not loss as that was our use case.

so being able to cancel the runs that do not work early is an advantage. Not only does it save time, but we can fill that time with more Bayesian optimization iterations for a chance at better results.

We, therefore, introduce a solution to the problem called *"nonconvergence filtering"*. Nonconvergence filtering is a technique for stopping training runs that fail to produce a metric which value reaches a given threshold in a given number of steps. The technique filters out training runs that fail produce metrics that converge to the desired metric space in the desired number of steps or epochs, hence the name. Nonconvergence filtering is inspired by early stopping [17] as it stops the run earlier than the planned number of epochs, and as such, the implementation draws inspiration from the code for early stopping in Keras [67]. We implemented nonconvergence filtering as a callback to be used in Keras, and present it in listing 3.1. The presented implementation is in Python and is the one we use in the proposed system. However, nonconvergence filtering can be applied to other use cases and implemented in other languages. Our presented implementation is simplified to only work with the validation accuracy metric, nevertheless, expanding the function to include other metrics is trivial, and inspiration can be drawn from how Keras implements early stopping [67].

Nonconvergence filtering draws inspiration from early stopping but solves different problems. Early stopping is a method for detecting when the training of a model has stopped improving. To evaluate the model the early stopping compares the accuracy or loss of the model when classifying the validation dataset. If the validation accuracy or validation loss fails to improve over a given number of epochs, early stopping terminates the training run. This is done to prevent the training run from training too long, as training too long will cause the model to overfit on the training dataset, making the model unable to classify images that are similar, correctly. Nonconvergence filtering tackles the problem of training runs which we early see will not be able to train to achieve acceptable results and are running for several epochs more than necessary. These situations happen very often when doing automatic hyperparameter optimization, but it could also work for other cases where hyperparameters are tuned, or new models are tested. In our results, we have found we save considerable amounts of time by using the nonconvergence filtering technique.

On April 28th, 2018, a Keras API Design Review was created. It was a draft of a proposal to add the same functionality as nonconvergence filtering brings to Keras [118]. On May 3rd, a pull request was approved and merged into the master branch of Keras [119]. The approach proposed in the design proposal was nearly identical to our approach in listing 3.1, and the signature of the init-function contained monitor, threshold, and patience. Nonconvergence filtering was implemented 20th of Match in our codebase before it was added to Keras, and it is unfortunate that our work came out at a later date as it can imply that we copied them. Regardless, the version implemented by Keras is called EarlyBaselineStopping, and it was implemented a little differently than suggested by the design proposal. Instead of creating a new class, it was implemented as part of Early stopping. In new versions of Keras, it is, therefore, possible to use the functionality of nonconvergence filtering by setting a parameter called baseline, which is the same as the threshold parameter in nonconvergence filtering. Our solution has the advantage of being useful for older versions of Keras. However, new versions should rely on the EarlyStopping class available in Keras.

3.6 Summary

The methodology chapter aims at giving an in-depth description of the methods applied to achieve the goal of creating a system for running our experiments. We present the datasets Kvasir and Nerthus. The datasets are both small compared to the huge datasets, such as ImageNet, required for training a CNN model from scratch, but they are perfect for fine-tuning. Kvasir shows eight classes of images of diseases and abnormalities in the GI tract, while Nerthus is a smaller dataset that is showing four classes of GI tract cleanliness. Both datasets have artifacts in the images, and each class of Nerthus is from different locations of the colon, but they are the best we have available and are sufficient for our use case.

We discuss the metrics we use for presenting the results of the thesis. We chose validation accuracy over validation loss as the metric to optimize. We talk about our choice of splitting the datasets into a test and validation set, and to not split into three datasets. We also present the metrics suggested in the Kvasir dataset paper. These are metrics known from statistics and often used in related work. We use these metrics to present our results.

We discuss our choice of hyperparameters to optimize and present each of them. We talk about how we want the dimensionality and size of the search space to be as low as possible to increase our chances of reaching the best results. We present each hyperparameter:

- **Pre-trained model.** Keras has several well-known pre-trained CNN models available. These are trained on the ImageNet dataset, and the weights are from the best training done by the researchers that created them or Keras themselves. The best model might be different for different datasets, and the nature of the model means we treat them as black-boxes. We want to know which model is the best for our problem, so we use the pre-trained model as a hyperparameter.

- **Model optimization function.** By model optimization function we mean the gradient descent optimizing algorithm. Similarly to the pre-trained models, there are several optimizers available in Keras. An optimizer is required for compiling the CNN model. The optimizers act as black-boxes, and we do not know which optimizer works best with what model. We want to find the best combination automatically, so the optimizer is a hyperparameter.

- **Learning rate.** Including the learning rate as a hyperparameter was not a trivial choice. There are arguments to be made for both decisions. The disadvantage of including it is that we increase dimensionality and the search space. Additionally, there are default values for each optimizer. The tradeoff of dimensionality and search space size when we could have used default values that might work good, might not be worth it. Some of the optimizers are also implemented to adjust the learning rate automatically, and it is adviced against changing the default values for these optimizers. However, we believe the automatic hyperparameter optimization will be able to achieve a better or at least the same learning rate. Additionally, fine-tuning and classification block training might require different learning rates.

- **Delimiting layer.** The pre-trained CNN models are complex and knowing at which layer to separate the model into a part we train and one we do not train is nontrivial. Intuitively, one would split the model on the connection between two convolutional blocks, but hyperparameter optimization might find a layer which is unintuitive to humans.

We introduce our proposed system. The proposed system is made for running automatic hyperparameter optimization experiments. It is built on Keras and TensorFlow for the machine learning and GPyOpt for the Bayesian optimization. The system contains a configurable test suite, where one can enable or disable certain features. The system runs each optimization strategy sequentially. After each strategy, the delimiting layer is optimized for the best-trained model. We explain how Bayesian optimization is implemented in our system. We decided to use standard Bayesian optimization with the default parameters supplied by the GPyOpt library. For each optimization strategy trains a classification block and fine-tune the best classification block, but they differ in their approach. The shared hyperparameters optimization shares the gradient descent optimization function and learning rate between the two training steps, while the separate hyperparameters optimization does not. The separate optimizations strategy, however, splits the two training steps into two separate optimizations and uses the best classification block model from the optimization for the fine-tuning. At the end of each optimization, a TensorBoard file is completed for analysis, and the Bayesian optimization writes convergence plots and an acquisition and surrogate model plot for those optimizations that have a dimension of two or one. Additionally, we get an overview of the hyperparameters that were chosen for each iteration.

We introduce nonconvergence filtering. Nonconvergence filtering is a technique for stopping training runs that fail to produce metrics past a given threshold in a given number of epochs. It can be used by researchers that are testing out different models or hyperparameters, but it is particularly effective for automatic hyperparameter optimization. Nonconvergence filtering saves us time because we do not have to run training runs that are hopeless at achieving results that can compete with the desired results. In our results, we have found that we save considerable amounts of time by using the nonconvergence filtering technique.

Chapter 4

Experiments

In the previous chapter, we described our methodology. We presented our system for testing, and we presented and discussed parts such as metrics, hyperparameters, and libraries. In this chapter, we will describe the experiments in detail and present the results of each experiment. We will start with description and discussion around the experiments and their design. Afterward, we will present the results of the experiment from each of the datasets. Finally, for each dataset, we present the best model and its performance metrics.

4.1 Design of Experiments

We want our experiments to show how hyperparameter optimization for the hyperparameters we chose can improve the performance of our way of doing transfer learning for CNN models. We want to produce results comparable to both choosing random hyperparameters and manually chosen hyperparameters. We ran an experiment for both the Kvasir and Nerthus datasets. Both experiments were ran through our proposed system presented in section 3.4 in the methodology chapter. We ran the experiments a server with the hardware specifications listed in table 4.1. We used Ubuntu 16.04 LTS for the server and used the software libraries

Table 4.1: Hardware specifications for our testing environment.

Device	**Type**
CPU1	Intel Xeon E5-2630 v3 2.40GHz
CPU2	Intel Xeon E5-2630 v3 2.40GHz
GPU1	Nvidia Tesla K40c
GPU2	Nvidia GeForce GTX TITAN X
GPU3	Nvidia GeForce GTX TITAN X
RAM	62 GiB System memory

listed in table 3.4. The experiments are designed as follows:

1. We split the given dataset into 70% training data and 30% validation data. For a Kvasir class of 1000 images, 700 images are randomly put as training data, and 300 images are randomly put as validation data.

2. Bayesian optimization from the GPyOpt library with default parameters was used for each optimization step. An optimization step is, therefore, a Bayesian optimization run where the goal is to minimize the validation accuracy based on a set of hyperparameters. After an optimization step, we have the best model and the best hyperparameters from that step.

3. We replace the classification block and train it on the training set, a procedure we call *classification block training*. Next, we train all layers after a given default layer of 2/3rd of the current CNN model's number of layers on the training set, a procedure we call *fine-tuning*. This procedure is the gist of the hyperparameter optimization for the three hyperparameters; pre-trained CNN model, model optimization function, and learning rate. The optimization strategy decides how hyperparameters are distributed and whether to divide the two training steps into two optimization steps or keep it at one step.

4. Each iteration of Bayesian optimization selects new hyperparameter values from a given domain. We call the search space the *hyperparameter bound* as the domain contains an upper and lower bound for continuous values and also for some discrete values such as the delimiting layer. The hyperparameter bound for the four hyperparameters are as follows:

    (a) **Pre-trained model:** An index into a list of models. The list is in the following order: Xception, VGG16, VGG19, ResNet50, InceptionV3, InceptionResNetV2, DenseNet121, DenseNet169, DenseNet201.

    (b) **Gradient decent optimization function:** An index into a list of optimization functions. The list is in the following order: Nadam, SGD, RMSprop, Adagrad, Adam, Adamax, Adadelta.

    (c) **Learning rate:** A continuous value between 1 and 10^{-4}. The values were chosen because 1 is the highest default value and 10^{-4} is the lowest default value of the optimization functions.

    (d) **Delimiting later:** A discrete value between 0 and number of layers of the pre-trained model that is used. The drawback to this is that the bound is dependent on the pre-trained model being chosen before optimizing the hyperparameter.

5. For each dataset, we run three optimization strategies: The shared hyperparameters, the separate hyperparameters, and the separate optimizations optimization strategy.

   (a) **The shared hyperparameters optimization strategy.** In this strategy, we use the same hyperparameters for both the classification block training and the fine-tuning. This results in the number of hyperparameters in the hyperparameters set being three: The pre-trained model, the gradient descent optimization function, and the learning rate. The optimization function and learning rate are used for both the classification block training and the fine-tuning. The hyperparameters and the hyperparameter bounds are described in section 3.3 in the methodology chapter. The dimensionality for the Bayesian optimization iteration is three.
   (b) **The separate hyperparameters optimization strategy.** The separate hyperparameters strategy is equal to the shared hyperparameters strategy but differs in the size of the hyperparameter domain and the distribution of the hyperparameters. Instead of using a hyperparameter set of three hyperparameters, the shared hyperparameters strategy uses five: The pre-trained model, the gradient descent optimization function and learning rate for the classification block training, and the gradient descent optimization function and learning rate for the fine-tuning. The point is to find whether we can improve the optimization by having different hyperparameters for the two training steps. As such, the dimensionality becomes five.
   (c) **The separate optimizations optimization strategy.** For this strategy, we split the classification block training and fine-tuning into two separate optimizations. The idea is to decrease the dimensionality in the search space and still have different hyperparameters for the two training steps. The classification block training has a dimensionality of three with the following hyperparameters: The pre-trained model, the gradient descent optimization function, and the learning rate. The best model is then used for the fine-tuning optimization, which has a dimensionality of two with the following hyperparameters: The gradient descent optimization function and the learning rate.

6. After a strategy has finished, we fine-tune again, but this time we use the best hyperparameters found by the optimization strategy and optimize the delimiting layer alone.

7. Data is produced for each training run in each optimization. We use TensorBoard to visualize this data into graphs. With TensorBoard we can plot the validation accuracy for both each epoch and time stamp. Both types of plot are used for analysis of the optimizations in the next sections.

8. Data is also produced for each optimization, offering details such as hyperparameters for each iteration, convergence plots, and acquisition and surrogate model plot.

9. Lastly, metrics are calculated for the best model based on the validation dataset.

4.2 Results and Discussion

In the following sections, we will present our results. We are dividing the result into a section for each dataset. The Kvasir results will be examined in detail, while only the important Nerthus results will be presented. The Kvasir results are divided into subsections for each optimization strategy. We divide each optimization strategy into sections for each optimization step. At the end of both the Kvasir results and the Nerthus results we discuss the best model from that full optimization run and compare their metrics to baseline metrics released with each dataset.

4.3 Results for Kvasir

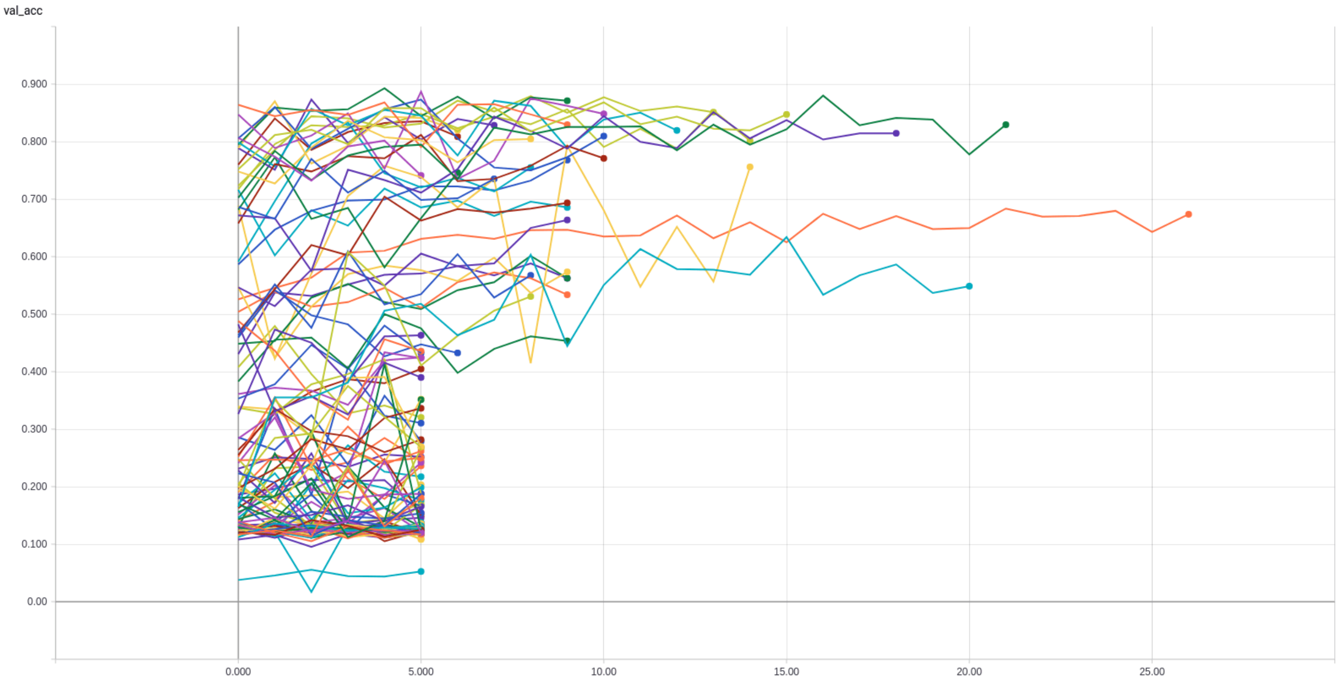


Figure 4.1: Plot of full test on the Kvasir dataset in epochs. X-axis is the number of epochs the training run has lasted, and Y-axis is the attained validation accuracy. Each line represents a training run. One Bayesian optimization iteration is, therefore, segmented into several lines. The training of the classification block produces one line, while the training of the fine-tuning produces another line. The optimization of the delimiting layer will also produce a line.

In figure 4.1, we see a plot of all training runs in a full experiment based on the Kvasir dataset. Each line represents a training run. They

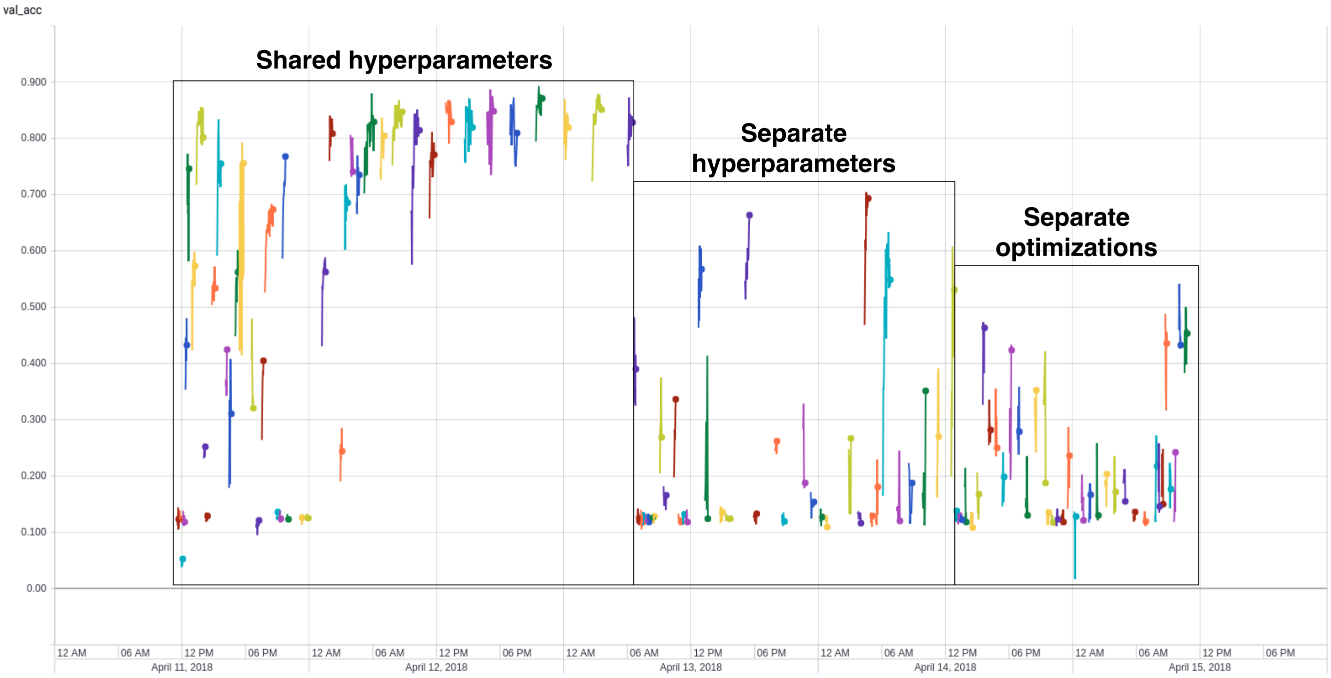


Figure 4.2: Plot of full test on the Kvasir dataset in time. X-axis is timestamps for each finished epoch. Y-axis is the attained validation accuracy. Each line represents a training run. The plot is from the same full optimization run that figure 4.1 shows. We have added boxes with captions to show where each optimization strategy is plotted.

do not represent a whole optimization run. A plot line could be from a classification block training, fine-tuning or layer optimization. Each iteration of such a training optimization has their own line. Figure 4.1 shows validation accuracy for each training run for each epoch, while figure 4.2 shows validation accuracy for each training run for time steps. In figure 4.1, we see a cluster of lines. However, we can draw some information from this plot.

1. The nonconvergence filtering is working as intended by stopping every training that fails to pass the threshold in five epochs. We can see the filtering in the plot by looking at all the lines that end after five epochs. From the plot, we can see that this is a majority of the runs. It is safe to conclude that we save a significant amount of time by doing this. Nevertheless, there are very few of the runs that are stopped early that are growing, so we could decrease the number of epochs before stopping. For the threshold, it is hard to say whether it should be adjusted or not. Finding out if the threshold should be adjusted would require more targeted tests, so we leave this for future work.

2. There are several high yielding training runs. Most produce results between 0.80 to 0.85, and some hit as high as 0.89. We can see these at the top of the plot. From this, we can see that transfer learning can reach excellent results. We will compare the results to the baseline in section 4.3.4. The highest producing training runs are from the shared hyperparameter optimization, which we will show later.

3. Some of the training runs stand out as they train for very long. Examples from the plot are the green line, hovering around 0.8 validation accuracy, ending on step 21, the orange line, almost reaching 0.7 validaiton accuracy, ending on 26 epochs, and the turquoise line, ending with 0.55 validation accuracy after 20 epochs. We use early stopping to avoid overfitting together with fifty epochs for fine-tuning. From the plot, it is clear that every training run is early stopped as none reach fifty epochs. The training runs that reach high numbers of epochs are able to do that as they manage to increase their top achieved validation accuracy before five epochs have occurred. We configured the early stopping callback to work like that. Even though the orange training run does barely grow and stalls the whole optimization, the green's top training run reaches its peak result late. We accept this compromise, so there is no reason to change the early stopping configuration by looking at this optimization run.

4. Many training runs fluctuate, nearly oscillate, significantly. One of the more prominent examples is the yellow line fluctuating from a validation accuracy of 0.74 in epoch 7 to 0.41 in epoch 8 back to a validation accuracy of 0.79 in epoch 9. There are examples of lesser fluctuation in the plot like the turquoise line mentioned in the point above as well. We can attribute this to high learning rates. These are

runs where the optimizer tries a too high learning rate for the gradient descent optimizer. We expect this as there is a random distribution at the beginning of the Bayesian optimization, and the optimization will have problems as there will be many local extrema since the relationship between the learning rate and the optimizer depends on the combination. For some optimizers a learning rate of 0.1 is good, but for others, that learning rate is really bad.

Figure 4.2 provides another perspective on the same optimization run shown in figure 4.1. In figure 4.2, we see the training runs for time steps. From this plot each optimization strategy is visible, and we have added boxes to mark the location of each within the plot. From the start of the run to April 12th 06 AM, shared optimization is visible. The other optimization strategies are more difficult to separate. The separate hyperparameters optimization strategy is from April 12th 06 AM to April 14th, and the separate optimizations optimization strategy is from April 14th 13 PM to April 15th 11 AM. We can also gain more information from this plot:

1. The shared hyperparameters optimization yields much better results than the other optimization strategies. We can see that many of the line, particularly the last lines reach between 0.85 to 0.89 near the end of the shared hyperparameters strategy. The separate hyperparameters optimization yields better results than the separate optimizations optimization, but they are much closer and produces worse results than the shared hyperparameter optimization. Additionally, the highest results in the shared hyperparameters and shared optimizations strategies are inconsistent, meaning they fluctuate and seem more like outliers.

2. The shared hyperparameter optimization is faster than the other strategies in converging. Both of the other strategies are converging at the end of their optimization runs, so this could indicate that these strategies need more iterations to be able to converge to the same values as the shared hyperparameter optimization. We can see that the shared hyperparameters strategy has converged a little before halfway into the optimization run, while the other two strategies produce spiky results and show signs of converging at the end of their optimization runs.

3. The whole run takes almost four days, but what stands out is that each optimization strategy takes different amounts of time. The shared hyperparameters strategy takes 1 day and 19 hours, the separate hyperparameters strategy takes 1 day and 6 hours, and the separate optimizations strategy takes 22 hours. We have no explanation for this difference. The reason could be resource related, but we do not know and finding it out as outside of the scope of the thesis.

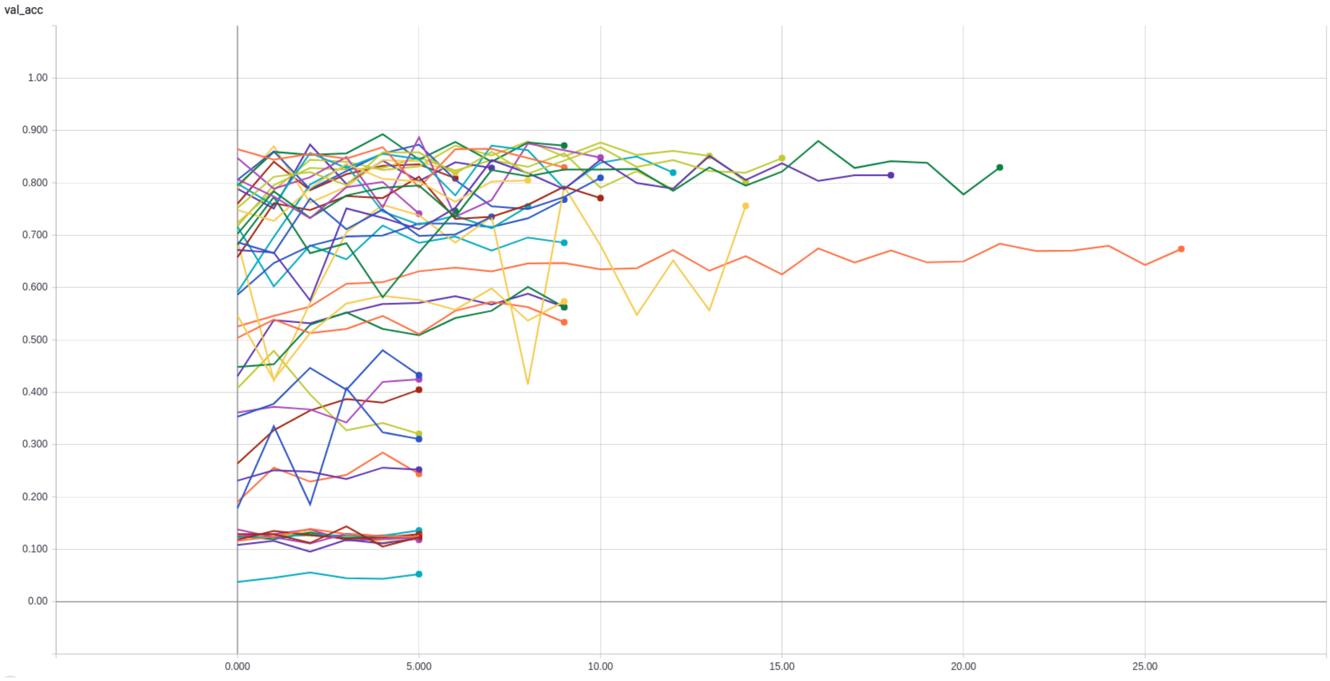


Figure 4.3: Plot of Shared hyperparameters optimization strategy run in steps on the Kvasir dataset. X-axis is the number of epochs the training run has lasted and Y-axis is the attained validation accuracy. Each line represents a training run. One Bayesian optimization iteration is, therefore, segmented into several lines. The training of the classification block produces one line, while the training of the fine-tuning produces another line. The optimization of the delimiting layer will also produce a line. The plot is from the same run as figure 4.1, but with filtering out every line that is not from the shared hyperparameter optimization strategy.

4.3.1 Shared Hyperparameters Optimization Strategy

The shared hyperparameters optimization strategy was shown in figure 4.2 to produce the best results of all the strategies. In this section, we will present the results of the shared hyperparameter optimization strategy. Figure 4.3 and 4.4 are plots from the same run as figure 4.2 and 4.1, but they are of the shared hyperparameter strategy results only. In figure 4.3, we see many of the same features as in figure 4.1:

1. We see some lines from training runs that did not pass the threshold of 0.5 in five epochs, a rule enforced by the nonconvergence filtering, and are subsequently stopped early.
2. We see the high yielding lines attaining validation accuracies above 0.80. Most of them converged before ten epochs, but there are several lines, such as the green line, ending at 21 epochs, and the purple line, ending at 18 epochs, that take time to converge.
3. We see fluctuating lines. The yellow line, ending on epoch 14 and

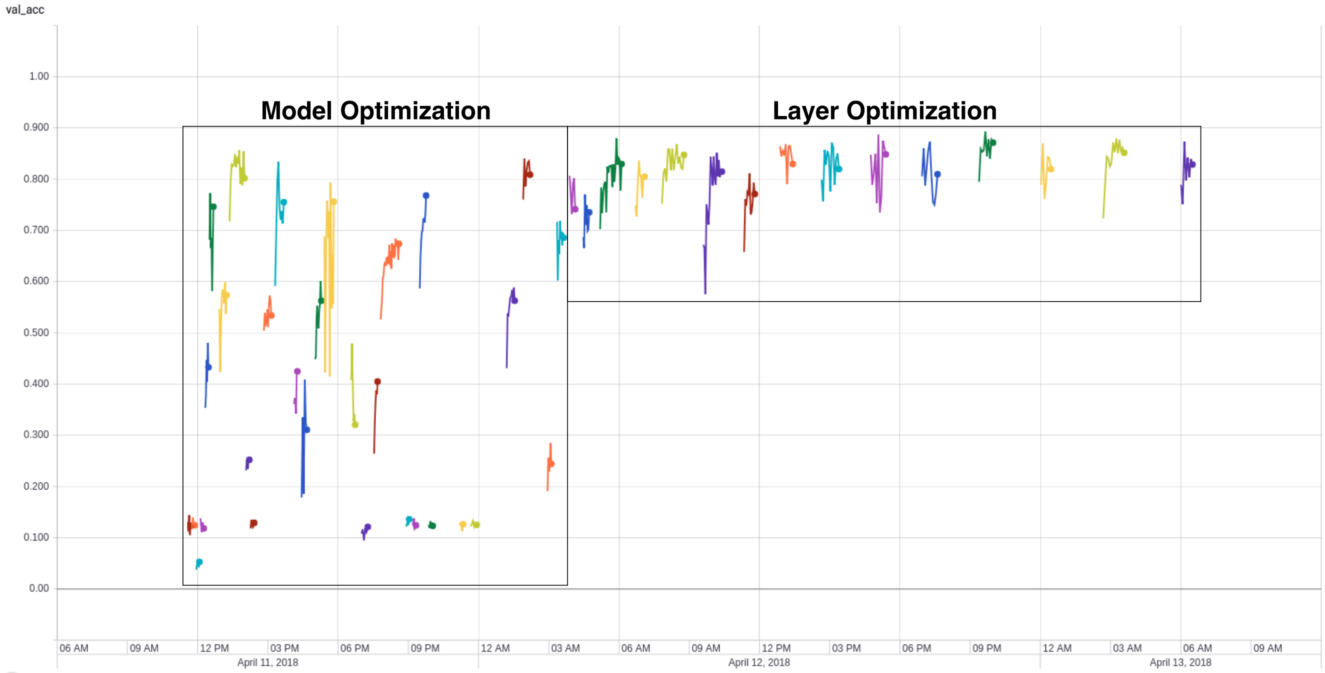


Figure 4.4: Plot of Shared hyperparameters optimization strategy run in time on the Kvasir dataset. X-axis is timestamps for each finished epoch. Y-axis is the attained validation accuracy. Each line represents a training run. The plot is a subplot from figure 4.2 of the marked box of shared hyperparameters optimization strategy. We have added boxes with captions to show where each optimization step is plotted.

exemplified in section 4.3, shows heavy fluctuations, but there are also other lines present, such as the blue line, fluctuating from 0.18 in epoch 0, up to 0.34 in epoch 1, down to 0.19 in epoch 2, and up to 0.41 in epoch 3.

4. There are less training runs that are stopped early due to nonconvergence filtering than there are training runs that run until early stopping stops them. This contrasts the findings in figure 4.1 which shows a heavy majority of training runs filtered by the nonconvergence filtering. We can draw from this that the shared hyperparameter optimization stands for a significant amount of the high yielding lines present in figure 4.1. Additionally, shared hyperparameter optimization needs fewer iterations before converging.

In figure 4.4, we see the shared hyperparameter optimization strategy marked in figure 4.1. We can see from figure 4.4 that we have marked the model optimization step, where we train the classification block and do the fine-tuning with a default delimiting layer, and marked the layer optimization, where we fine-tune with only the delimiting layer as a hyperparameter. In figure 4.4 we can deduce the following:

1. Both the model optimization step and the layer optimization step

reaches high validation accuracies. From the two, layer optimization reaches the highest validation accuracy of 0.89. This run can be seen as the green line on April 12th at 09 PM. This run shows us that optimizing the layers beyond the default value of 2/3 of the number of layers in the model works and in this case increased the result. The light green line April 11th at 02 PM achieved the highest validation accuracy of the model optimization. The run reached a validation accuracy of 0.86, which is close to the layer optimization run of 0.89. The small increase in validation accuracy indicates that layer optimization does indeed help, but only with small gains. However, these relatively small gains are important as we want to be as close to a hundred percent accuracy on classification as possible. For the medical field, a few percents could be the difference of diagnosing a disease or not.

2. The model optimization has very spiky results, meaning some training runs are really bad with validation accuracies close to 0.1, while others reach over 0.5 validation accuracy. The layer optimization, which uses the best model for training and the best hyperparameters from the fine-tuning, is very consistent and show little difference between each run. The lack of difference between the runs could mean that the delimiting layer does not matter too much in our use case to make a big impact on the validation accuracy outcome. We will look into this in more detail in section 4.3.1.

Model Optimization

In this section, we will take a closer look into model optimization for shared hyperparameter optimization by looking at figure 4.5 and 4.6. Figure 4.4 showed us that model optimization reach high validation accuracies, but also have significant differences between all of the runs. By examining figure 4.5, we can see the same features as discussed in section 4.3.1 about figure 4.3, such as runs being filtered by the nonconvergence filtering, fluctuations in validation accuracy for many of the runs, some runs achieving high validation accuracies and a few runs running for many more epochs than the majority. However, we can see in figure 4.5 that some of the runs are the examples we pointed out in the previous sections:

1. The yellow line that fluctuates and ends on epoch 14 comes from the model optimization run. The training run is a fine-tuning run and uses a learning rate of 0.8 with Adadelta as optimizer and InceptionV3 as the pre-trained CNN model. The default learning rate for Adadelta is 1.0, which should indicate that the fluctuations are indeed not from a high learning rate. However, during the fine-tuning step, it would, from intuition, be advantageous to have a smaller learning rate as we do not want to change the pre-trained weights too much since that might cause them to lose their previous knowledge instead of building on top of it. There is not way we can say for certain the reason for the fluctuation from these plots. We can

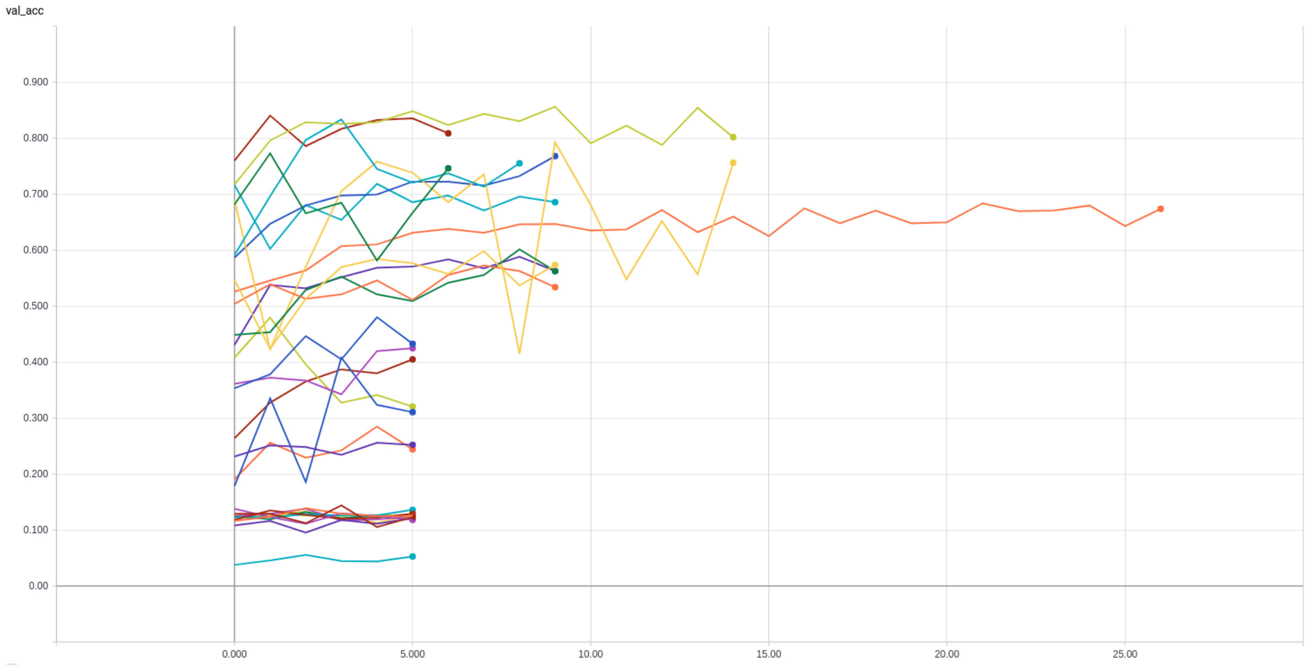


Figure 4.5: The shared hyperparameters optimization strategy's model optimization run for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

only speculate that it must be the combination of model, optimizer and transfer learning that do not work well with such a high learning rate.

2. The orange line that lasts until epoch 26 comes from the model optimization run. This run contrasts the yellow line run in that it barely fluctuates. However, the run takes a long time to converge. The training run is a fine-tuning run and uses a learning rate of 0.014 with Adadelta and Xception as the pre-trained CNN model. In this run, the learning rate is much lower, which explains the slow convergence and long run.

3. The light green line yielding the highest validation accuracy and ends on epoch 14 comes from the model optimization. This run is also a fine-tuning run, and uses a learning rate of 0.84 with Adadelta as optimizer and InceptionReNetV2 as the pre-trained CNN model. We can see from the results and the plot that the relationship between the learning rate and the gradient descent optimizer makes a difference for the fluctuations. However, it seems that the most important aspect is the pre-trained CNN model. For the yellow line, using Adadelta with a learning rate of 0.8 made it flucutate heavly, but for the light green line, the roughly same learning rate made it yield the best validation accuracy. We can conclude from this that the learning

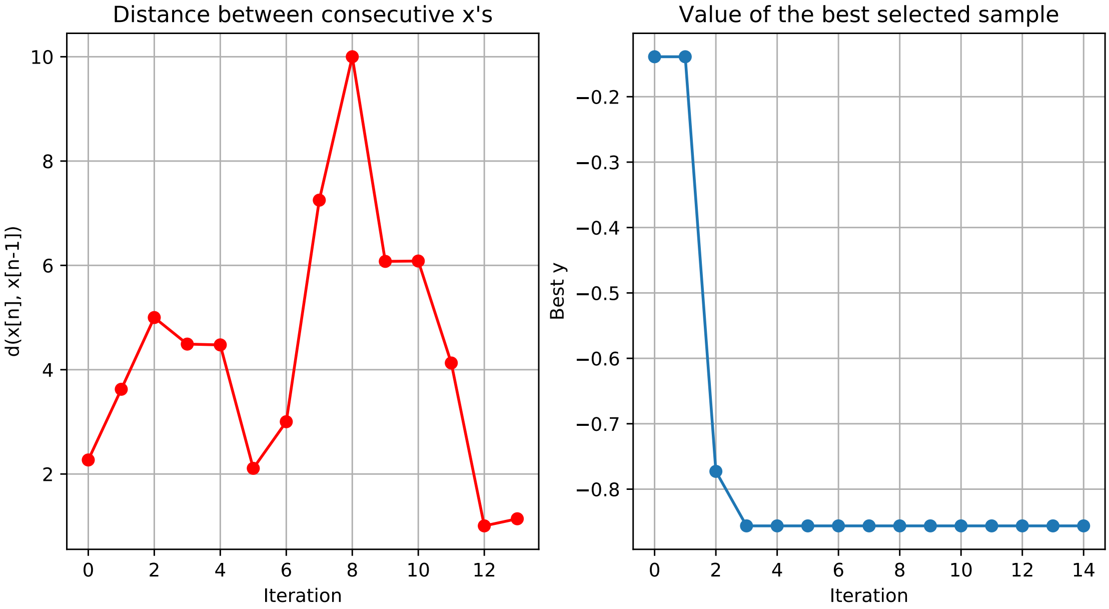


Figure 4.6: Convergence plot of the shared hyperparameters optimization strategy's model optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

rate is different for each model and gradient descent combination, and that using the default learning rate does not work for every pre-trained CNN model.

In this section, we will take a closer look at model optimization for shared hyperparameter optimization by looking at figure 4.5 and 4.6. Figure 4.4 showed us that model optimization reach high validation accuracies, but also have significant differences between all of the runs. By examining figure 4.5, we can see the same features as discussed in section 4.3.1 about figure 4.3, such as runs being filtered by the nonconvergence filtering, fluctuations in validation accuracy for many of the runs, some runs achieving high validation accuracies and a few runs running for many more epochs than the majority. However, we can see in figure 4.5 that some of the runs are the examples we pointed out in the previous sections:

1. The yellow line that fluctuates and ends on epoch 14 comes from the model optimization run. The training run is a fine-tuning run and uses a learning rate of 0.8 with Adadelta as optimizer and

InceptionV3 as the pre-trained CNN model. The default learning rate for Adadelta is 1.0, which should indicate that the fluctuations are indeed not from a high learning rate. However, during the fine-tuning step, it would, from intuition, be advantageous to have a smaller learning rate as we do not want to change the pre-trained weights too much since that might cause them to lose their previous knowledge instead of building on top of it. There is no way we can say for certain the reason for the fluctuation from these plots. We can only speculate that it must be the combination of model, optimizer and transfer learning that does not work well with such a high learning rate.

2. The orange line that lasts until epoch 26 comes from the model optimization run. This run contrasts the yellow line run in that it barely fluctuates. However, the run takes a long time to converge. The training run is a fine-tuning run and uses a learning rate of 0.014 with Adadelta and Xception as the pre-trained CNN model. In this run, the learning rate is much lower, which explains the slow convergence and long run.
3. The light green line yielding the highest validation accuracy and ends on epoch 14 comes from the model optimization. This run is also a fine-tuning run and uses a learning rate of 0.84 with Adadelta as optimizer and InceptionReNetV2 as the pre-trained CNN model. We can see from the results and the plot that the relationship between the learning rate and the gradient descent optimizer makes a difference for the fluctuations. However, it seems that the most important aspect is the pre-trained CNN model. For the yellow line, using Adadelta with a learning rate of 0.8 made it fluctuate heavily, but for the light green line, the roughly same learning rate made it yield the best validation accuracy. We can conclude from this that the learning rate is different for each model and gradient descent combination and that using the default learning rate does not work for every pre-trained CNN model.

Figure 4.6 shows two plots. The first plot shows the distance between two consecutive x's. X is the hyperparameter values of an iteration. How the X is calculated depends on the dimensionality of the hyperparameters. For example, the first plot in figure 4.8, which is from layer optimization, shows the distance between two consecutive x's for the delimiting layer, which is a one-dimensional hyperparameter set since because we only optimize the delimiting layer. For one-dimensional hyperparameter sets, the distance between two consecutive x's is the difference between the two numbers. If the Bayesian optimization in iteration 5 try a delimiting layer of 178 and in iteration 6 try a delimiting layer of 478, the distance between the two consecutive x's for that iteration is $300 = 478 - 178$. The first plot in figure 4.6 is from model optimization, which has three hyperparameters for the shared hyperparameter optimization strategy. Three hyperparameters make the dimensionality three-dimensional, which means we must use a different technique for calculating the distance. We use Euclidean distance [8] for the calculation of the distance between each of the hyperparameter sets the Bayesian optimization tries. Euclidean distance can calculate the straight-line distance between points in any dimension and is defined by:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2} \tag{4.1}$$

where **p** and **q** are two points in n-dimensional space.

The first plot in figure 4.6 shows the Euclidean distance between consecutive hyperparameter set iterations for each iteration. As an index represents each model and optimizer in a list, the combination creates a point in three-dimensional space. By plotting the distance between these points, we can get a sense of how much difference it is between the hyperparameters of each iteration. However, the drawback is that there will be a bigger distance between for example a pre-trained model with index 1 and another pre-trained model with index 7 than a pre-trained model with index 1 and a pre-trained model with index 2. Changing between models should not be weighted, but they are when calculating distance as the index is included in the calculation. Nevertheless, we can see from the plot when the hyperparameters are not changing, and we can gain more insight into how the Bayesian optimization evaluates the hyperparameters. GPyOpt generates the plot, so because of the previously mentioned advantages, we choose to show it.

We can see in the first plot in figure 4.6 that there are some big changes in the tested hyperparameters happening between iteration nine, and ten. The iteration of the plot is zero-indexed, so the distance between iteration nine and ten is plotted in iteration 8 in the graph. In iteration nine, DenseNet201 is used with Nadam and learning rate of 0.05, and in iteration ten, Xception is used with Adadelta with a learning rate of 0.01. The point at iteration 8 in the graph is the highest point in the graph, but it does not reflect reality. The reason for the large distance is because numeric values that represent indices represent the pre-trained model and the gradient descent optimizer. The problem is that using numeric values implies a numeric relationship between each model or optimizer. For example, a model with index 1 will be closer to a model with index 2 than a model with index 8. In reality, there is no relationship between the models, but the Bayesian optimization mistakenly assumes there is.

The second plot in figure 4.6 shows us the best result of all iterations for each iteration. The Y-axis is the validation accuracy negated. There is no reason for us to use the negated number, but that is how GPyOpt calculated the plot. We can see from this graph that the training converge to the maximum validation accuracy after only four Bayesian optimization iterations. There is a huge difference from the iteration marked 1, and the iteration marked 2 in the plot. It indicates that the optimization does not converge properly, but is lucky to try combinations that work well early and fail to improve on this. This observation reflects the same observations made in section 4.3.1 about figure 4.4, where we see that training runs in the model optimization seemingly are all over the graph. This variance of attained validation accuracies is also reflected in table 4.2.

Iteration	Val Acc	Model	Optimizer	Learning Rate
1	0.1389	InceptionV3	RMSprop	0.8463
2	0.1379	ResNet50	Adam	0.4595
3	0.7728	Xception	Adadelta	0.8310
4	0.8562	InceptionResNetV2	Adadelta	0.8417
5	0.1349	VGG16	Adam	0.4336
6	0.8333	InceptionResNetV2	Adadelta	0.2626
7	0.4077	DenseNet169	Adadelta	0.9307
8	0.7927	InceptionV3	Adadelta	0.8033
9	0.1210	DenseNet201	Nadam	0.0489
10	0.6835	Xception	Adadelta	0.0144
11	0.1379	Xception	Nadam	0.9764
12	0.1319	VGG16	Adadelta	0.8948
13	0.1329	InceptionResNetV2	Adamax	0.6861
14	0.8403	InceptionResNetV2	Adadelta	0.5789
15	0.7183	InceptionV3	Adadelta	0.0278

Table 4.2: Validation accuracy and hyperparameters for each Bayesian optimization iteration in the shared hyperparameters optimization strategy's model optimization.

All of the Bayesian optimization iterations and their attained validation accuracies and used hyperparameters for the model optimization run are listed in 4.2. The first plot in figure 4.6 should be evaluated in the context of table 4.2, and we can see from the plot that there are many different combinations of hyperparameters that are tested and that the Bayesian optimization does not converge to a particular set of hyperparameters. Nevertheless, the graph does not reflect the true performance of the Bayesian optimization but shows us that using indices for representing models and optimizers might present a problem for the convergence of the Bayesian optimization algorithm as it detects relationships that are not present in reality.

Layer Optimization

The layer optimization is different from the model optimization in that it only has one hyperparameter, the delimiting layer. It uses the best hyperparameters from the model optimization and the best CNN model for training, then repeats the fine-tuning while only changing the delimiting layer between iterations. Using only one hyperparameter makes the

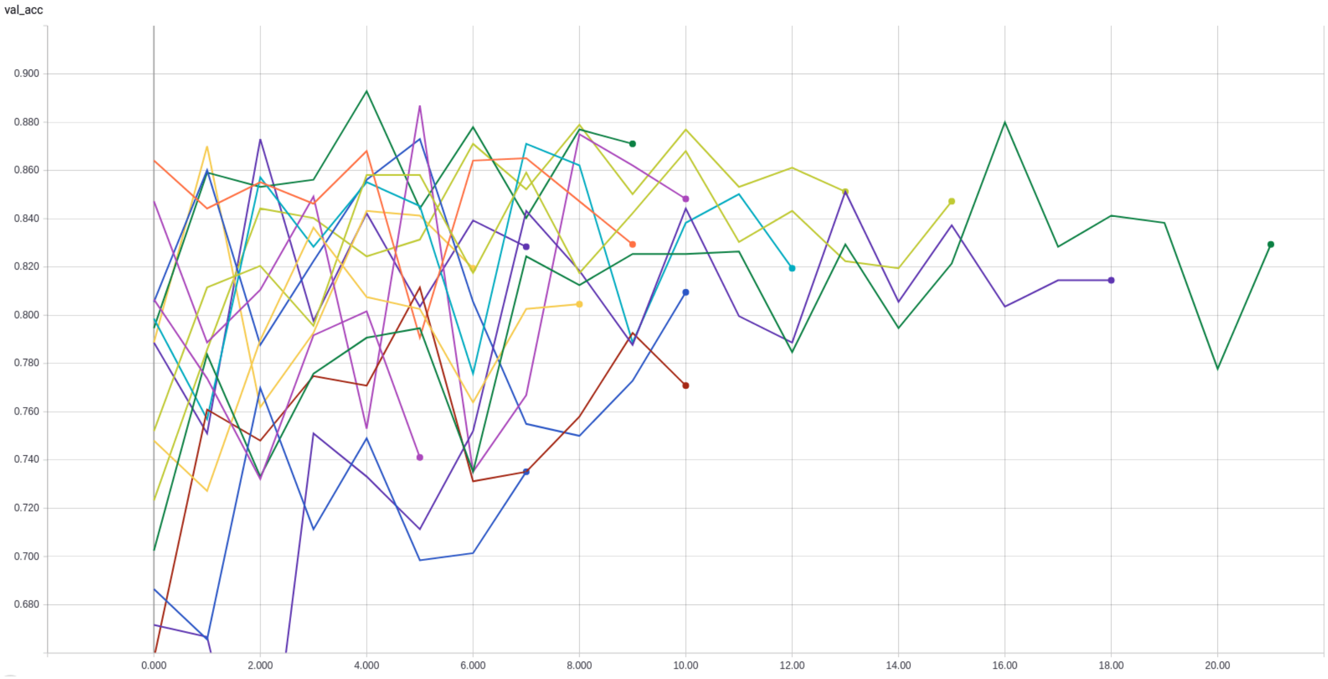


Figure 4.7: The shared hyperparameters optimization strategy's layer optimization run for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning are plotted together.

search space one-dimensional. One-dimensional search space combined with a relatively small search space of the number of layers in the model, makes the optimization perform better. Additionally, the layer numbers are numeric values with normal numeric relationships between them. Numeric values in comparison to representative values remove the complexity the model optimization presents to the Bayesian optimization, making the run converge faster.

Figure 4.7 shows the shared hyperparameters optimization strategy's layer optimization. The test is the same as the test plotted in figure 4.1. We can see from figure 4.7 that all the training runs reach validation accuracies above 0.74. The layer optimization is based on the best model from the model optimization which is from iteration 4 in table 4.2, which is an InceptionResNetV2 model with Adadelta as optimizer and 0.8417 as learning rate. We can observe in figure 4.7 that many of the training runs start out and end with lower values of validation accuracy. An example of this is the blue line at the bottom of the graph, ending on epoch seven with a validation accuracy of right under 0.74. The blue line never reaches the same values as the model it starts out with, which means there are delimiting layers which are worse than the 2/3rd of model layers default layer we use for fine-tuning. However, we can also see several lines reaching validation accuracies above the starting point of 0.856 which is the validation accuracy reached by the model optimization. The highest value we reach is by the green line at epoch four, where it reaches 0.893 in validation accuracy. From this result, we can deduce that we have increased the validation accuracy by 3.7% by tuning the delimiting layer. The result for each Bayesian optimization iteration is listed in table 4.3.

Iteration	Val Acc	Layer
1	0.8065	345
2	0.7698	603
3	0.8800	479
4	0.8363	417
5	0.8681	178
6	0.8512	478
7	0.8115	480
8	0.8681	177
9	0.8710	179
10	0.8869	177
11	0.8730	152
12	0.8929	136
13	0.8700	92
14	0.8790	131
15	0.8730	131

Table 4.3: Validation accuracy and hyperparameters for each Bayesian optimization iteration in the shared hyperparameters optimization strategy's layer optimization.

Since the layer optimization searches in one-dimensional space and each hyperparameter has a numeric value, the first plot in figure 4.8 is more useful for visualizing the distance between each delimiting layer that the Bayesian optimization tries than the first plot in figure 4.6. We can see in the first plot in figure 4.8 that the optimization tries many different fine-tuning layers before leveling out at the seventh iteration. The large fluctuations from iteration two to seven can be explained by looking at the second plot in figure 4.8. Similarly to the second plot in figure 4.6, the plot converges quickly to a high value. However, in the second plot in figure 4.8, we see that it manages to improve the best validation accuracy after seven iterations. Bayesian optimization tries to balance exploration with exploitation, which means it tries to search in regions of search space with high uncertainty and search in regions with high estimated value. The first five iterations are randomly placed to map the search space. We see large variances in the hyperparameters tried as the Bayesian optimization tried to explore the whole search space. After the exploration, the optimization

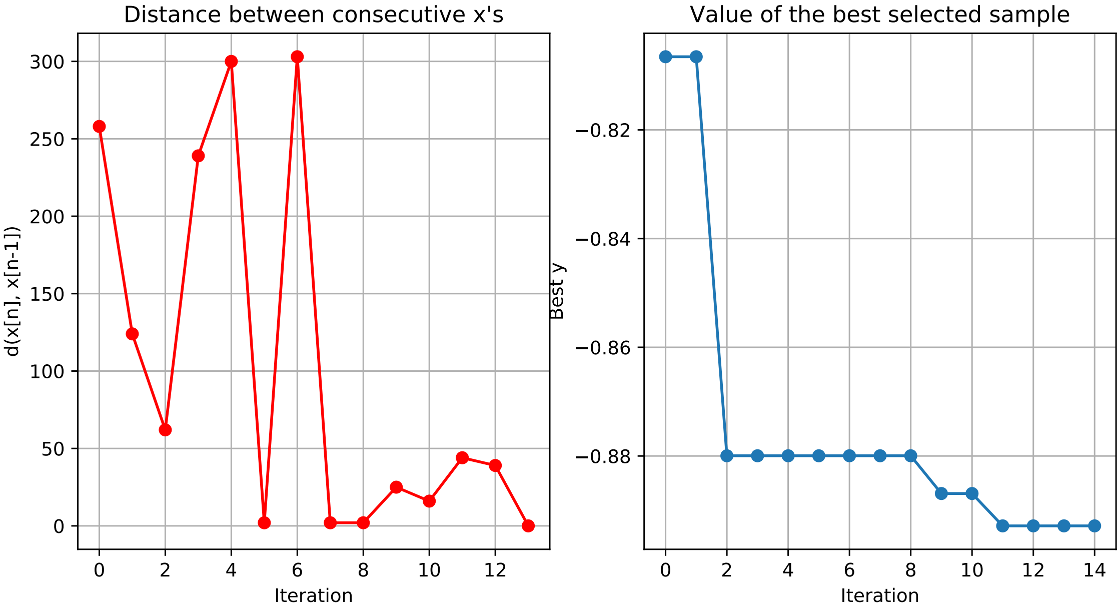


Figure 4.8: Convergence plot of the shared hyperparameters optimization strategy's layer optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

tries to exploit the numeric area it found the best validation accuracy. We can see it works in the second plot in figure 4.8 as the Bayesian optimization manages to find a better result twice after starting to exploit the area.

The acquisition function is responsible for determining where to explore or exploit next. We can only plot it for one- and two-dimensional search spaces. We use the acquisition function "Expected improvement" [101], which tries to improve the validation accuracy by calculating a tradeoff between exploration and exploitation. In figure 4.9, we see the Gaussian process surrogate model and the acquisition function, expected improvement, plotted for the layer optimization. Bayesian optimization works by creating a surrogate model, which is a Gaussian process in our case. The surrogate model is much cheaper to optimize than the real model and is used by the acquisition function to determine where in the search space to fetch the new values for the hyperparameters. The Y-axis in figure 4.9 shows the surrogate model's output for a value of a hyperparameter, which is, in this case, the delimiting layer. The X-axis shows the value of the hyperparameter. The Gaussian model is tuned over time by each result,

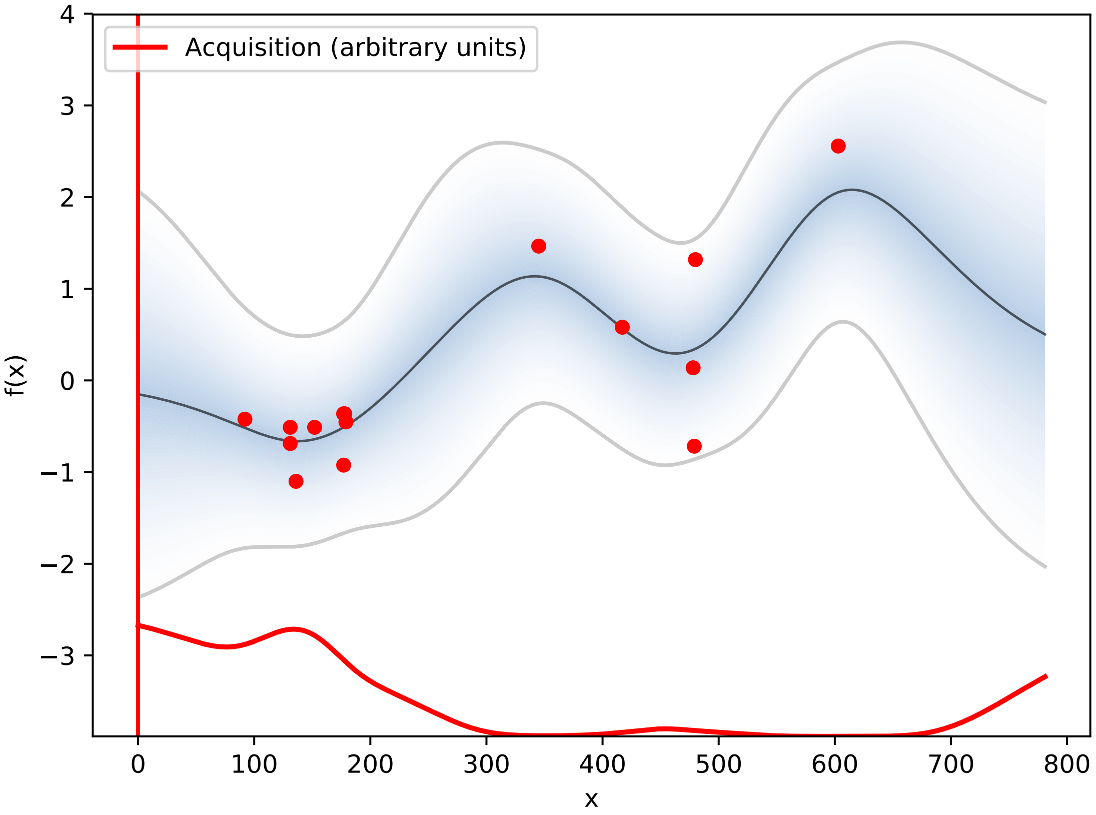


Figure 4.9: Gaussian Process surrogate model and Expected Improvement acquisition function for the shared hyperparameters optimization strategy's layer optimization. The Y-axis is the expected result of a hyperparameter in the surrogate model. X-axis is the hyperparameter, which is this case is the delimiting layer. The black line represents the posterior mean, the gray lines represent the posterior uncertainty, the red dots are tested hyperparameters and their results on the surrogate model, the red vertical line is where the acquisition function would try the next test should it run for another iteration, and, finally, the horizontal red curve is the acquisition function in arbitrary units. A lower posterior mean is better, and a higher value for the acquisition function is where the next hyperparameter values will be taken from.

and we see the surrogate model plotted after the optimization is done. The black line represents the output of the Gaussian process surrogate model for each delimiting layer value, which becomes the posterior mean as it is a prediction. The gray lines, with gradient blue between them and the black line in the middle, represents the posterior uncertainty of the surrogate model. Each red dot is a tested hyperparameter value and the results. Lower outputs for the surrogate model is better, and the lowest dot gives the best validation accuracy. The red, bottom line represents the acquisition function and is given in arbitrary units. The highs of the acquisition function show wherein the search space it wants to explore next. The red vertical line shows where the optimization would get the next value for the delimiting layer in the search space should the optimization continue for one more iteration. As we can see, the red vertical line is where the acquisition function is at its highest.

The plot of the fitted surrogate model and acquisition function in figure 4.9 is valuable to us as we can get a deeper insight into how the Bayesian optimization works. We see in the figure, that the acquisition function finds success with lower values for the delimiting layer, meaning most of the CNN model is re-trained. Additionally, we see that much of the search space is left unexplored and several attempts have been made at exploiting the area around the best performing delimiting layer. Perhaps more iterations would achieve better results. Nevertheless, we are content with the improvement as it achieved the best result of the full run of the optimization.

4.3.2 Separate Hyperparameters Optimization Strategy

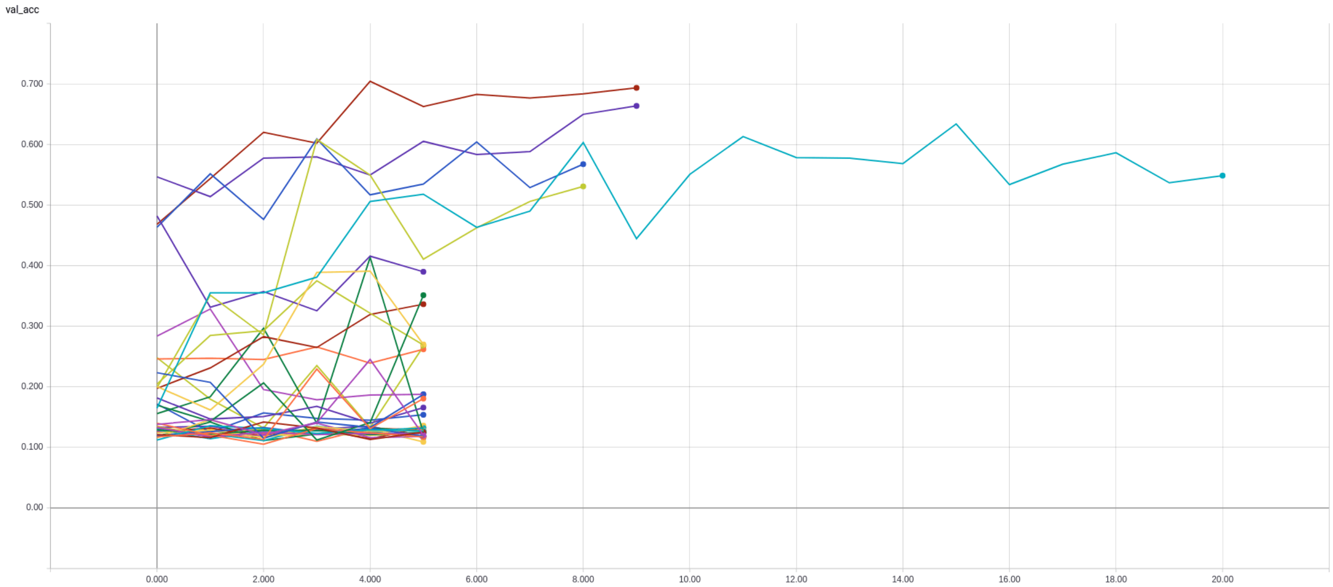


Figure 4.10: The separate hyperparameters optimization strategy's plot of training runs for each epoch.

From figure 4.2 we know that the separate hyperparameters optimization strategy yielded the worst results in regards to validation accuracy. Figure 4.10 and 4.11 shows us the same full optimization from the perspective of the separate hyperparameters optimization strategy. From figure 4.10, we can make the following observations:

1. No training run in the graph achieve the same levels of validation accuracy as the best shared hyperparameters optimization strategy training runs achieve. The highest validation accuracy achieved is by the brown line, reaching a best value of 0.70.

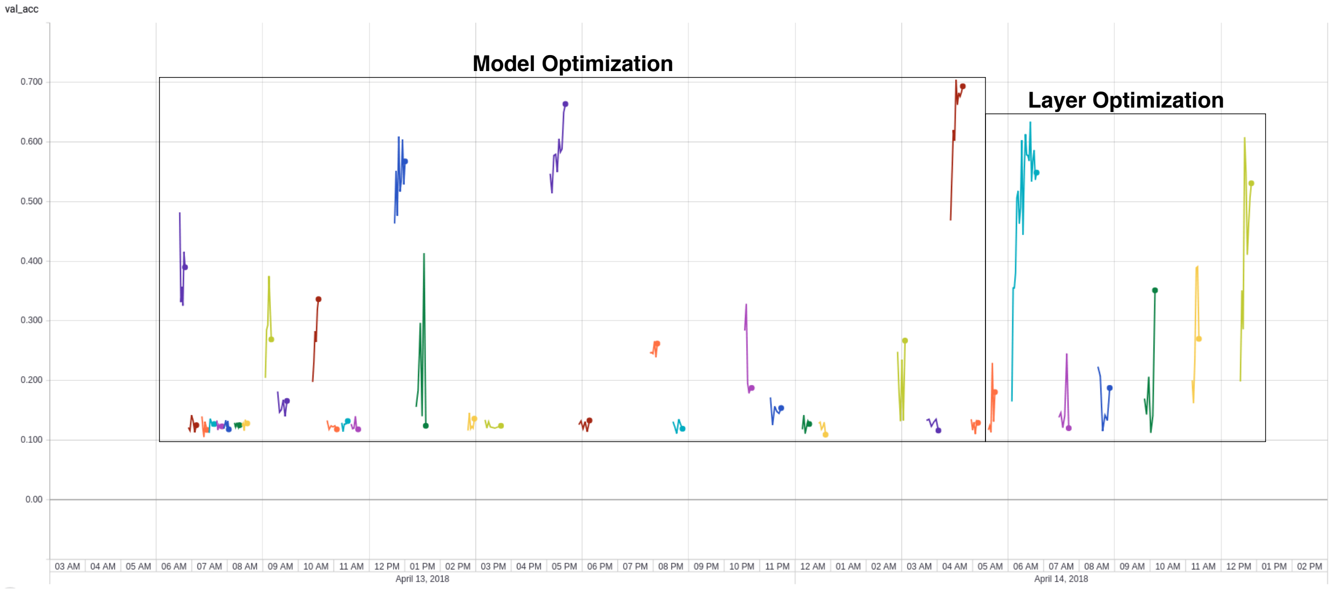


Figure 4.11: The separate hyperparameters optimization strategy's plot of training runs for time. The model optimization and layer optimization has been marked with black boxes.

2. Only five training runs pass the 0.5 validation accuracy threshold. It is unlikely that any of the runs filtered by the nonconvergence filtering would pass the threshold, even with more epochs, as they do not converge fast enough, so filtering them is the right choice.

Figure 4.11 shows us that the model optimization has some random spikes, but do not converge at all. We can also see that the layer optimization fails to exceed the model optimization. We can see that the Bayesian optimization algorithm during the layer optimization stopped after seven iterations instead of the fifteen it was allowed to do. We will examine what happened in the next sections.

Model Optimization

Figure 4.12 gives us more insight into how the separate hyperparameters optimization strategy did. We see that only three training runs pass the threshold of 0.5. The brown line achieves the best validation accuracy of 0.70. The brown line is the same highest reaching training run we discussed in section 4.3.2 from figure 4.10. The validation accuracies are subpar compared to those achieved by the shared hyperparameters optimization strategy, but there is more to the lines than what the graph shows.

By looking at table 4.4, we can see that iteration eight reach a validation accuracy of 0.41, while all other iterations reach validation accuracies below 0.19. These values are not consistent with the values we saw in figure 4.12 and 0.41 is far from the 0.70 validation accuracy we saw. The reason for this discrepancy is that the three lines above 0.5 validation accuracy we see in figure 4.12 are from the classification block training run and not

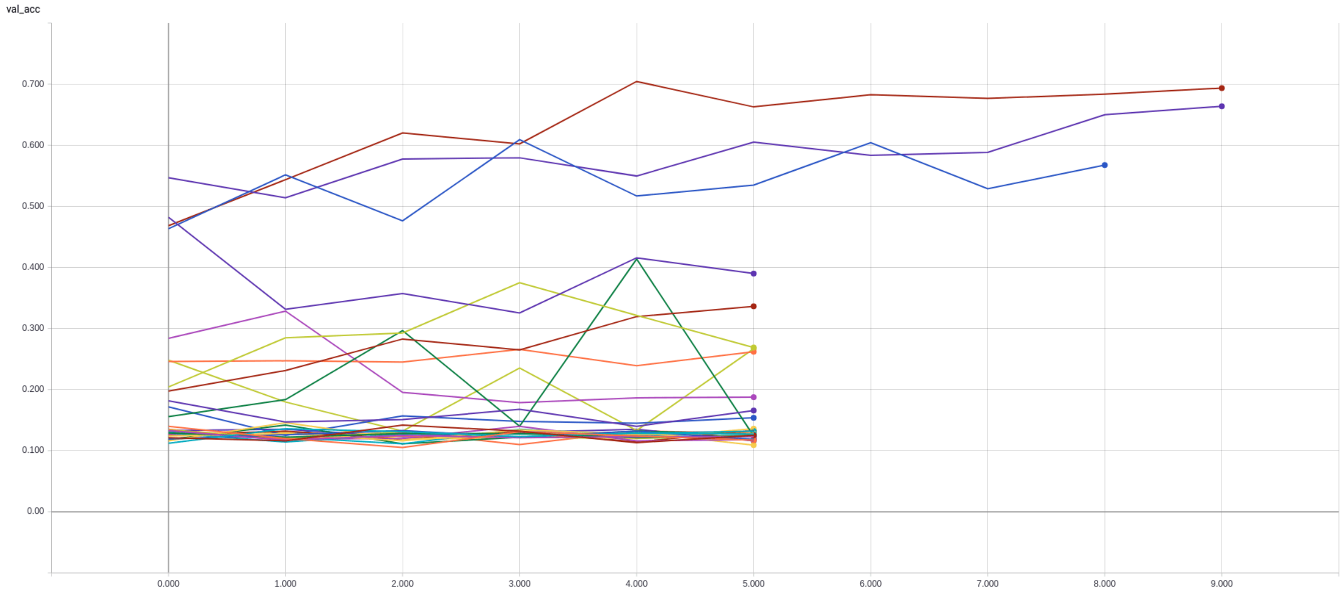


Figure 4.12: The separate hyperparameters optimization strategy's plot of model optimization training runs for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

from the fine-tuning. As such we can say the following about separate hyperparameters optimization:

1. The classification block training optimization did significantly better than the fine-tuning. The fine-tuning degraded the performance of the trained classification block it was built on. Even though some of the performance attained from the classification block runs were good, the Bayesian optimization failed to improve the performance by each iteration as can be seen in table 4.4 where only iteration eight stands out. However, even iteration eight's attained validation accuracy of 0.41 is below the threshold of 0.5. Iteration eight was trained on a classification block training with a validation accuracy of 0.61, and the best classification block training with a validation accuracy of 0.70 resulted in a fine-tuning reaching a validation accuracy of 0.13, shown as iteration fifteen in table 4.4.

2. All the fine-tuning runs failed to pass 0.5 validation accuracy and were consequently terminated by the nonconvergence filtering.

3. We can see in table 4.4 that once the Bayesian optimization found a combination that gave a better validation accuracy in iteration eight, it continued with the same model and optimizers and only tried adjusting the learning rates in the next three iterations. When changing the learning rates, the Bayesian optimization got terrible validation accuracies around 0.13. We find this interesting because it

Iteration	Val Acc	Model	Block Optimizer	Block LR	Tune Optimizer	Tune LR
1	0.1419	DenseNet169	SGD	0.6081	Adam	0.2029
2	0.1359	Xception	RMSprop	0.5402	RMSprop	0.5203
3	0.1329	VGG16	Adamax	0.8095	SGD	0.3851
4	0.1339	ResNet50	Adagrad	0.1632	RMSprop	0.6369
5	0.1815	DenseNet201	Nadam	0.4378	Adamax	0.7506
6	0.1329	DenseNet201	Nadam	0.2015	Adamax	0.7203
7	0.1399	VGG19	Adam	0.7275	RMSprop	0.2186
8	0.4137	InceptionResNetV2	SGD	0.1999	Adamax	0.0720
9	0.1339	InceptionResNetV2	SGD	0.8519	Adamax	0.6365
10	0.1329	InceptionResNetV2	SGD	0.0811	Adamax	0.6922
11	0.1349	InceptionResNetV2	SGD	0.4545	Adamax	0.3055
12	0.1716	DenseNet201	Nadam	0.5744	Adamax	0.9106
13	0.1300	Xception	Adamax	0.6329	SGD	0.3818
14	0.1349	DenseNet201	Nadam	0.7666	Adamax	0.3332
15	0.1349	VGG16	Adadelta	0.3300	Nadam	0.8331

Table 4.4: Validation accuracy and hyperparameters for each Bayesian optimization iteration in the separate hyperparameters optimization strategy's model optimization.

shows the effect of the learning rate alone on the performance of the training runs.

It is clear from the results that the separate hyperparameters optimization strategy failed. It could only train a few classification blocks above the threshold of 0.5 validation accuracy. There is a combination of reasons for the bad results:

1. High dimensionality in an already complicated search space makes it less likely for the Bayesian optimization to find a good combination in the same number of iterations compared optimizing in a lesser dimensionality. We can not say for certain that the Bayesian optimization would not have found a good combination eventually, but these are high resource training runs, so the separate hyperparameters optimization strategy does not fit for our use case.

2. Different hyperparameters between the classification block training and the fine-tuning might work with more iterations or different Bayesian optimization settings. However, since shared hyperparameters worked so much better, we have reason to suspect that changing the optimizer and the learning rate between classification block training and fine-tuning is fundamentally wrong. Nevertheless, to

give merit to such a statement would require more testing and longer runs. Additionally, related work suggests using different learning rates between the classification block training and the fine-tuning, with a slower learning rate for the fine-tuning than the classification block [40]. Our limited results do, however, not reflect that.

Our results in table 4.4 reflects the result per Bayesian optimization iteration, and as such only consider the results after the fine-tuning is finished. It would be interesting to see the performance of the model optimization in terms of validation accuracy as well. While possible to add for future tests, we have not implemented it as our focus was on achieving the best validation accuracies for each iteration. Having results for the model optimization would only benefit us in comparing the model optimizations between optimization strategies. However, it is irrelevant to the problem statement as the performance of the model optimization and the hyperparameters that the Bayesian optimization finds decide the fine-tuning. The model optimization cannot be considered alone as the success of the optimization is always evaluated by the result of the fine-tuning, which is dependent on the model optimization. As such, the results we got from the separate hyperparameters optimization strategy does not mean we need to change the model optimization, but rather change the way the Bayesian optimization selects the hyperparameters.

Layer Optimization

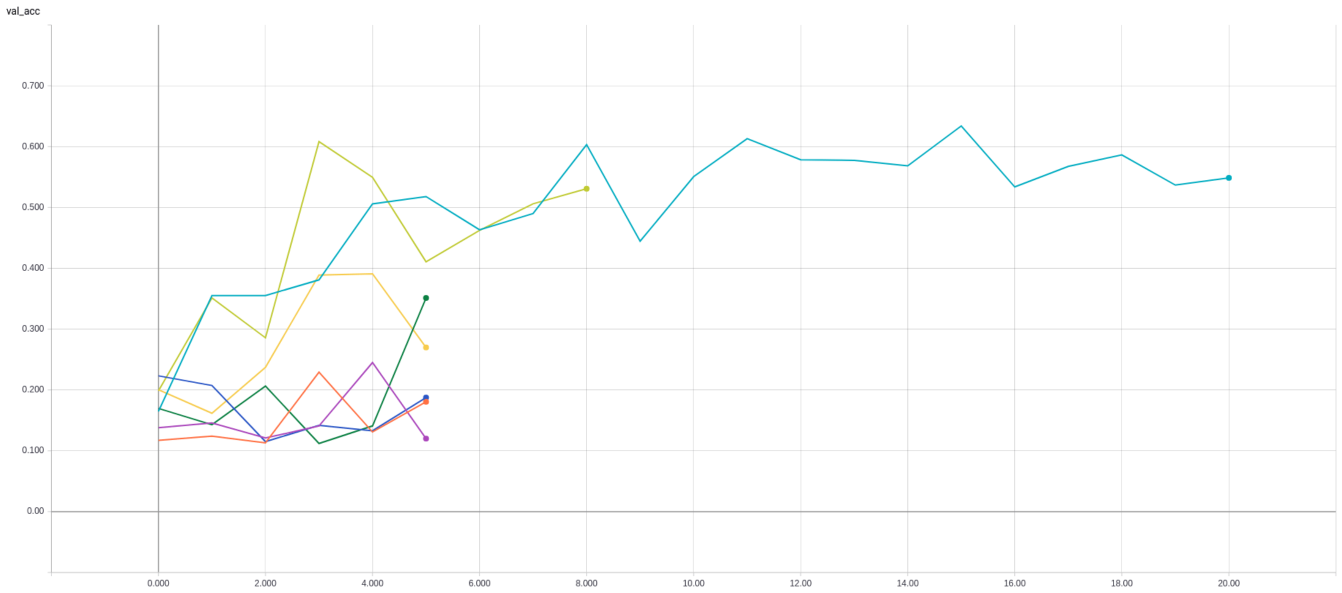


Figure 4.13: The separate hyperparameters optimization strategy's plot of layer optimization training runs for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

In the previous section, we found that we failed in finding good hyperparameters for the separate hyperparameters strategy. The best-trained model reached 0.41 validation accuracy, which is considerably lower than the best-trained model from the model optimization of the shared hyperparameters optimization strategy, reaching a validation accuracy of 0.86. It was the fine-tuning step in the model optimization that lowered the results from the classification block step. A reason for this could be that the default layer we selected produce bad results. As we chose the default layer arbitrarily, it would make sense that certain models could produce bad results. This could give unfair advantages to some models over others. For example, the default layer could work great for fine-tuning an InceptionResNetV2 model but work really bad for a VGG16 model. After all, the best runs from both the shared hyperparameters optimization and the separate hyperparameters optimization are both using the InceptionResNetV2 pre-trained model. This connection could mean that InceptionResNetV2 is the best model for the default delimiting layer. Solving this source of unfairness would require us to either include the layer in the same optimization step as the model optimization, which would increase the dimensionality or remove the pre-trained model from the hyperparameters and instead run the whole Bayesian optimization for each pre-trained model.

The best result from the shared hyperparameters optimization used InceptionResNetV2, and the layer optimization was successful in increasing the validation accuracy by adjusting the delimiting layer. The best layer tested was 136. We would expect the layer optimization to achieve the same results for the separate hyperparameter optimization strategy as they share the pre-trained model. The same results being increased validation accuracies and the best delimiting layer being around the same.

In figure 4.13, we see that the layer optimization does indeed increase the achieved validation accuracy. The light green line and the turquoise line both reach a validation accuracy above 0.6. We can say the following about the lines:

1. The light green line is early stopped because it fails to increase the validation accuracy after five epochs. We can see it reach a validation accuracy of 0.60 in epoch three but falls in the next step to 0.41. Even though it is increasing from there, it does not reach a better validation accuracy, so the early stopping callback, meant to avoid overfitting, stops the training run. However, the light green line was increasing, so stopping it might have been wrong as it could perhaps reach a higher validation accuracy after more epochs. We can only speculate, but the light green line shows us that we need to consider how we configure early stopping carefully.

2. The turquoise line manage to last for twenty epochs, which is many more than the eight epochs the light green line, which is the second longest run of the layer optimization, lasted. The turquoise line managed to increase the validation accuracy before five epochs several times and avoided early stopping until epoch twenty. Because

of this, it managed to outperform the light green line and became the highest yielding training run with a validation accuracy of 0.63.

Iteration	Val Acc	Layer
1	0.2292	608
2	0.6339	758
3	0.2450	257
4	0.2232	99
5	0.3512	241
6	0.3909	758
7	0.6081	758

Table 4.5: Validation accuracy and hyperparameters for each Bayesian optimization iteration in the separate hyperparameters optimization strategy's layer optimization.

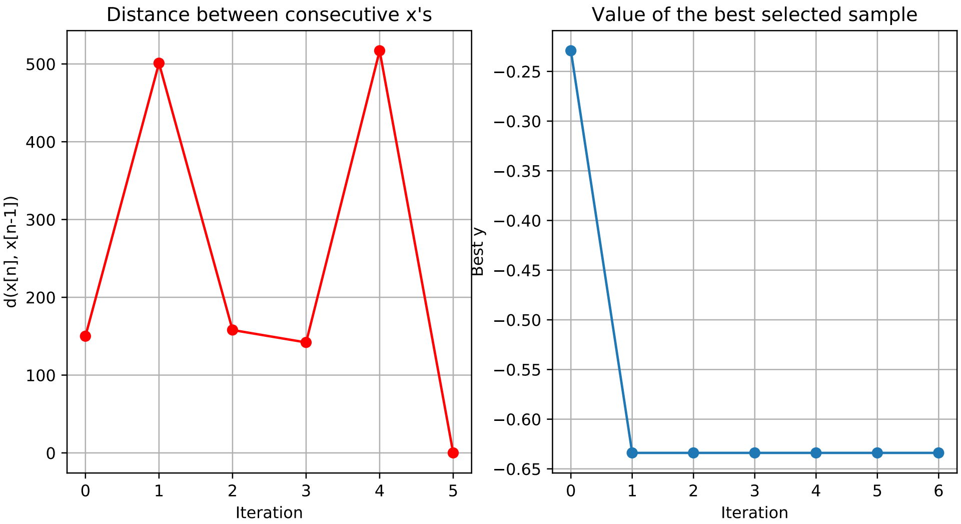


Figure 4.14: Convergence plot of the separate hyperparameters optimization strategy's layer optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

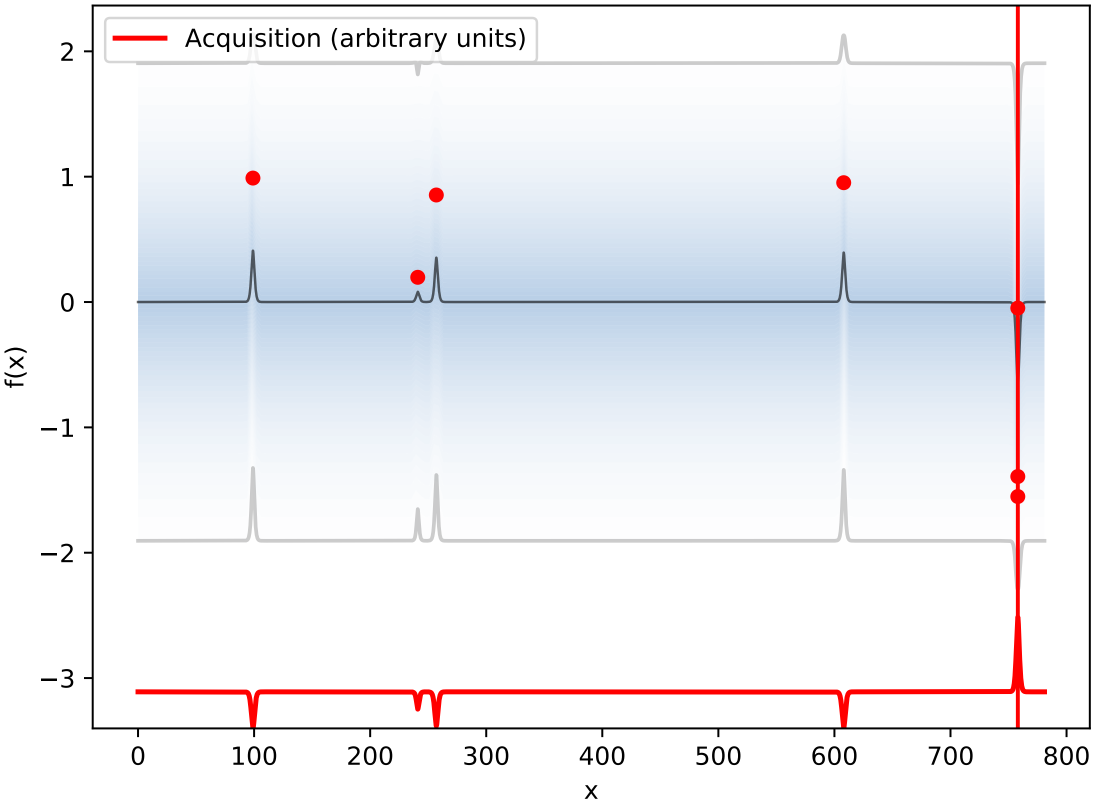


Figure 4.15: Gaussian Process surrogate model and Expected Improvement acquisition function for the separate hyperparameters optimization strategy's layer optimization. The Y-axis is the expected result of a hyperparameter in the surrogate model. X-axis is the hyperparameter, which is this case is the delimiting layer. The black line represent the posterior mean, the gray lines represents the posterior uncertainty, the red dots are tested hyperparameters and their results on the surrogate model, the red vertical line is where the acquisition function would try the next test should it run for another iteration, and, finally, the horizontal red curve is the acquisition function in arbitrary units. A lower posterior mean is better, and a higher value for the acquisition function is where the next hyperparameter values will be taken from.

Layer optimization was successful in increasing the validation accuracy significantly. From table 4.5, we see that in iteration two we achieve a validation accuracy of 0.63, which is 0.22 validation accuracy better than using the default delimiting layer. Aside from the better validation accuracy, we expect the best layer to stay somewhat consistent with the one found in the layer optimization of the shared hyperparameter optimization. From table 4.5, we see that this is not the case. We can see from the table, that the best training run was from the delimiting layer of 758. This is very interesting because of the following:

1. The delimiting layer found by the layer optimization is far from the best delimiting layer found in the best shared hyperparameters optimization strategy, which was 136. This discrepancy indicates that either the Bayesian optimization failed to find the best delimiting layer within the search space, or the best delimiting layer is different for different hyperparameter configurations aside from the pre-trained model.

2. Layer number 758 is tested three times where one of the tests did considerably worse than the other two. We can see in figure 4.5 that iteration two, six, and seven tests the delimiting layer 758. Iteration two and seven have validation accuracies which are within 0.04 of each other, but iteration six, which should also be close to the other two iterations, has a validation accuracy of 0.39. We expect the same hyperparameters to yield roughly the same results. There will always be some difference in the training outcome because of a degree of randomness in the way CNNs are trained. We see no other reason for the varying results other than the degree of randomness. The yellow line in figure 4.13 represents the training run reaching the 0.39 validation accuracy. We can see from this line that it is similar to the light green performance but produces lower validation accuracies. It fails to reach the threshold of 0.5 set by the nonconvergence filtering and is terminated early. However, there is little reason to believe the yellow line would reach much higher results as it was on a downward trend.

3. Only seven iterations were performed by the Bayesian optimization even though we set the maximum allowed iterations to fifteen. The optimization stops performing new iterations when it believes it has reached the best result for validation accuracy. We can see from the second plot in figure 4.14 that the optimization reaches the highest validation accuracy in the second iteration. We see in the first plot from figure 4.14 that it tries different delimiting layers with large distances between them, but in the last iteration tries the same delimiting layer twice in a row. From the results, it seems that the Bayesian optimization stops early because it became certain that 758 was the best delimiting layer after trying it three times. This interpretation of the results is further confirmed by figure 4.15, where we see a plot of the Gaussian Process model and the expected improvement acquisition function. We can see that the model has only found one minimum and that the acquisition function only expects improvement in that location.

Figure 4.15 looks very different than figure4.9. In figure 4.15, we see that the surrogate model is barely being fitted as the posterior mean and uncertainty is almost the same for all the delimiting layer values. The acquisition function does almost no exploration as it does not expect the other areas of the search space to be as good as the one area found in layer number 758. We can see that it tries to get results from 758 three times and gets varied results. However, they are still better than the other areas in the acquisition functions point of view, so the next iteration would be using the same layer. As such, the Bayesian optimization determines that it has found the best achievable validation accuracy with layer number 758. But, as we can see, the area is not fully explored, and something must have affected the surrogate model to be unable to fit properly. This should be explored in future work when optimizing the Bayesian optimization.

4.3.3 Separate Optimizers Optimization Strategy

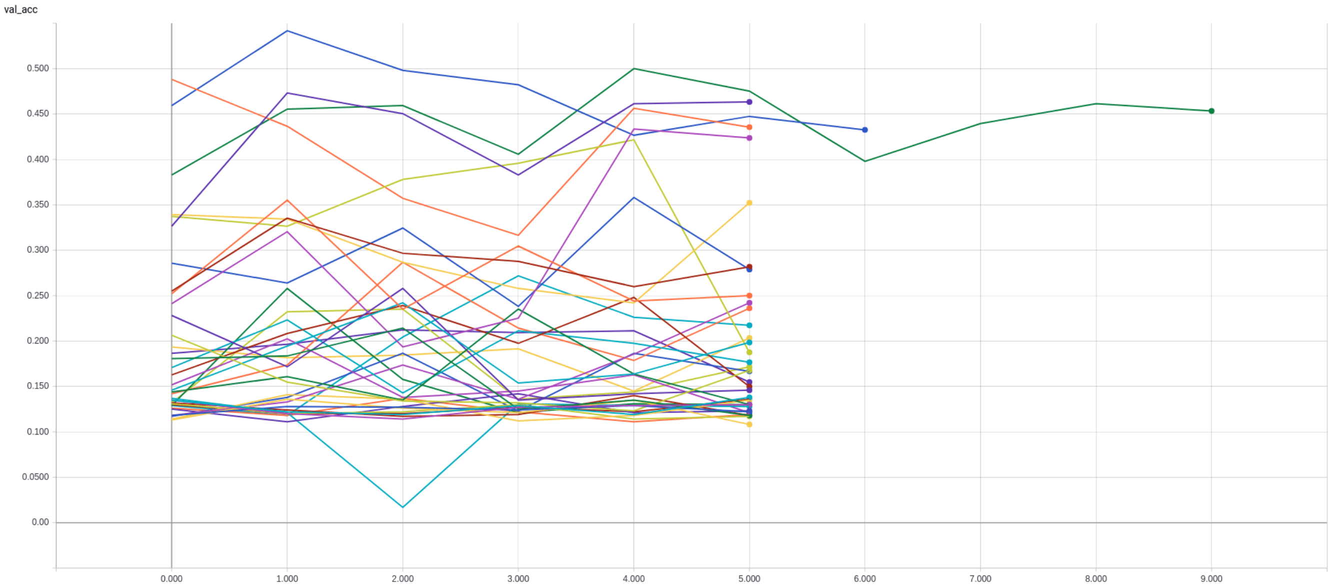


Figure 4.16: The separate optimizations optimization strategy's plot of training runs for each epoch.

The separate hyperparameters optimization strategy failed to reach the same levels of validation accuracy as the shared hyperparameters optimization strategy. The main contributing factor was that the Bayesian optimization did not manage to optimize the hyperparameters in five-dimensional space over the limited number of optimizations imposed by us. The layer optimization did do the job of increasing the validation accuracy, but we saw it failing to explore the search space and ending early as a result. From figure 4.2, we already know that the separate optimizations strategy did worse in terms of validation accuracy than the other strategies. The results were surprising to us. We expected the separate optimizations to solve the problem of dimensionality for separate hyperparameters optimization and using the same hyperparameters for both classification block training and fine-tuning. The problem of dimensionality would be solved by splitting the optimizations in two, making classification block training a three-dimensional search and fine-tuning a two-dimensional search. The optimization split would train different hyperparameters for both of the training types and would solve the problem of using the same hyper-

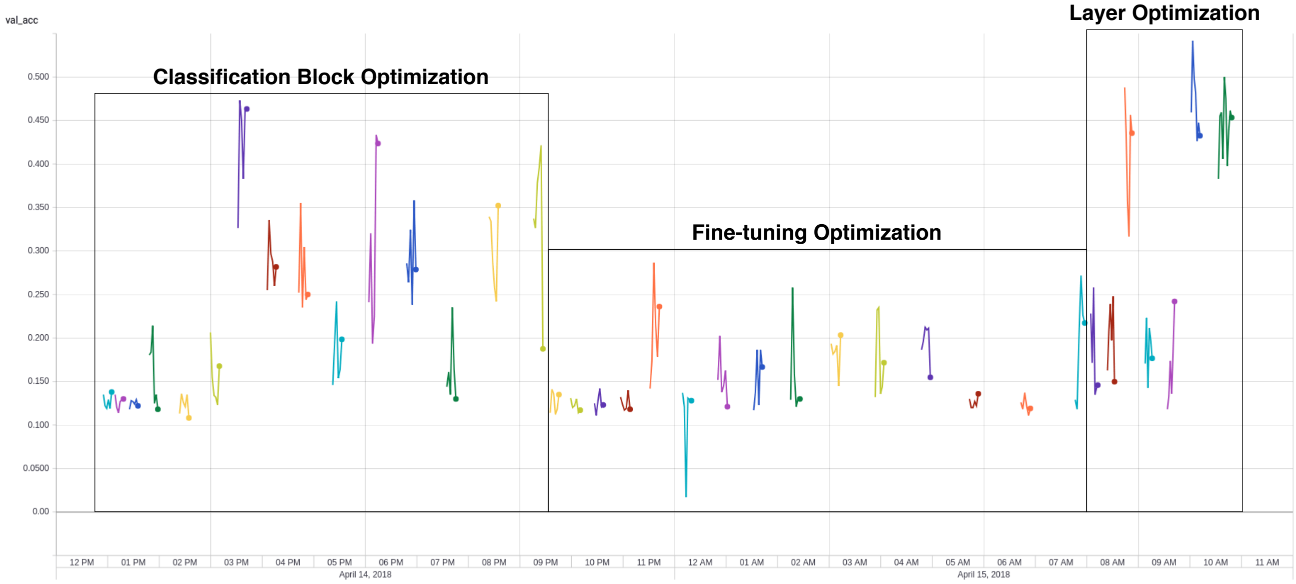


Figure 4.17: The separate optimizations optimization strategy's plot of training runs for time. The classification block optimization, fine-tuning optimization, and layer optimizationa is marked with black boxes.

parameters for both training types. We mentioned in section 4.3.2 when discussing why the separate hyperparameters optimization strategy failed that having different hyperparameters between classification block training and fine-tuning might be a reason why the optimization strategy failed. We will see from the results from the separate optimizations optimization strategy if different hyperparameters are part of the reason it failed.

From figure 4.16, we can see that the separate optimizations strategy fails. The highest validation accuracy is reached in epoch one by the top blue line, but that validation accuracy is barely over 0.5 validation accuracy. The blue line is terminated by both the early stopping and the nonconvergence filtering as it does not improve the validation accuracy over the threshold of 0.5 and does not increase the validation accuracy over the next five epochs. The nonconvergence filtering filters all training runs except the blue and green line. The turquoise line at the bottom even reaches a validation accuracy well below the other lower-achieving training runs of under 0.05 in epoch two, which is by far the lowest across every training run from all optimization strategies.

In figure 4.17, we have the time plot of the separate optimizations optimization strategy marked with a black box in figure 4.2. We have marked the three optimization steps in the separate optimizations strategy in figure 4.17 with black boxes, and we point out the following:

1. The fine-tuning does not achieve better results than the classification block training. The worse results do not necessarily mean the optimization step failed as it uses the default delimiting layer, which could be the reason for the low validation accuracies. If the reason is

the default delimiting layer, we would expect the layer optimization to reach higher validation accuracies than the classification block training. We can see that this is the case in figure 4.17. Layer optimization is the only optimization step that reaches above 0.5 validation accuracy.

2. Classification block training and fine-tuning take roughly the same time to finish, but the layer optimization takes a fraction of the time. We can see that the layer optimization only runs for 7 epochs, which is the same number of epochs the layer optimization of the separate hyperparameters optimization strategy ran.
3. The whole optimization strategy ran for almost 24 hours, which is the lowest of all the strategies. However, as we can see from figure 4.16 and 4.17, only two training runs passed the threshold for not being terminated by nonconvergence filtering, and the longest run lasted for nine epochs. By comparing the separate optimizations strategy with the shared hyperparameters strategy which had a much lower rate of nonconvergence filtering, we see that nonconvergence filtering does help considerably with saving time on training runs which produce low validation accuracies.
4. The optimization of the classification block training and the layer optimization shows convergence. Both show an increase, where the optimization step starts with low validation accuracies, but over time reach higher levels as it converges. However, we can see in both optimization steps that the convergence happens as a leap and not progressively. In the classification block optimization, we see that the validation accuracy leaps from 0.21 to 0.47 around April 14th 03:30 PM. After the leap, the average validation accuracy of the next training runs is higher than the average validation accuracy of the previous training runs. The same behavior is seen in the layer optimization after the orange line peaks.

Classification Block Optimization

We can see in figure 4.18 that no training runs pass the threshold of 0.5 and are consequently stopped early. Many of the runs are producing validation accuracies around 0.125, which is the validation accuracy we would get should we classify randomly. The highest yielding training run is represented by the dark purple top line, which only reaches a top validation accuracy of 0.47.

From table 4.6 and figure 4.19 we can see some strange behaviour by the Bayesian optimization. In table 4.6, we see that the hyperparameters barely changes between iteration seven and fifteen. The same behavior is reflected in the first plot in figure 4.19, where we can see that the first iterations explore the search space, but after iteration 6 in the plot, which represents the distance between iteration 7 and 8 in table 4.6, the distance is low. This indicates that the Bayesian optimization's

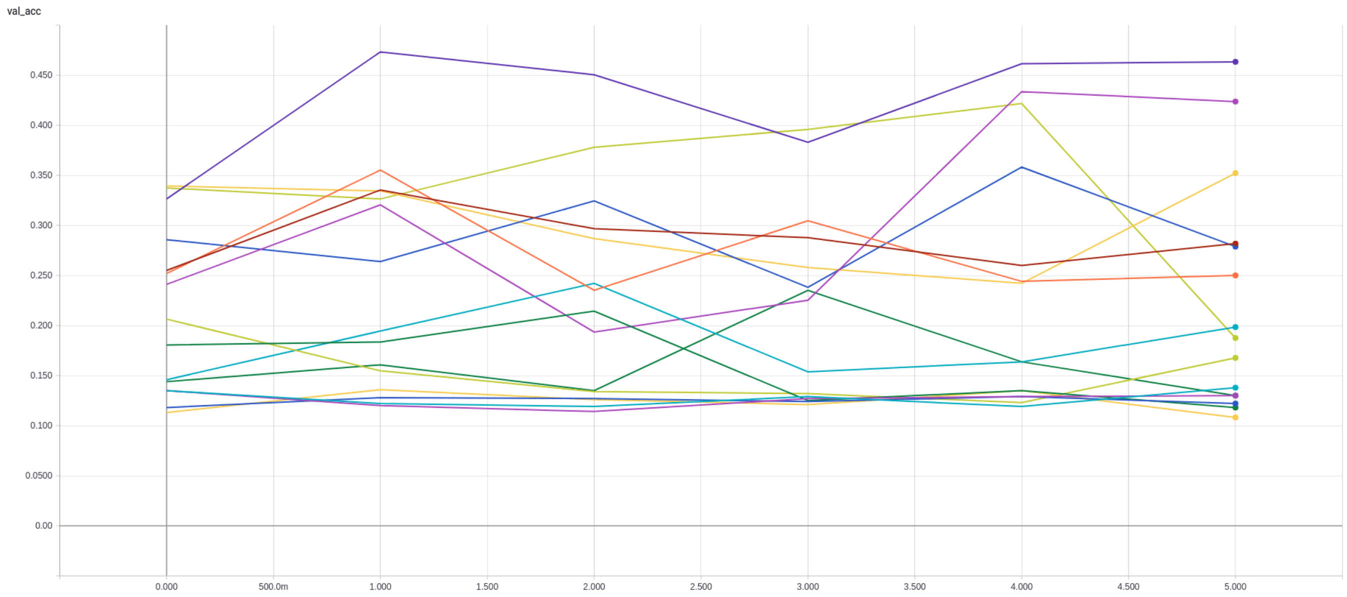


Figure 4.18: The separate optimizations optimization strategy's classification block optimization run for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

expected improvement acquisition function explores the search space in the early iterations then exploits the area immediately after determining it is the best area. We can see that iteration 6 in the second plot from figure 4.19 coincides with iteration 6 in the first plot and iteration 7 in figure 4.6. This iteration is where the best validation accuracy of 0.47 is reached, which was significantly better than the previous best of 0.21. As a result, the Bayesian optimization decided that iteration 7 in table 4.6 had the best hyperparameters, so it stopped exploring and started exploiting the search space area by choosing hyperparameters close to the highest yielding combination. Considering the whole run did poorly compared to the classification blocks trained in the shared hyperparameter optimization strategy, we can conclude that this is a weakness in the Bayesian optimization as it does not properly explore the search space and settles too early. Changing the parameters of the Bayesian optimization could help alleviate the problem, but that is outside of the scope of the thesis and should be looked into for future work.

Fine-tuning Optimization

The separate optimizations optimization strategy's fine-tuning optimization is a two-dimensional optimization since we use the best CNN model from the classification block optimization. A lower dimensionality would intuitively make the task of optimization easier for the Bayesian optimization. However, from figure 4.20 we see the same behaviour as for the classi-

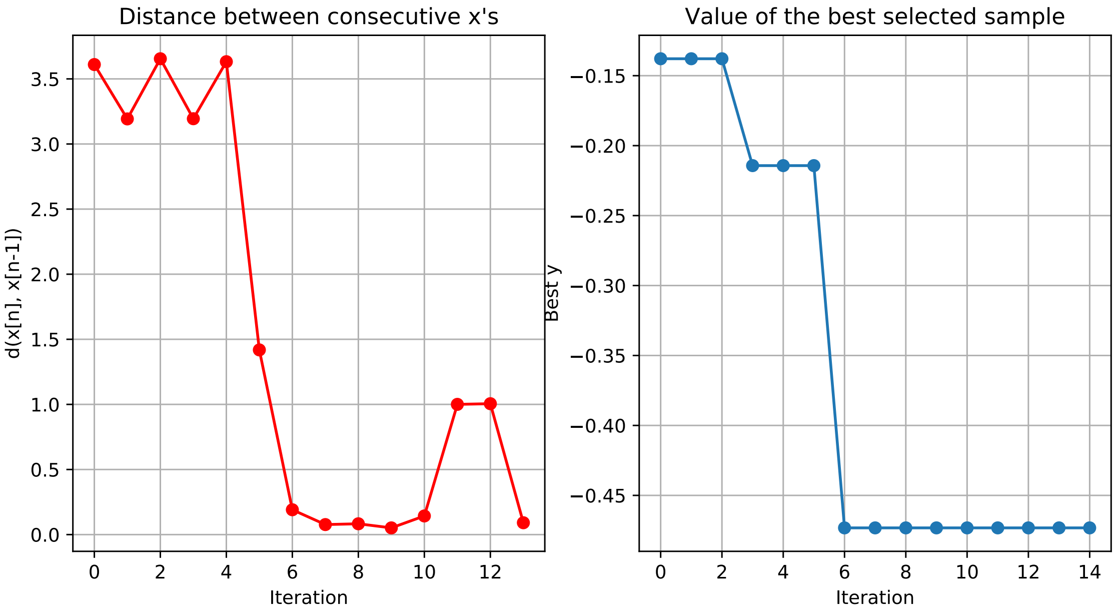


Figure 4.19: Convergence plot of the separate optimizations optimization strategy's classification block optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

fication block optimization figure 4.18 discussed in section 4.3.3. Similarly to the classification block optimization, we see that none of the training runs escape nonconvergence filtering. Additionally, most of the training runs yield validation accuracies around 0.125, which is what random classification would yield. However, the validation accuracies of the best training runs are worse than for the classification block optimization and reach at best 0.29. At worst it reaches 0.02, which means the CNN has learned wrong and classified not randomly, but based on wrong knowledge. The validation accuracies of the fine-tuning being worse than the classification block training could be attributed to the default layer being bad for the chosen model. In contrast to the other optimization strategies, the default delimiting layer does not affect the choice of the best pre-trained model. Instead, the pre-trained model's ability to train the best classification block is how we choose it. Whether this is problematic or not is unknown to us. Finding out would require more runs for comparison. We had limited resources, so finding out will have to be future work.

To further highlight the similarities between the classification block

Iteration	Val Acc	Model	Optimizer	Learning Rate
1	0.1379	Xception	Nadam	0.7906
2	0.1349	ResNet50	RMSprop	0.6060
3	0.1290	InceptionV3	Adamax	0.1660
4	0.2143	DenseNet121	RMSprop	0.7641
5	0.1359	InceptionResNetV2	Adamax	0.3135
6	0.2063	DenseNet169	RMSprop	0.7528
7	0.4732	DenseNet121	SGD	0.8635
8	0.3353	DenseNet121	SGD	0.6727
9	0.3552	DenseNet121	SGD	0.7496
10	0.2421	DenseNet121	SGD	0.8327
11	0.4335	DenseNet121	SGD	0.7811
12	0.3581	DenseNet121	SGD	0.9243
13	0.2351	DenseNet121	Nadam	0.9456
14	0.3522	DenseNet121	SGD	0.8413
15	0.4216	DenseNet121	SGD	0.9324

Table 4.6: Validation accuracy and hyperparameters for each Bayesian optimization iteration in the separate optimizations optimization strategy's classification block optimization.

optimization and the fine-tuning optimization, we look to figure 4.21 and table 4.7. We can see that both plots in figure 4.21 share similarities to the plots in figure 4.19 from the classification block optimization. The first graphs in each figure are similar in that each iteration has a greater distance between them until iteration four in the first graph in figure 4.21 and iteration six in the first graph in figure 4.19. Then after several iterations with almost no distance between the hyperparameters, the optimization gets a spike over three iterations with a higher distance between the hyperparameters. By looking the second graph in each figure, we see that the best validation accuracy is achieved in the fourth and sixth iteration and therefore correlates to the first plot for both figures. We can tell from this behavior that the Bayesian optimization in both optimizations runs emphasized exploring the search space until it found a better validation accuracy it wanted to exploit, meaning it tries different hyperparameters close in the search space to the current best hyperparameters. Because we use the default parameters for the Bayesian optimization, the optimization will first use five iterations with randomly distributed hyperparameters to map the search space. We can see this affecting the run, as the optimization randomly found one area in the search space much better than the others and decided to use most of the remaining optimization runs into exploiting this area. For both runs, it could not achieve better results from the

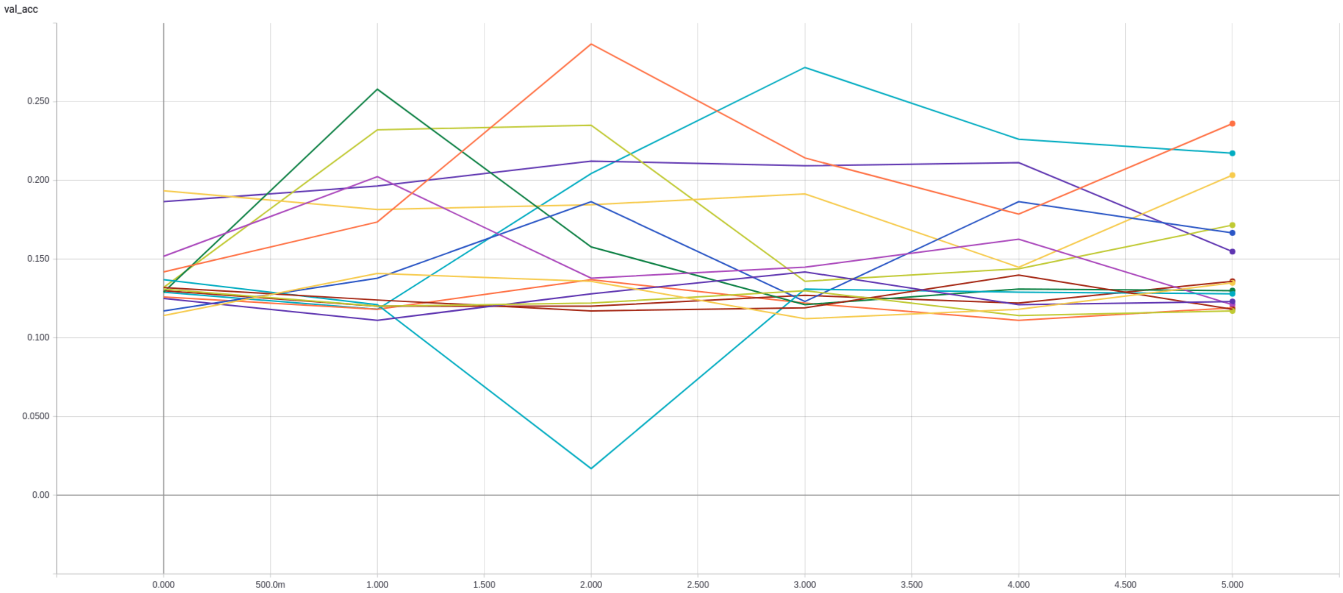


Figure 4.20: The separate optimizations optimization strategy's fine-tuning optimization run for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

exploitation, and therefore it starts exploring again at the end. Future work could try to increase the initial random mapping iterations in an attempt to increase the odds of finding more high achieving areas in the search space.

The fine-tuning step uses a hyperparameter set of two: The gradient descent optimization function and the learning rate. The previous classification block optimization gives the model. As a result, the search space is two-dimensional, making it possible to plot it. In figure 4.22, we see the Gaussian model, and the acquisition function plotted. Instead of having it all plotted together in one graph, we must use three different graphs to plot it because of the two-dimensionality. The graphs show the posterior mean, the posterior standard deviation and the acquisition function as a color gradient. The Y-axis, X2, is the normalized learning rate and the X-axis, X1, is the index of the gradient descent optimization function in a list. The list is the following with the corresponding index:

0. Nadam
1. SGD
2. RMSprop
3. Adagrad
4. Adam
5. Adamax

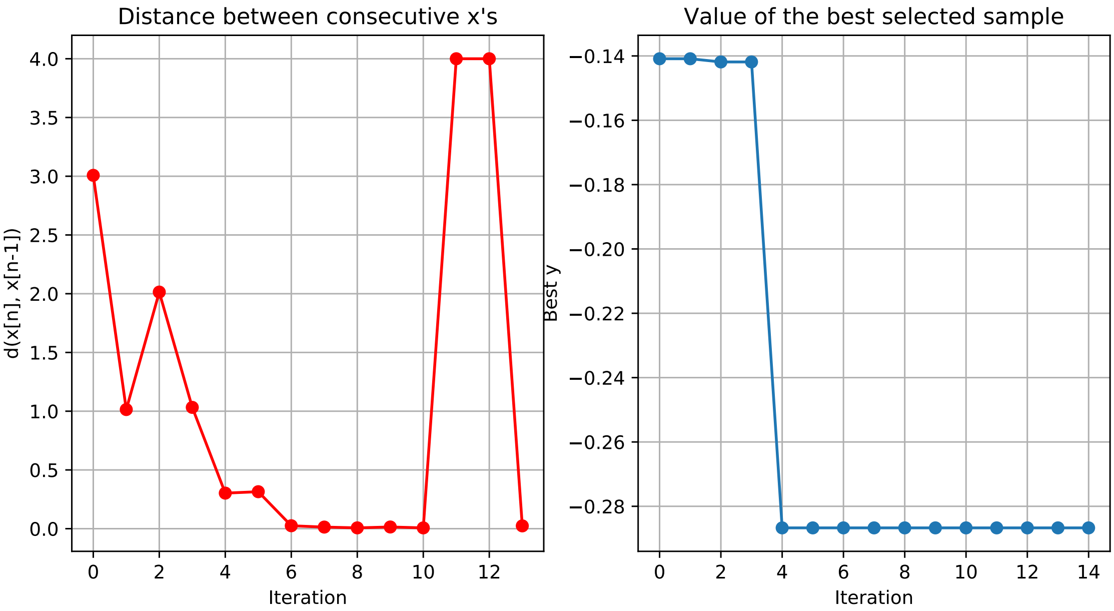


Figure 4.21: Convergence plot of the separate optimizations optimization strategy's fine-tuning optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

6. Adadelta

The red dots in the posterior mean and posterior standard deviation plots are the tested hyperparameters, and the gradient surrounding them is the posterior mean calculated from the result of an iteration with those hyperparameters. The black dot in the acquisition function is wherein the search space the next hyperparameters would have been taken from should the optimization do another iteration.

The plots in figure 4.22 are interesting as they show us some important aspects of Bayesian optimization:

1. Large portions of the search space are unexplored. For example, the gradient descent functions Nadam, Adagrad, and Adadelta is not even tested once.

2. The areas that are explored and their gradients are limited to their class. This means that Bayesian optimization does not create a relationship between each gradient descent optimizer, but treats them

Iteration	Val Acc	Optimizer	Learning Rate
1	0.1409	RMSprop	0.5256
2	0.1310	Adamax	0.3128
3	0.1419	Adam	0.4809
4	0.1399	RMSprop	0.2453
5	0.2867	SGD	0.5022
6	0.1369	SGD	0.8047
7	0.2024	SGD	0.4901
8	0.1865	SGD	0.5150
9	0.2579	SGD	0.5014
10	0.2034	SGD	0.5083
11	0.2351	SGD	0.4940
12	0.2123	SGD	0.5008
13	0.1359	Adamax	0.5124
14	0.1369	SGD	0.5009
15	0.2718	SGD	0.4770

Table 4.7: Validation accuracy and hyperparameters for each Bayesian optimization iteration in the separate optimizations optimization strategy's fine-tuning optimization.

as separate with their own range of possible learning rates. This contradicts our observations in previous sections, such as section 4.3.1, where we saw fluctuations between Bayesian optimization iterations that we attributed to a numeric relationship between the gradient descent optimization functions. The observations in figure 4.22 instead show that Bayesian optimization is able to separate each gradient descent class and find learning rates for that class in the search space.

3. We see from the posterior mean function that there is only one area in the search space which gives lower posterior mean than 0. As a result, instead of exploring further, the acquisition function continuously tries to exploit that area. It seems Bayesian optimization tries too much to exploit an area it deems good instead of exploring the search space for better solutions. Perhaps adding a threshold for when the Bayesian optimization can begin exploiting would increase the final results.

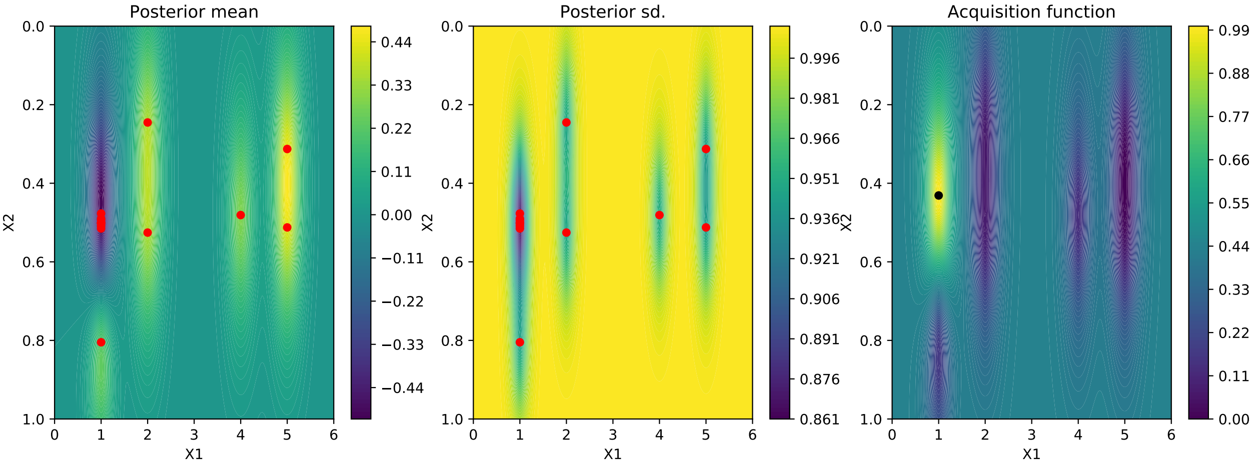


Figure 4.22: Gaussian Process surrogate model and Expected Improvement acquisition function for the separate optimizations optimization strategy's fine-tuning optimization. The Y-axis is the normalized learning rate domain, and the X-axis is the index of each gradient descent optimization function in a list. The color gradient represents the posterior mean in the first graph, the posterior standard deviation in the second graph, and the acquisition function in the final graph. The graph shows the model after the optimization. The red dots in the first two graphs are the tests that have been done during the optimization. The black dot in the acquisition function is where the next test would have occurred should the optimization continue for another iteration.

Layer Optimization

The classification block optimization failed to produce a model which could compete with the models produced in the shared hyperparameters optimization strategy. The fine-tuning optimization failed to improve the classification block model, which could be attributed to the default delimiting layer. However, compared to the share hyperparameters optimization, where the fine-tuning increased the classification block model's validation accuracy, the delimiting layer seems not to be the cause for the failed fine-tuning. By analyzing the layer optimization, we can examine in what way the default delimiting layer have affected the fine-tuning by looking at the attained validation accuracies after optimization.

We start by looking at figure 4.23. In this figure, we can see the following:

1. Only two lines reach above 0.5 validation accuracy and avoid being stopped early by nonconvergence filtering in the fifth epoch.

2. Four of the training runs are producing validation accuracies below 0.3, while three give validation accuracies above 0.3. There is a clear separation here.

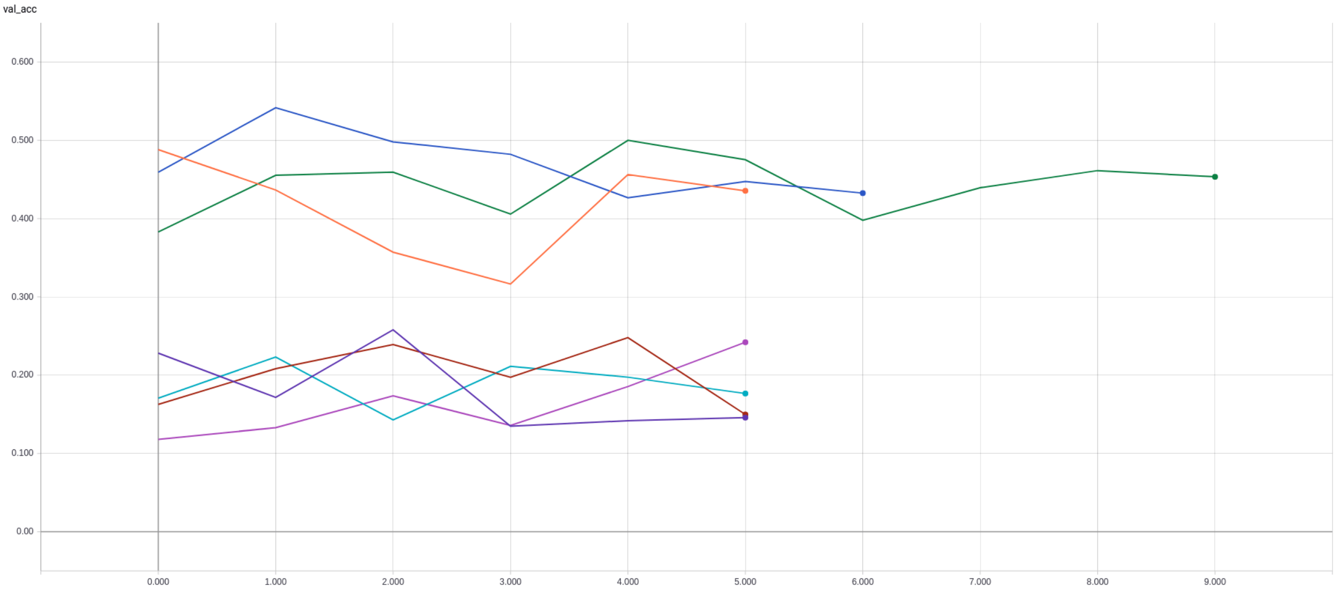


Figure 4.23: The separate optimizations optimization strategy's layer optimization run for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

3. The validation accuracy achieved by the highest yielding training run is 0.54, which is higher than both the classification block and fine-tuning optimization.

4. Similarly to the layer optimization for the separate hyperparameters optimization strategy, only seven iterations of the allowed fifteen were conducted. Seven iterations are low because the Bayesian optimization by default uses five iterations for randomly mapping the search space. It seems the Bayesian optimization, in this case, found a validation accuracy it deemed good and instead of searching more of the area, was confident it had found the global best result.

We see in table 4.8 and figure 4.24, that the layer optimization resembles the layer optimization of the separate hyperparameters strategy. The two layer optimizations are similar in that they both last for only seven iterations instead of the fifteen allowed. Additionally, table 4.8 and table 4.5 show that the layer optimization in both optimization strategies first does the five random iterations mapping the search space, then one of the five iterations achieve much better results than the other iterations. The optimization then tries two more runs at the same search space area as the current best and gives up early when it cannot find a better result.

The convergence plots show the same behavior. The first plot of figure 4.24 first has high, varying distance between each tested hyperparameter, but at the final iteration, after the initial random search, has no distance between the hyperparameters anymore and stops. The second plot in

Iteration	Val Acc	Layer
1	0.2579	154
2	0.2480	219
3	0.4881	425
4	0.2232	149
5	0.2421	184
6	0.5417	427
7	0.5000	427

Table 4.8: Validation accuracy and hyperparameters for each Bayesian optimization iteration in the separate optimizations optimization strategy's layer optimization.

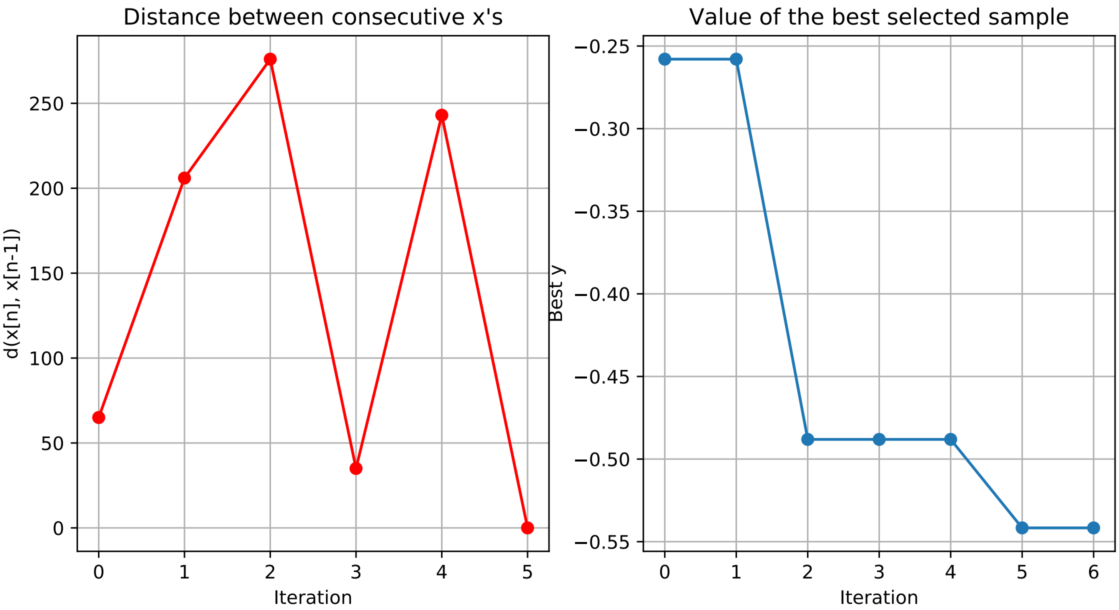


Figure 4.24: Convergence plot of the separate optimizations optimization strategy's layer optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

figure 4.24, shows that the optimization does improve the result over time. However, it still stops after trying the same layer twice and failing to reach a higher result in the final try.

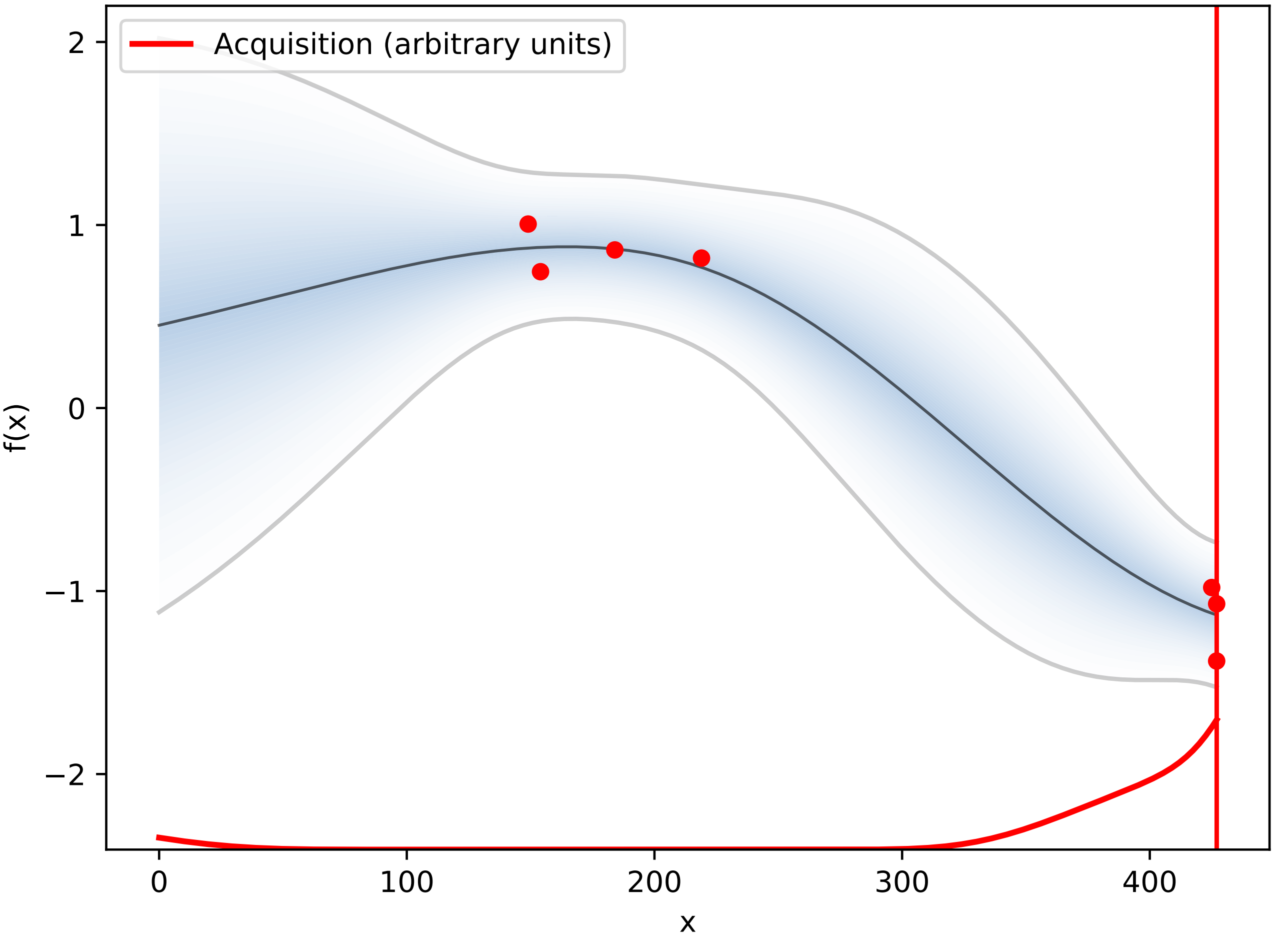


Figure 4.25: Gaussian Process surrogate model and Expected Improvement acquisition function for the separate optimizations optimization strategy's layer optimization. The Y-axis is the expected result of a hyperparameter in the surrogate model. X-axis is the hyperparameter, which is this case is the delimiting layer. The black line represents the posterior mean, the gray lines represents the posterior uncertainty, the red dots are tested hyperparameters and their results on the surrogate model, the red vertical line is where the acquisition function would try the next test should it run for another iteration, and, finally, the horizontal red curve is the acquisition function in arbitrary units. A lower posterior mean is better, and a higher value for the acquisition function is where the next hyperparameter values will be taken from.

Looking at the surrogate model and acquisition function in figure 4.25, we can see that little of the model was explored and that the random initial search was primarily based around the middle of the search space, except the one at the highest delimiting layer values. We can see that the model suffers from this initial search being so centered as we only have two areas of information. We can see from the acquisition function that it thinks the best area is at the end of the delimiting layer search space. The only area it still thinks it would be worth exploring later is at the beginning of the search space. We see this behavior in the other strategies as well, and it would seem future work would have to solve this by using a higher result threshold and more iterations for the initial random search space mapping.

4.3.4 Best Trained Model

Table 4.9: Table of hyperparameter values of the best trained model on the Kvasir dataset.

Hyperparameter	**Optimal value**
Pre-trained Model	InceptionResNetV2
Model Optimizer	Adadelta
Learning Rate	0.8417
Delimiting layer	136

The best performing hyperparameters over all the optimization strategies from the full optimization run of Kvasir presented in the previous sections were achieved by the shared hyperparameters optimization strategy and are listed in table 4.9. The iteration reached a validation accuracy of 0.8929 and is listed as iteration 12 in table 4.3. The layer optimization reaching the 0.8929 validation accuracy is built on the best model from the classification block optimization, which reached a validation accuracy of 0.8562 and is listed as iteration 4 in table 4.2. In table 4.9, we see the hyperparameters achieving the best validation accuracy. Since the best validation accuracy comes from the shared hyperparameters optimization strategy, the same model optimizer and learning rate are used for both the classification block training and the fine-tuning. The best delimiting layer is 136, which is low compared to the number of layers in the CNN model. Table 3.2 in the hyperparameters section of the methodology chapter shows an overview of the pre-trained models used. Here we see that the InceptionResNetV2 model, which is the model reaching the highest validation accuracy, contains 782 layers and is the deepest model with 572 different blocks.

We used metrics presented in the metrics section 3.2 of the methodology chapter to evaluate the performance of the best-trained model. The metrics were calculated by running predictions on the validation set of 2400 images, 300 for each class, from the Kvasir dataset. Using these metrics gives a deeper insight into the behavior of the trained model and makes the result easier to compare to related work using the same metrics. We

Class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Dyed lifted polyps	15	24	2076	285	0.94	0.98	0.93	0.92	0.95	0.99
Dyed resection margins	23	8	2092	277	0.95	0.99	0.94	0.97	0.92	1.00
Esophagitis	181	4	2096	119	0.56	0.92	0.59	0.97	0.40	1.00
Normal cecum	8	18	2082	292	0.96	0.99	0.95	0.94	0.97	0.99
Normal pylorus	2	18	2082	298	0.97	0.99	0.96	0.94	0.99	0.99
Normal z-line	7	183	1917	293	0.76	0.92	0.74	0.62	0.98	0.91
Polyps	19	7	2093	281	0.96	0.99	0.95	0.98	0.94	1.00
Ulcerative colitis	20	13	2087	280	0.94	0.99	0.94	0.96	0.93	0.99
Average	34.38	34.38	2065.62	265.62	0.88	0.97	0.87	0.91	0.89	0.98

Table 4.10: Metrics for each class after hyperparameter optimization for the best model on the Kvasir dataset, including average values for comparing with baseline metrics presented in the Kvasir paper [98].

can see in table 4.10, that the trained model reach F_1 scores of well above 0.90 in all the classes except *Esophagitis* and *Normal z-line*. We see that 181 images are false-negative for the esophagitis class and 183 are false-positive for the normal z-line class. For esophagitis, we see that there are more false-negative than true-positive, which means most of the images for esophagitis was wrongly classified as normal z-line. However, aside from the esophagitis being wrongly classified as normal z-line, the classification accuracy for all other classes are high.

We can compare the metrics to those obtained from the Kvasir paper [98]. The metrics from the Kvasir paper serves as comparisons to and baselines for our results and are listed in table 4.11. In the table, there are metrics from results of experiments based on global features, GF in the table, CNNs and transfer learning, TFL in the table. The first three methods in the table are methods based on CNN classification, the next six methods are based on methods using handcrafted global image features, and the last method is random classification. The methods were used on the Kvasir dataset, and the metrics are the average of all the eight classes. Information about the methods and their implementations can be found in the Kvasir paper [98].

The 3-layer and 6-layer CNN implementations were trained from scratch using Keras. The CNN implementations were custom basic implementations not based on one of the CNN models used as hyperparameters in our experiments. The custom implementations contained either three or six layers of convolutional layers with the rectified linear unit (ReLU) [86] as activation function and max-pooling for pooling. For all layers, a dropout of 0.5 was included, and the last classification step was performed using two dense layers with first ReLu and then Sigmoid as activation functions. The Adam optimizer was used with 200 epochs of

Method	F_1	ACC	MCC	PREC	REC	SPEC
3 Layer CNN	0.453	0.959	0.430	0.589	0.408	0.890
6 Layer CNN	0.651	0.914	0.602	0.661	0.640	0.953
Inception v3 TFL	0.693	0.924	0.649	0.698	0.689	0.957
JCD Random Forest	0.706	0.927	0.666	0.708	0.710	0.958
2 GF Random Forrest	0.711	0.928	0.672	0.713	0.715	0.959
2 GF Logistic Model Tree	0.705	0.926	0.664	0.706	0.707	0.958
6 GF Random Forrest	0.727	0.933	0.692	0.732	0.732	0.962
6 GF Logistic Model Tree	0.747	0.937	0.711	0.748	0.748	0.964
Random/Majority	0.000	0.016	0.666	0.016	0.125	0.000

Table 4.11: Performance metrics for different methods of image classification done on the Kvasir dataset [98]. The methods and their metrics are taken from the Kvasir paper. TFL stands for transfer learning and GF stands for global features. JCD stands for joint composite descriptor, and is a global color and texture feature. The metrics from JCD random forest and random/majority was used as baseline for the Kvasir run, but we use all the methods as baseline and comparison to our optimized solution.

training.

In the transfer learning implementation, an InceptionV3 model was used. The training protocol was similar to the one we used for our experiments. For the classification block training, Pogorelov et al. used two dense classification layers in contrast to our global average 2d pooling and dense layer classification block. Another difference is that the classification block was trained for 1,000 epochs using the RMSprop optimizer. The fine-tuning was done in the same manner as in our experiments, but on the two top convolutional layers of the classification block model. The fine-tuning was done with the SGD optimizer and a ”low learning rate” [98].

The global feature methods used different image features for classification using an open source software library for content-based image retrieval called LIRE [80]. The extracted features are JCD, Tamura, Color Layout, Edge Histogram, Auto Color Correlogram and Pyramid Histogram of Oriented Gradients. The 2 GF runs use combined JCD and Tamura resulting in a feature vector of 187. The 6 GF runs use all of the previously mentioned features, resulting in a feature vector of 1186. These combinations were chosen based on previous work done by Pogorelov et al. Classifiers used were Random Forest and Logistic Model Tree provided by the Weka machine learning library [141].

By comparing the average metrics from the result of our best-trained model in table 4.10 with those of the baseline metrics provided by the Kvasir dataset paper in table 4.11, we can see that our model outperforms all of the methods in all metrics. In comparison, the transfer learning method, which reached an F_1 score of 0.70, had an F_1 score 0.18 lower than our optimized method reached, which achieved an F_1 score of 0.88. Our hyperparameter optimized transfer learning method is also better than the 6 global feature logistic model tree, which was the best performing baseline with 0.75 F_1 score. Our method's F_1 score is 0.13 better.

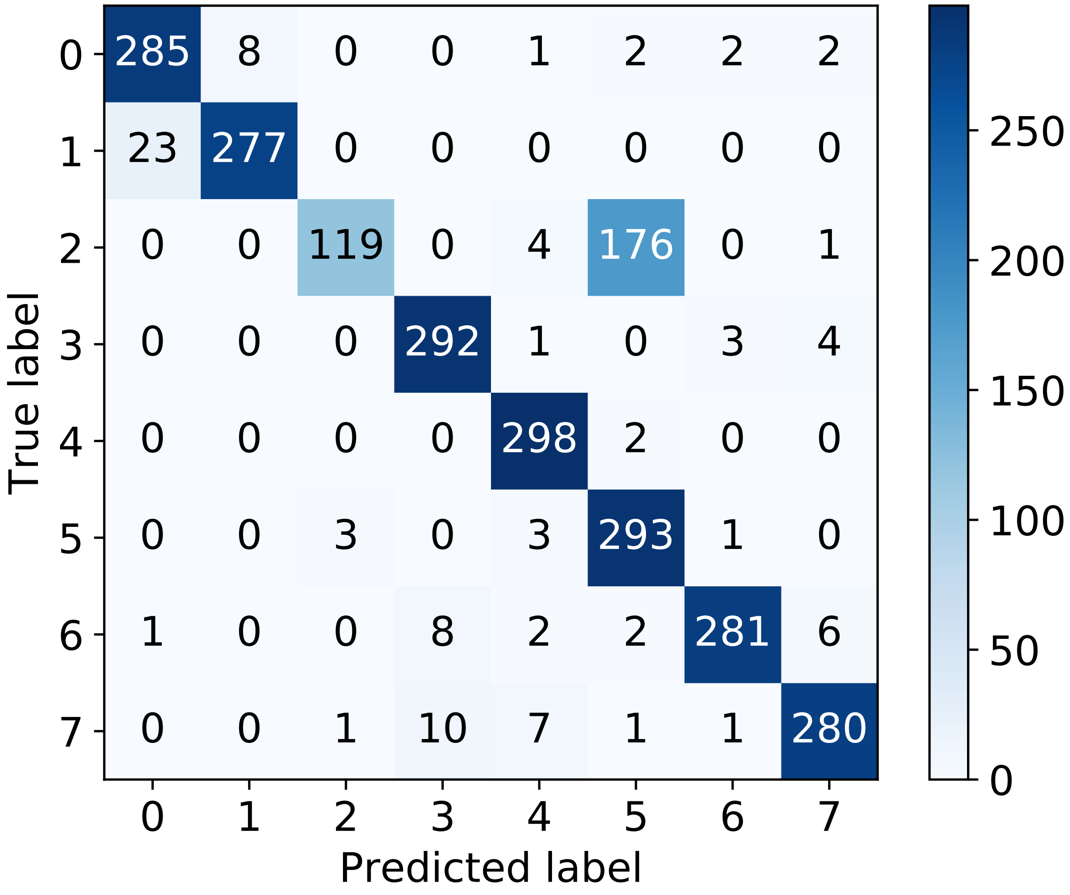


Figure 4.26: Confusion matrix for the best model trained on Kvasir. There are 300 images per class and the labels are the following: 0. Dyed lifted polyps, 1. Dyed resection margins, 2. Esophagitis, 3. Normal cecum, 4. Normal pylorus, 5. Normal z-line, 6. Polyps, 7. Ulcerative colitis.

Compared to the baseline methods provided in the Kvasir dataset, our optimized method did significantly better. However, from the confusion matrix in figure 4.26 of the optimized model, we can see that the model classifies most images correctly. The labels in the plot correspond to the following classes:

0. Dyed lifted polyps
1. Dyed resection margins

2. Esophagitis
3. Normal cecum
4. Normal pylorus
5. Normal z-line
6. Polyps
7. Ulcerative colitis

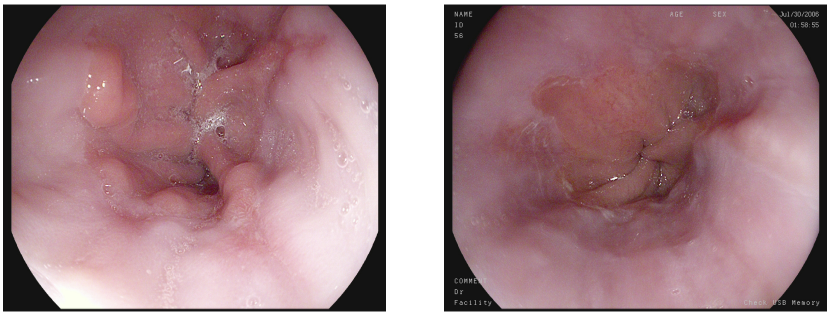
From the confusion matrix, we can see that there is a smaller outlier, which is a misclassification of dyed resection margins as dyed lifted polyps. The reason for the misclassification is probably due to both classes having blue dye covering the area of interest. However, the outlier is small in comparison to the correctly classified images. Only 23 of the 300 dyed resection margins images are classified wrongly into, and all of the 23 are classified to the dyed lifted polyps class.

The big outlier is the one we mentioned previously in the current section: The esophagitis class is wrongly classified as a normal z-line in 176 instances, which are more instances of wrongly classified images than correctly classified images for the esophagitis class. This misclassification is the only reason the average F_1 score is not around 0.95. However, looking at figure 4.27, we can see a side-by-side comparison of the two classes. From the comparison, we can see that there are visual similarities which can even make it difficult for humans to distinguish. It is our opinion that the similarities are so striking, that the algorithm, similarly to a human, is unable to spot the differences. This leaves one question: If some images are so similar that they are interchangeable, why are there almost no misclassifications where the normal z-line class is classified as the esophagitis class? By looking at the confusion matrix in figure 4.26, we see the misclassifications where the normal z-line class is classified as the esophagitis class involving only three images. We can only speculate that it must be of some CNN implementation-specific reason.

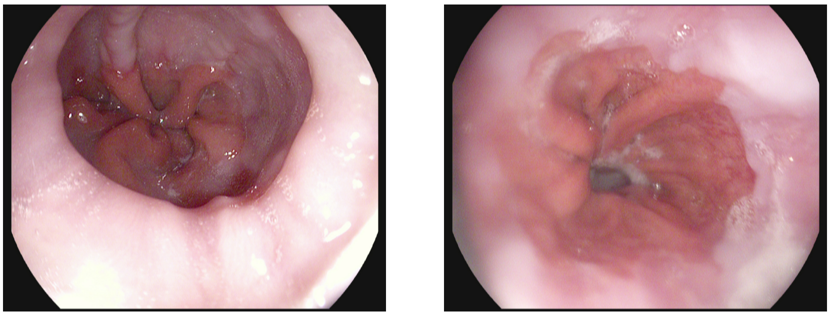
4.4 Results for Nerthus

We also ran a test for the Nerthus dataset. However, we came to many of the same conclusions from the Nerthus results as we did for the tests on the Kvasir dataset. We will, therefore, only briefly discuss them and present the best-trained model. The results not presented here will be added to appendix B.

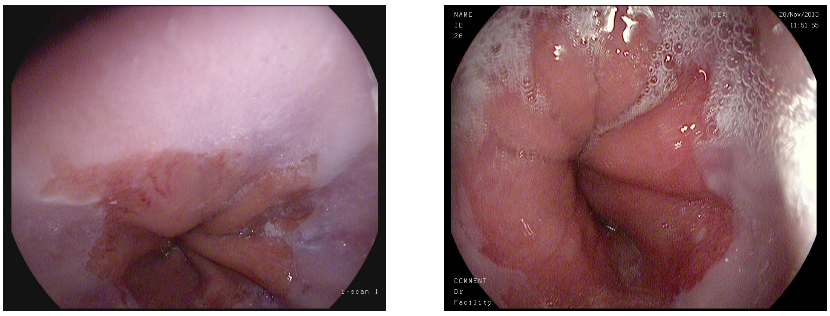
The Nerthus dataset differs from the Kvasir dataset in many ways, and as such makes it impossible to compare to the Kvasir experiment directly. However, we can still see much of the same behavior from the optimization run, and we can see if we reach some of the same results for the optimization strategies. In figure 4.28, we can see the whole experiment for Nerthus plotted in the same graph with validation accuracy for each



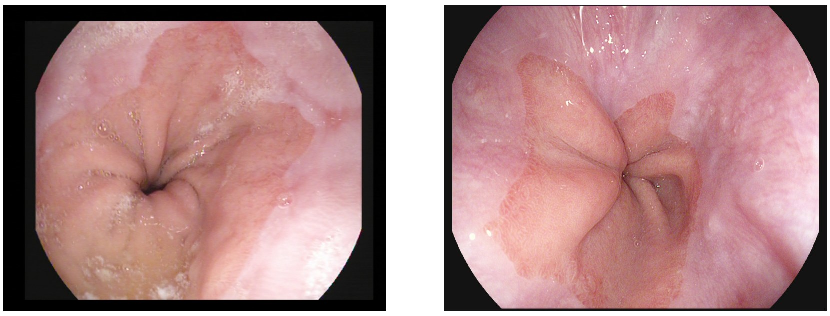
(a) Esophagitis (b) Normal z-line



(c) Esophagitis (d) Normal z-line



(e) Esophagitis (f) Normal z-line



(g) Esophagitis (h) Normal z-line

Figure 4.27: Similar images from two different classes in the Kvasir dataset [98]. The figures are side-by-side comparisons of selected images from the esophagitis class and the z-line class.

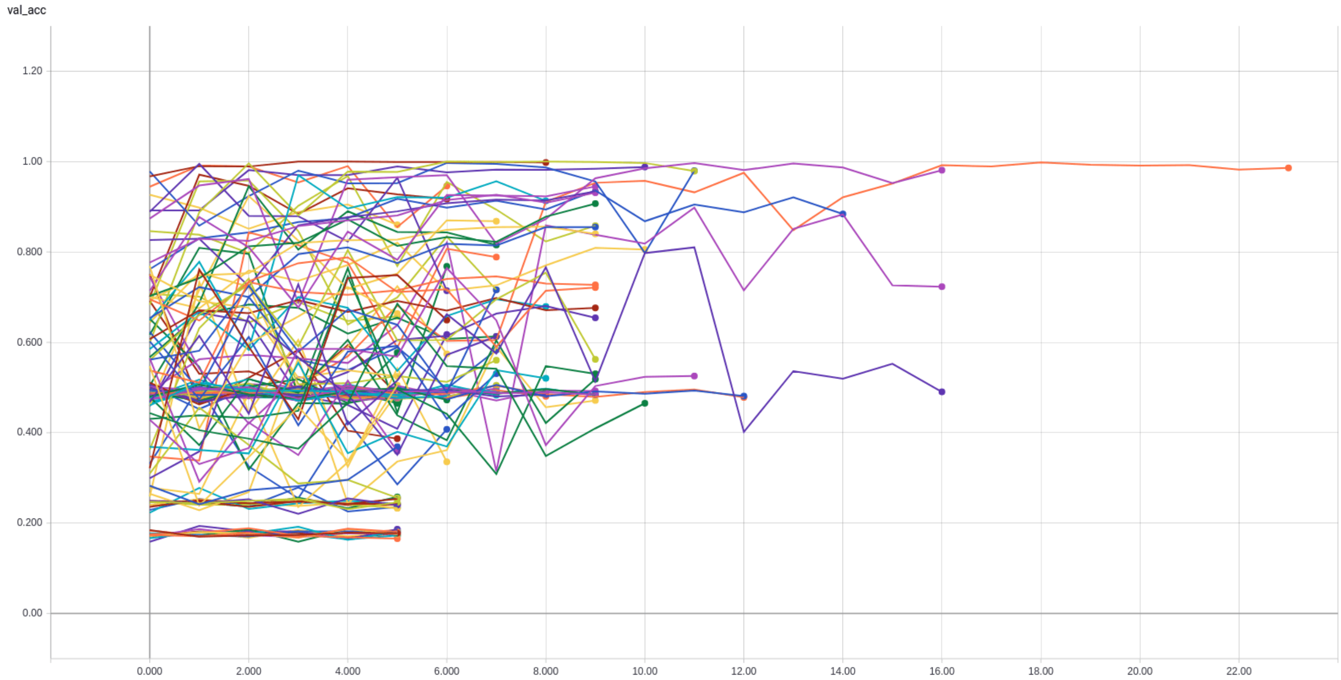


Figure 4.28: Plot of all training runs done on the Nerthus dataset [97]. Each line represents a training run, which can be a classification block training run or fine-tuning training run. The Y-axis is the validation accuracy and the X-axis is the current epoch.

epoch. We see that in contrast to the Kvasir experiment, the majority of training runs are not stopped early by the nonconvergence filtering. We see that many training runs are centered around 0.5 validation accuracy, and of those that got stopped early by the nonconvergence filtering, most were around 0.24 and 0.18 validation accuracy, which were the lowest validation accuracies. Additionally, we can see that the highest achieving runs achieved a validation accuracy of 1, which means the models managed to classify every image, or enough to round up to 1. Lastly, we see that many training runs fluctuate and oscillate heavily in addition to some training runs lasting for many iterations. We also saw this when analysing the Kvasir run in figure 4.1 in section 4.3.

In figure 4.29, we can spot the differences between each optimization strategy and each optimization step within each strategy. We can see that it is the separate hyperparameters optimization strategy that takes the most amount of time to finish instead of the shared hyperparameters optimization strategy which took the most amount of time in the Kvasir test. The reason for this is because of the nonconvergence filtering.

We see many of the same features in the Nerthus training as we did in the Kvasir training shown in figure 4.2:

1. The shared hyperparameter optimization strategy is the best performing strategy, reaching a validation accuracy of 1. The model optimization in the shared hyperparameter optimization strategy

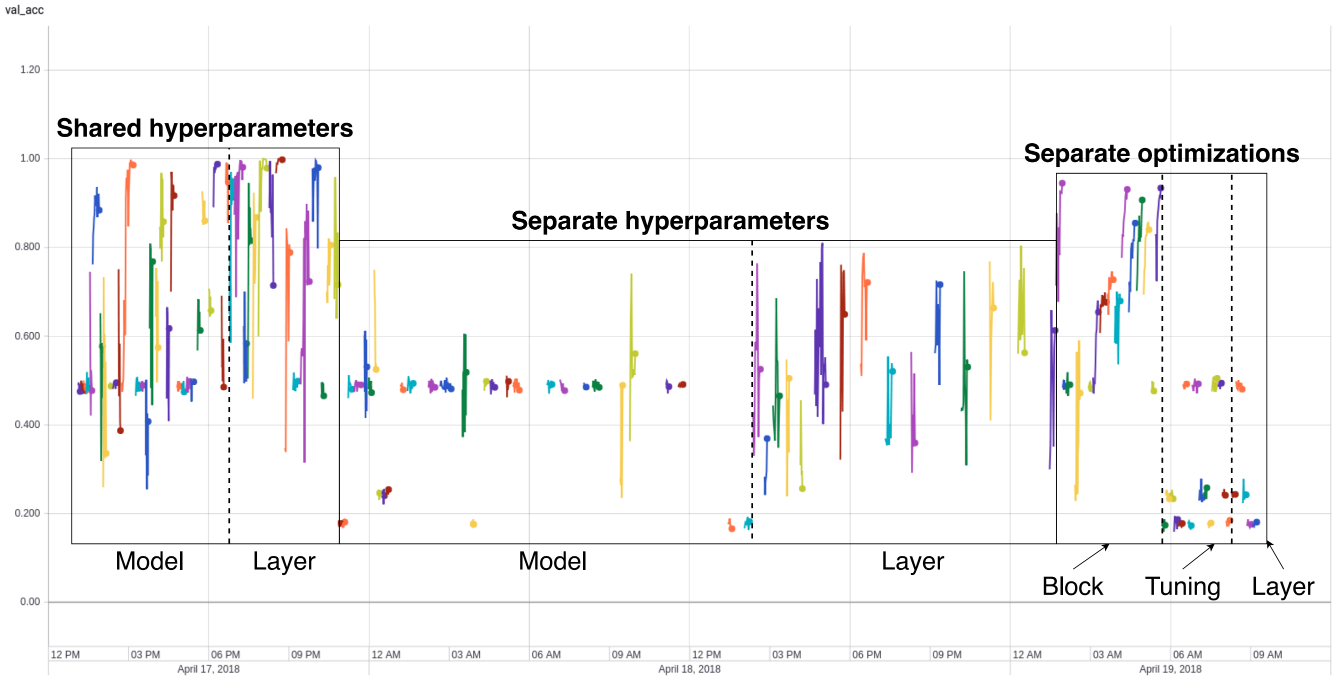


Figure 4.29: Plot of all training runs done on the Nerthus dataset for each time stamp. Each optimization strategy has been marked with black boxes and each optimization step within an optimization strategy has been separated by dotted lines.

achieves a very high score of 0.99, and the layer optimization manages to bump this up to 1. This is similar to how the model optimization for the same strategy on the Kvasir dataset reached 0.86 validation accuracy, and the layer optimization bumped it up to 0.89.

2. The separate hyperparameter optimization strategy fails to converge. The model optimization barely improves the validation accuracy above 0.5, just as it failed to improve it when running on Kvasir. However, compared to Kvasir, it did worse in the model optimization, and the layer optimization barely improved the validation accuracy.

3. The separate optimizations optimization strategy did much better for Nerthus than it did for the Kvasir dataset, but only for the classification block optimization. In the fine-tuning optimization, the optimization failed completely and produced validation accuracy that was worse than random classification along with validation accuracy around 0.5, which seemed to be where most validation accuracy that failed to converge where centered. The layer optimization failed to make the results better, which points to the default delimiting layer not being the problem, but rather the other hyperparameters.

Table 4.12: Table of hyperparameter values of the best trained model on the Nerthus dataset.

Hyperparameter	**Optimal value**
Pre-trained Model	Xception
Model Optimizer	SGD
Learning Rate	0.5495
Delimiting layer	0

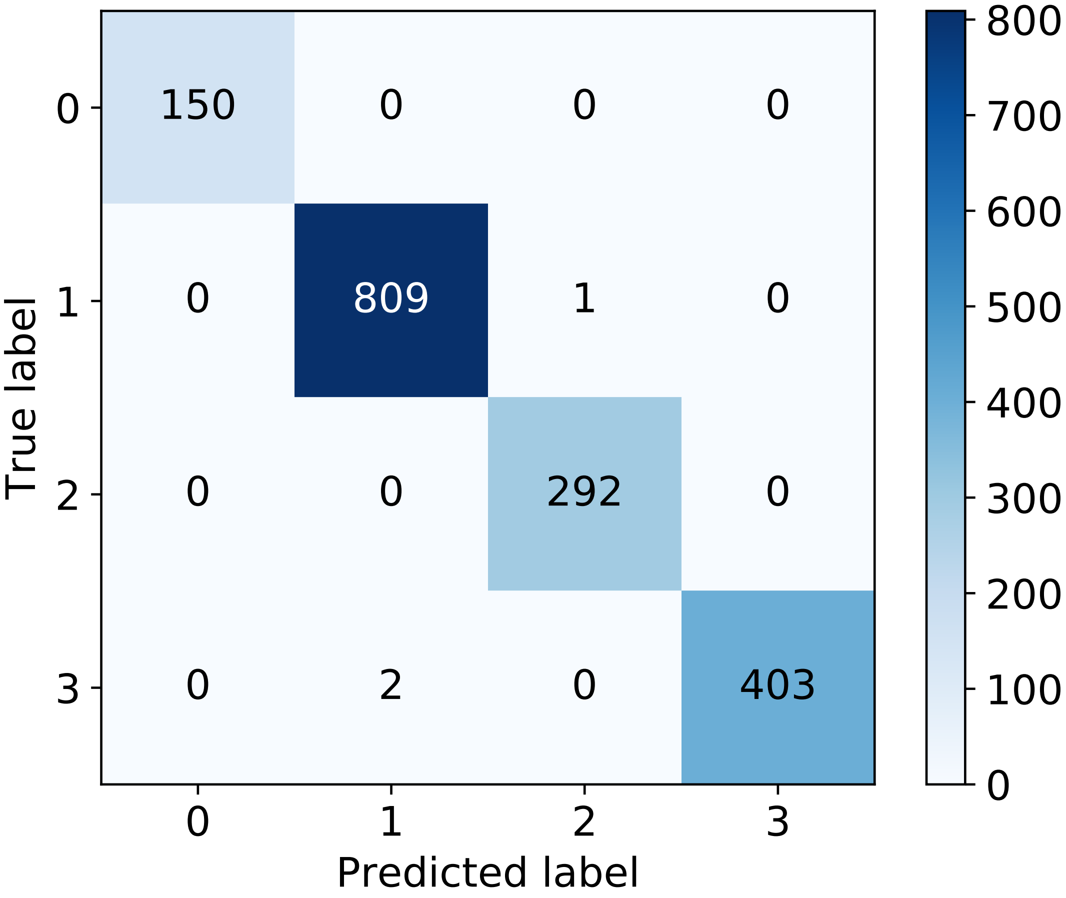
Class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
BBPS 0	0	0	1507	150	1.00	1.00	1.00	1.00	1.00	1.00
BBPS 1	1	2	845	809	1.00	1.00	1.00	1.00	1.00	1.00
BBPS 2	0	1	1364	292	1.00	1.00	1.00	1.00	1.00	1.00
BBPS 3	2	0	1252	403	1.00	1.00	1.00	1.00	1.00	1.00
Average	0.75	0.75	1242.00	413.50	1.00	1.00	1.00	1.00	1.00	1.00

Table 4.13: Metrics for each class after hyperparameter optimization for the best model on the Nerthus dataset, including average values for comparing with baseline metrics presented in the Nerthus paper [97].

4.4.1 Best Trained Model

The best-trained model in Nerthus achieved 1 in validation accuracy, which means the model was able to classify the validation dataset of Nerthus 100% correctly, or at least closer to 100% than 99%. However, there were more than one trained model to reach 1 in validation accuracy, so we chose the first one. When more than one model reaches 1 in validation accuracy, we see the drawback to optimizing using the validation accuracy instead of the validation loss. As the validation accuracy will be 1 even when a few images are classified wrongly because it is still closer to 1 than 0.9999, the granularity of the validation loss would help us find the best model among those with 1 in validation accuracy. Even though we here talk about minor differences, for the correct best model to be selected, future work should use validation loss over validation accuracy for the optimization.

From looking at table 4.12, we can see that the best delimiting layer is 0, which means all the layers in the fine-tuning step was trained. We can see from the metrics in figure 4.13, that the model is successful at classifying all except three images. Because we round to the closest digit, we get 1 in F_1 score, which is the highest possible. Figure 4.30 shows the confusion matrix, in which we can see that two images of 3 in BBPS score, which is the cleanest category, were set to 1 in BBPS score, which is the second dirtiest category. One image from the second dirtiest category was falsely categorized as have 2 in BBPS score.



	0	1	2	3
0	150	0	0	0
1	0	809	1	0
2	0	0	292	0
3	0	2	0	403

Figure 4.30: Confusion matrix for the best model trained on Nerthus. There are differing numbers of images per class, and the labels are the BBPS score. A higher BBPS score is cleaner than a lower BBPS score.

An almost perfect classification is tough to beat, and, indeed, we see in table 4.14, that the metrics from our best model is much better than the metrics produced in the Nerthus paper. The overall metrics are higher than those in the Kvasir paper, shown in table 4.11, meaning classification is overall easier for Nerthus than for Kvasir. Reaching better metrics for the best Nerthus model is, therefore, expected. From looking at table 4.14, we see that the best F_1 score was reached by the 6 global features logistic model tree, the same method that reached the best result in the Kvasir baseline, of 0.9. In comparison to our model, that is 0.1 validation accuracy, 10% percent, less.

Method	F_1	ACC	MCC	PREC	REC	SPEC
3 Layer CNN	0.742	0.772	0.621	0.811	0.694	0.937
6 Layer CNN	0.854	0.854	0.805	0.856	0.852	0.952
Inception v3 TFL	0.748	0.748	0.665	0.751	0.745	0.918
JCD Random Forest	0.790	0.861	0.679	0.805	0.794	0.870
2 GF Random Forrest	0.769	0.847	0.647	0.792	0.774	0.849
2 GF Logistic Model Tree	0.737	0.825	0.594	0.744	0.737	0.862
6 GF Random Forrest	0.860	0.913	0.801	0.885	0.866	0.895
6 GF Logistic Model Tree	0.899	0.949	0.863	0.901	0.901	0.960
Random/Majority	0.322	0.652	0.000	0.240	0.489	0.512

Table 4.14: Performance metrics for different methods of image classification done on the Nerthus dataset [97]. The methods and their metrics are taken from the Nerthus paper. TFL stands for transfer learning and GF stands for global features. JCD stands for joint composite descriptor and is a global color and texture feature. The metrics from JCD random forest and random/majority were used as the baseline for the Nerthus run, but we use all the methods as baseline and comparison to our optimized solution.

4.5 Summary

In summary, we have presented and discussed the experiments and the results of the thesis. We started by describing the design of the experiments. We presented the hardware we ran the experiments on and details on how they were conducted. After describing the design of the experiments, we presented the results from running the experiment on the Kvasir dataset. The results presented were from one full run where three different optimization strategies were tested. We split the presentation and discussion of the results into each of the different strategies. For each optimization strategy we went into detail of the results from each optimization step. The optimization strategies were:

- **Shared hyperparameters optimization strategy.** This strategy has one set of hyperparameters which it uses for both the classification block training and the fine-tuning in each Bayesian optimization iteration. The dimensionality of the search space is, therefore, three.

- **Separate hyperparameters optimization strategy.** This strategy has two sets of hyperparameters which it uses for both the classification block training and the fine-tuning in each Bayesian optimization iteration. The dimensionality of the search space is, therefore, five.

- **Separate optimizations optimization strategy.** This strategy sepa-

> rates the classification block training and fine-tuning into each their own optimization. The next optimization step uses the best model achieved from each optimization. The dimensionality of the classification block optimization is three, and the dimensionality of the fine-tuning is two.

Initially, we evaluated the whole run. From this evaluation, we found that the nonconvergence filtering was working as intended by terminating training runs that achieved bad results early. Additionally, we found that the shared hyperparameters optimization strategy achieved better validation accuracies than the other two optimization strategies. The shared hyperparameters optimization strategy achieved a validation accuracy of 0.89, which was much higher than the best of the separate hyperparameters optimization strategy at 0.63 validation accuracy and the best of the separate optimizations optimization strategy at 0.54 validation accuracy. Moreover, between most training runs the results would fluctuate significantly, and some training runs would train for many more epochs than the average training run would. Lastly, we found that the shared hyperparameters optimization took more time to finish than the other two optimization strategies.

We examined the shared hyperparameter optimization strategy results by themselves and found that, while many training runs were stopped early by nonconvergence filtering, the majority of training runs reached above the nonconvergence threshold. We found that the optimization strategy contained training runs where the validations accuracy would fluctuate heavily, take long to converge, reach validation accuracy comparable to a random classification of around 0.125, and reach the best validation accuracies over the whole test. By looking at the model optimization step, we found that the model optimization alone achieves a higher validation accuracy than the other optimization strategies of 0.86. However, the model optimization was very spiky in that it would in one step reach a high validation accuracy, but in the next step reach a low validation accuracy. From looking at the convergence plots and the table showing the hyperparameters and validation accuracy for each iteration, we saw that the best validation accuracy achieved happened in iteration four and that the Bayesian optimization would change the hyperparameters frequently between iterations after that, meaning it focused on exploring the search space.

The layer optimization of the shared hyperparameter optimization strategy showed us that it was successful in increasing the best validation accuracy from the model optimization from 0.86 to 0.89. The layer optimization ran for fifteen iterations and all the runs were from 0.73 to 0.89, which made the run the least fluctuating of all the optimization runs in the test. Some of the runs lasted up to 24 epochs, but the results were overall good and showed that the optimization was successful for the shared hyperparameter optimization strategy. The best run used InceptionResNetV2 as CNN model, Adadelta and 0.8417 learning rate for both classification block training and fine-tuning, and 136 as the delimiting layer.

The other optimization strategies could not compete with the shared hyperparameters optimization strategy in terms of attained validation accuracy. Upon closer examination of the lesser optimization strategies, we saw that both the separate hyperparameters optimization strategy and the separate optimizations optimization strategy failed to converge to higher validation accuracies and were most of the time terminated early by the nonconvergence filtering.

In the case of the separate hyperparameters optimization strategy, we found that the higher dimensionality of five made the optimization unable to converge, and all except a few training runs were stopped early by nonconvergence filtering. Additionally, the classification block training would reach higher validation accuracies than the fine-tuning. In the model optimization, the best run was one that reached 0.41, which is much lower than the shared hyperparameters model optimization best of 0.86. The validation accuracy of 0.41 stood alone, where all the other iterations reached below 0.19 validation accuracy and can be seen as the algorithm being lucky instead of actually converging on a good result. However, the optimization of the fine-tuning layer turned successful, and we saw an increase from 0.41 in the model optimization step to 0.61 in the layer optimization step.

In the case of the separate optimizations optimization strategy, we found that similarly to the separate hyperparameters strategy, the training runs were stopped by the nonconvergence filtering because of validation accuracies failing to reach the threshold of 0.5 validation accuracy. Where the separate hyperparameters strategy would have five training runs reach above the threshold, the separate optimizations strategy only had two runs pass the threshold. There were several similarities between the two strategies:

1. Both had higher classification block results than the fine-tuning results. In the case of separate optimizations, the fine-tuning optimization degraded the best classification block model from 0.47 to 0.29.

2. Both had most of their runs stopped early by nonconvergence filtering.

3. Both had layer optimization only running for seven iterations instead of the fifteen allowed.

4. Layer optimization increased the validation accuracy for both strategies, indicating that the default delimiting layer in the fine-tuning is a problem.

5. Regarding performance, both strategies failed compared to the shared hyperparameters optimization strategy, as both had best validation accuracies well below the shared hyperparameters strategy's best.

We presented the best-trained model from the Kvasir dataset experiment. The model was trained from the shared hyperparameters optimization strategy in the layer optimization step. The attained validation accuracy was 0.89. We calculated metrics suggested by the Kvasir paper, and in turn, compared the metrics to metrics given as baselines in the Kvasir paper. The best model from the optimization reached an F_1 score of 0.88, which was significantly higher than the best baseline F_1 score of 0.75, which was a method using handcrafted global image features. Compared to a transfer learning method using InceptionV3 achieving an F_1 score of 0.69, which did not include hyperprameter optimization, our results show that hyperparameter optimization makes a significant impact on the accuracy of the model.

Besides the Kvasir dataset experiment, we also presented an experiment conducted on the Nerthus dataset. We did not present the whole dataset as we saw many similarities to the Kvasir experiment, but we added the rest of the plots to appendix B. However, there were some distinctions such as the validation accuracies of training runs being generally higher for Nerthus than Kvasir, and the highest achieved validation accuracy being higher than for Kvasir. This behavior is expected as there are fewer classes to classify into. We saw from the plots we presented in the Nerthus section 4.4, that the shared hyperparameter optimization strategy was the clear winner among the strategies. Similarly to the Kvasir test, both the separate hyperparameters and separate optimizations optimization strategies failed.

The best-trained model from the Nerthus dataset reached a validation accuracy of 1, which means it classified the images from the validation set with almost 100% accuracy. As with the Kvasir experiment for the Kvasir dataset paper, we calculated the suggested metrics from the Nerthus dataset paper for the best model and compared them to the baseline methods presented in the Nerthus dataset paper. The metrics were all 1 for the best model, so they were impossible to beat without greater precision for the validation accuracy. The best method from the Nerthus paper used six handcrafted global image features and reached an F_1 score of 0.9, which is 10% less than the best model with an F_1 score of 1. When we compare the transfer learning method using Inception v3 from the Nerthus paper, with an F_1 score of 0.75, we see an even greater difference.

Chapter 5

Conclusion

5.1 Summary and Main Contributions

In this thesis, we presented our experiences with researching hyperparameter optimization in transfer learning. Transfer learning has risen in popularity the last years because it promises a solution for datasets insufficient for training deep neural networks from scratch because of size or variation. There are several research fields where acquiring enough data for training from scratch is challenging. One such field and the one we decided to focus on is the medical field, and more specifically, gastroenterology. Technology allows doctors to use sensors and multimedia equipment for diagnosis. However, the diagnosis is heavily dependant on the capabilities of the doctor. Deep convolutional neural networks are used in computer-aided detection systems to help reduce the number of missed diseases or abnormalities. Those computer-aided detection systems we discovered utilize or examine in many cases transfer learning. Transfer learning has many hyperparameters, and in most cases, these hyperparameters are not automatically optimized. This thesis explored the benefits of automatically optimizing the hyperparameters for transfer learning and gave answers to the questions posed in section 1.2 in the introduction.

We chose a method for transfer learning fitting to our use case of image classification and analysis in medical image datasets. The gist of the technique was to replace the classification block of a pre-trained deep convolutional neural network model and only train the classification block on the dataset. After training, the next step was to fine-tune the model, meaning we took the best performing classification block model, selected a layer number and trained all layers after that number only. Which layer number to choose is not a trivial matter, so we included this delimiting layer as a hyperparameter. Moreover, we decided to use the pre-trained deep convolutional neural network mode, the gradient descent optimizing function and the learning rate as hyperparameters targeted for automatic optimization.

To perform the optimization, we selected an optimization technique called Bayesian optimization. We chose this technique over methods such as random and grid-search optimization because recent research has shown

Bayesian optimization to perform better. Bayesian optimization creates a surrogate model and sequentially runs experiments which tunes the surrogate model. An acquisition function chooses where hyperparameter values of the next iteration are selected from in search space. However, Bayesian optimization is challenging to use, as it, like the machine learning models, has options we can consider hyperparameters. Optimizing the configuration of the Bayesian optimization would be out of the scope of this thesis, so we are using standard Bayesian optimization with default parameters.

We defined the hyperparameter bounds which the Bayesian optimization will take hyperparameter values from before initiating the optimization. The optimization then chose for each iteration hyperparameters for testing. However, we found that there are at least three different strategies for optimizing the hyperparameters:

1. **The shared hyperparameters optimization strategy.** In this strategy, we had three hyperparameters: The pre-trained model, the optimizing function, and the learning rate. The same optimizing function and the learning rate was used for both classification block training and the fine-tuning. The dimensionality in the search space was then three.

2. **The separate hyperparameters optimization strategy.** This strategy was similar to the shared hyperparameter strategy, except we used separate optimization functions and learning rates for the two training runs. The dimensionality was, therefore, five for the search space.

3. **The separate optimizations optimization strategy.** This strategy was not similar to the others. Instead of having one optimization run with a large set of hyperparameters, this strategy split the optimization into two and divided the hyperparameters. The classification block was then optimized separately and had three hyperparameters: The pre-trained model, the optimization function, and the learning rate. The dimensionality was then 3. The fine-tuning optimization used the best model from the classification block optimization and only had two hyperparameters: The optimization function and the learning rate.

The delimiting layer hyperparameter was separately optimized after an optimization strategy finished and the dimensionality of the optimization was one. We based the choice of separating the delimiting layer from the other hyperparameters because we wanted to reduce the dimensionality of the optimizations and saw an opportunity in the delimiting layer since it did not impact the other hyperparameters. However, this meant we had to use a default delimiting layer for the fine-tuning steps in the optimization strategies. We did not recognize before later in the thesis that this delimiting layer posed a problem for how the optimizer chose the pre-trained model. The issue was that some models could produce better results initially with the default delimiting layer than other models. The issue meant that some models would potentially never win the optimization because they were not as compatible with the default delimiting layer as others. We could not see that this happened in our results, but the theory was against us, so we urge other researchers to either include the pre-trained model and the delimiting layer in the same step or merely exclude the pre-trained model from the hyperparameter set and do optimizations for each model instead.

We ran experiments for two medical image datasets called Kvasir and Nerthus. Both datasets showed frames from the gastrointestinal tract. Kvasir was an eight-class dataset containing 8000 images with diseases, polyps, medical procedures, and anatomical landmarks. Nerthus was a four-class dataset containing 5525 images of different cleanliness scores in the bowel. We ran these experiments on a system we designed to run full tests using all of the optimization strategies automatically. The proposed system used Keras and TensorFlow for the machine learning capabilities and GPyOpt for the Bayesian optimization. We designed it to be configurable, and it used a technique called nonconvergence filtering which could be disabled. Nonconvergence filtering was a technique presented in this thesis which stops a training run early if it does not pass a given threshold in a given number of epochs. The goal of the technique was to avoid having runs train for a long time when they could not compete with other training runs that did pass the given threshold. Automatic optimization with Bayesian optimization tried many hopeless hyperparameters, so we saved a lot of time by doing this.

We presented two sets of results in this thesis, one from Kvasir and one from Nerthus. We examined the results of Kvasir in-depth and presented only the most important observations from the Nerthus dataset as we could observe much of the same behavior between them. For both experiments, we saw that the shared hyperparameters optimization was the best optimization strategy, achieving a validation accuracy of 0.89 and F_1 score of 0.88 on Kvasir and 1.0 in both validation accuracy and F_1 score on Nerthus. We compared the results to results given as baselines by the respective dataset papers and found that our best model for Kvasir had an F_1 score that was 0.13 better than the best baseline method, and that the best model for Nerthus had an F_1 score that was 0.10 better than the best baseline method. We achieved these results after layer optimization, but even without layer optimization, we achieved a validation accuracy of 0.86 for Kvasir and 0.99 for Nerthus. We found that the hyperparameter optimization done in the optimization strategies had a greater impact on the results than the layer optimization done after. However, the layer optimization succeeded in nudging the results even higher. We also saw that the delimiting layer chosen by the algorithm was nontrivial to select manually.

The other two strategies failed to converge, and the reason was most likely due to the higher number of hyperparameters to search through, and that different hyperparameter between the classification block training and fine-tuning rarely worked in our results.

In conclusion, we have answered the research questions in section 1.2:

1. *Can hyperparameters for transfer learning be optimized automatically? If so, how does it influence the performance and what should be optimized?* We have shown results for two small medical image datasets using four hyperparameters and three different optimization strategies. In the results, we saw that the performance increased for one of the strategies over ten percent past comparable methods from the dataset papers. From the results, we found a flaw in having the delimiting layer and the pre-trained model in different hyperparameter optimizations, so we suggested removing the pre-trained model as a hyperparameter in future work.
2. *How should a system be built that automatically can fulfill the task of automated hyperparameter optimization for transfer learning?* We proposed and developed a system for running automatic experiments with Bayesian optimization on hyperparameters with a transfer learning approach. The system is configurable and runs three different hyperparameter optimization strategies. For each optimization, graphs and summaries for each optimization strategy are saved to disk.

We conclude that automatic hyperparameter optimization is an effective strategy for increasing performance in transfer learning use cases and that automatically adjusting the delimiting layer reveals layers that are nontrivial to select manually.

5.2 Future Work

For future work, researchers can further improve the way we optimize the transfer learning method. Other optimization methods than Bayesian optimization could be tested, and other methods that are building on Bayesian optimization might give better results than using the standard Bayesian optimization algorithm. The standard Bayesian optimization we use in this thesis could be improved by tuning the options available to us, such as changing the surrogate model option, the acquisition function, the acquisition function optimizer, the number of data points collected initially, the batch size, the type of evaluation, and more. These options is one of the drawbacks of Bayesian optimization, as they can also be considered hyperparameters, which only increases the total amount of hyperparameters to optimize. However, optimization of the Bayesian optimization hyperparameters can be done manually based on observations from the results in this thesis and further future experiments. For example, we saw in the results in several of the surrogate model and acquisition plots that much of the search space is omitted because the acquisition function relies on more iterations, both initial, randomly distributed among the search space, and maximum allowed. We advise future work to, therefore, increase the number of iterations if possible.

Future work should learn from the methods we found did not work in this thesis. Those methods being the default delimiting layer and the separate hyperparameters and separate optimizations optimization strategies.

The default delimiting layer introduces a bias in the shared and separate hyperparameters optimization strategies for those pre-trained CNN models which performs better with the default delimiting layer than others. Future work should consider solutions which remove any dependencies between a model and the default delimiting layer, such as for example removing the pre-trained model as a hyperparameter and run full optimizations for each model and later compare those to find the best one.

The separate hyperparameters and separate optimizations optimization strategies failed to achieve results that could compete with the shared hyperparameters optimization strategy for both the Kvasir and Nerthus datasets. Future work should conduct more tests to find if there is an underlying issue, such as different hyperparameters not working for the two steps, or if there are ways to improve the optimization to make the strategies viable.

Finally, if researchers can make the optimization perform better and more reliable, future work can also include optimization of other hyperparameters, such as the batch size, different classification blocks, and gradient descent parameters such as momentum.

5.3 Final Remarks

We relied on external tools for the majority of our plots. For the plots showing the validation accuracy of training runs, we use TensorBoard, and for the plots showing the convergence and surrogate model with the acquisition function, we used GPyOpt. To make the plots clearer, we should have employed our own solutions for the plotting. Aside from this, even though splitting up the model optimization and the layer optimization resulting in the use of a default delimiting layer was flawed, we managed to show this flaw and be successful at optimizing the transfer learning method we used. This work can be used for further work into transfer learning hyperparameter optimization, or be used as is in current computer-aided diagnosis systems to increase their detection accuracy. By showing that hyperparameters can be models and optimizers and that optimizing the delimiting layer is important to improve performance, we hope to offer a valuable contribution to the research into transfer learning, but more importantly, into medical multimedia systems that in the future can save lives by detecting some of the most lethal diseases affecting mankind.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] 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. 2016.

[3] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A Livezey,

Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A {Python} framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.0, 2016.

[4] Alex Alemi. Improving Inception and Image Classification in TensorFlow, 2016.

[5] Md Zahangir Alom, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Mahmudul Hasan, Brian C Van Esesn, Abdul A S. Awwal, and Vijayan K. Asari. The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches. *arXiv*, mar 2018.

[6] The GPyOpt authors. GPyOpt: A bayesian optimization framework in python. `http://github.com/SheffieldML/GPyOpt`, 2016.

[7] Robert L. Barclay, Joseph J. Vicari, Andrea S. Doughty, John F. Johanson, and Roger L. Greenlaw. Colonoscopic Withdrawal Times and Adenoma Detection during Screening Colonoscopy. *New England Journal of Medicine*, 355(24):2533–2541, dec 2006.

[8] Paul Barrett. Euclidean Distance: raw, normalized, and double-scaled coefficients. *pbarret.net*, 2005.

[9] James O. Berger. *Statistical Decision Theory and Bayesian Analysis*. Springer Science & Business Media, 2 edition, 2013.

[10] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2546–2554, 2011.

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

[12] J. Bond. Colon Polyps and Cancer. *Endoscopy*, 37(03):208–212, feb 2005.

[13] Léon Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proc. of COMPSTAT'2010*, pages 177–186, Heidelberg, 2010. Physica-Verlag HD.

[14] Eric Brochu, Vlad M. Cora, and N De Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *ArXiv*, page 49, dec 2010.

[15] Lynn Butterly, Christina M Robinson, Joseph C Anderson, Julia E Weiss, Martha Goodrich, Tracy L Onega, Christopher I Amos, and Michael L Beach. Serrated and Adenomatous Polyp Detection Increases With Longer Withdrawal Time: Results From the New Hampshire Colonoscopy Registry. *The American Journal of Gastroenterology*, 109(3):417–426, mar 2014.

[16] World Health Organization International Agency for Research on Cancer Cancer Research UK. World cancer factsheet, 2014. `http://www.cancerresearchuk.org/sites/default/files/cs_report_world.pdf` Accessed: 2017-11-02.

[17] Rich Caruana, Steve Lawrence, and Lee Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *the 13th International Conference on Neural Information Processing Systems*, pages 402–408, 2000.

[18] Ronald A Castellino. Computer aided detection (CAD): an overview. *Cancer imaging : the official publication of the International Cancer Imaging Society*, 5(1):17–9, aug 2005.

[19] American Cancer Society Cancer Statistics Center. 5-year relative survival, 2006-2012, colorectum, by stage at diagnosis, 2016. `https://cancerstatisticscenter.cancer.org/cancer-site/Colorectum/i5NQiDbI` Accessed: 2017-11-06.

[20] Souad Chaabouni, Jenny Benois-Pineau, and Chokri Ben Amar. Transfer learning with deep networks for saliency prediction in natural video. In *Proc. of ICIP*, volume 2016-Augus, pages 1604–1608. IEEE, sep 2016.

[21] Shawn C. Chen and Douglas K. Rex. Endoscopist Can Be More Powerful than Age and Male Gender in Predicting Adenoma Detection at Colonoscopy. *The American Journal of Gastroenterology*, 102(4):856–861, apr 2007.

[22] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, 2014.

[23] François Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. *CoRR*, abs/1610.02357, 2016.

[24] François Chollet and Collaborators. Applications - Keras Documentation, 2018.

[25] François Chollet and et al. Keras, 2015. `https://keras.io`. Accessed: 2018-05-30.

[26] Marc Claesen and Bart De Moor. Hyperparameter Search in Machine Learning. 2015.

[27] Philomena M Colucci, Steven H Yale, and Christopher J Rall. Colorectal polyps. *Clinical medicine & research*, 1(3):261–2, jul 2003.

[28] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, January 1989.

[29] TensorBoard Contributors. TensorBoard - TensorFlow's Vizualisation Toolkit, 2018.

[30] Tensorflow Contributors. Tensorflow GitHub Repository, 2018.

[31] George Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Math. Control Signals Systems*, 2:303–314, 1989.

[32] Dallas Gastroenterologist. What is a Colonoscopy Procedure and How to Prepare for it?, 2018.

[33] Li Deng and Dong Yu. Deep Learning: Methods and Applications. *Foundations and Trends® in Signal Processing*, 7(3-4):197–387, may 2014.

[34] Pedro Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, October 2012.

[35] Timothy Dozat. Incorporating Nesterov Momentum into Adam. *ICLR Workshop*, (1):2013–2016, 2016.

[36] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[37] Mark Ebden. Gaussian Processes: A Quick Introduction. may 2015.

[38] Katharina Eggensperger, Matthias Feurer, and Frank Hutter. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. *NIPS, BayesOpt workshop*, pages 1–5, 2013.

[39] Jacques Ferlay, Isabelle Soerjomataram, Rajesh Dikshit, Sultan Eser, Colin Mathers, Marise Rebelo, Donald Maxwell Parkin, David Forman, and Freddie Bray. Cancer incidence and mortality worldwide: Sources, methods and major patterns in GLOBOCAN 2012. *International Journal of Cancer*, 136(5):E359–E386, mar 2015.

[40] François Chollet. Building powerful image classification models using very little data, 2016.

[41] Iris Fu and Cambron Carter. Benchmarking Training Time for CNN-based Detectors with Apache MXNet, 2017.

[42] N. M. Gatto, H. Frucht, V. Sundararajan, J. S. Jacobson, V. R. Grann, and A. I. Neugut. Risk of Perforation After Colonoscopy and Sigmoidoscopy: A Population-Based Study. *JNCI Journal of the National Cancer Institute*, 95(3):230–236, feb 2003.

[43] Dave Gershgorn. ImageNet: the data that spawned the current AI boom. *Quartz*, 2017.

[44] B. Giritharan, Xiaohui Yuan, Jianguo Liu, B. Buckles, JungHwan Oh, and Shou Jiang Tang. Bleeding detection from capsule endoscopy videos. In *Proc. of EMBS*, 2008.

[45] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google Vizier: A Service for Black-Box Optimization.

[46] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google Vizier: A Service for Black-Box Optimization. 2017.

[47] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[48] GPy. GPy: A Gaussian process framework in python. `http://github.com/SheffieldML/GPy`.

[49] Kevin Gurney. *An introduction to neural networks*. UCL Press, 2004 edition, 1997.

[50] Hayashi N. et. al. Endoscopic prediction of deep submucosal invasive carcinoma: validation of the narrow-band imaging international colorectal endoscopic (nice) classification. *Gastrointest Endosc*, 2013.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition.

[52] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015.

[53] Jeremy T Hetzel, Christopher S Huang, Jennifer A Coukos, Kelsey Omstead, Sandra R Cerda, Shi Yang, Michael J O'Brien, and Francis A Farraye. Variation in the Detection of Serrated Polyps in an Average Risk Colorectal Cancer Screening Cohort. *The American Journal of Gastroenterology*, 105(12):2656–2664, dec 2010.

[54] Geoffrey E. Hinton, Nitish Srivastava, and Kevin Swersky. Lecture 6a- overview of mini-batch gradient descent. *COURSERA: Neural Networks for Machine Learning*, page 31, 2012.

[55] O. Holme, M. Bretthauer, A. Fretheim, J. Odgaard-Jensen, and G. Hoff. Flexible sigmoidoscopy versus faecal occult blood testing for colorectal cancer screening in asymptomatic individuals. *Cochrane Database of SR*, 2013.

[56] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *ArXiv*, page 9, 2017.

[57] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely Connected Convolutional Networks.

[58] Gao Huang, Zhuang Liu, and Kilian Q Weinberger. Densely Connected Convolutional Networks. *CoRR*, abs/1608.06993, 2016.

[59] IARC Working Group. Cancer Screening in Report on the implementation of the Council Recommendation on cancer screening. 2017.

[60] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[61] Tinu Theckel Joy, Santu Rana, Sunil Gupta, and Svetha Venkatesh. Hyperparameter tuning for big data using Bayesian optimisation. In *Proc. of Pattern Recognition*, pages 2574–2579. IEEE, dec 2017.

[62] Janusz Kacprzyk and Witold Pedrycz. *Springer handbook of computational intelligence*. Springer, 2015.

[63] Charles J. Kahi, David G. Hewett, Dustin Lee Norton, George J. Eckert, and Douglas K. Rex. Prevalence and Variable Detection of Proximal Colon Serrated Polyps During Screening Colonoscopy. *Clinical Gastroenterology and Hepatology*, 9(1):42–46, jan 2011.

[64] M. F. Kaminski, J. Regula, E. Kraszewska, M. Polkowski, U. Wojciechowska, J. Didkowska, M. Zwierko, M. Rupinski, M. P. Nowacki, and E. Butruk. Quality indicators for colonoscopy and the risk of interval cancer. *New England Journal of Medicine*, 362(19):1795–1803, 2010.

[65] Kaminski et. al. Leadership training to improve adenoma detection rate in screening colonoscopy: a randomised trial. *Gut*, 2015.

[66] Kaminski MF et. al. Increased rate of adenoma detection associates with reduced risk of colorectal cancer and death. *Gastroenterology*, 2017.

[67] Keras. Keras Callbacks Source Code. `https://github.com/keras-team/keras/blob/master/keras/callbacks.py`, 2018.

[68] Diederik P Kingma and Jimmy Ba. Adam: {A} Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[69] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICML*, 2015.

[70] Greg Kochanski, Daniel Golovin, John Karro, Benjamin Solnik, Subhodeep Moitra, and D Sculley. Bayesian Optimization for a Better Dessert. In *Proc. of NIPS Workshop on Bayesian Optimization*, number Nips, 2017.

[71] David Kriesel. A brief introduction to neural networks. `http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-2col-dkrieselcom.pdf`, 2007.

[72] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, pages 1–9, 2012.

[73] Edwin J Lai, Audrey H Calderwood, Gheorghe Doros, Oren K Fix, and Brian C Jacobson. The Boston bowel preparation scale: a valid and reliable instrument for colonoscopy-oriented research. *Gastrointestinal endoscopy*, 69(3 Pt 2):620–5, mar 2009.

[74] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1:541–551, 1989.

[75] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer Feedforward Networks With a Nonpolynomial Activation Function Can Approximate Any Function. *Neural Networks*, 6:861–867, 1993.

[76] Julien Charles Levesque, Audrey Durand, Christian Gagne, and Robert Sabourin. Bayesian optimization for conditional hyperparameter spaces. In *Proc. of NN*, volume 2017-May, pages 286–293. IEEE, may 2017.

[77] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. 2016.

[78] Xiangang Li and Xihong Wu. Constructing Long Short-Term Memory based Deep Recurrent Neural Networks for Large Vocabulary Speech Recognition. *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4520–4524, 2014.

[79] Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network. *arXiv preprint*, page 10, 2013.

[80] Mathias Lux and Savvas A. Chatzichristofis. Lire: Lucene image retrieval: An extensible java cbir library. In *Proc. of ACM MM*, pages 1085–1088, 2008.

[81] Mathias Lux, Michael Riegler, Pål Halvorsen, Konstantin Pogorelov, and Nektarios Anagnostopoulos. LIRE: Open Source Visual Information Retrieval. In *Proc. of MMSys*, MMSys '16, pages 30:1—-30:4, New York, NY, USA, 2016. ACM.

[82] Shawn Mallery and Jacques Van Dam. Advances in diagnostic and therapeutic endoscopy. *Medical Clinics of North America*, 84(5):1059–1083, 2000.

[83] Marcin Marszalek, Ivan Laptev, and Cordelia Schmid. Actions in Context. (i):2929–2936, 2010.

[84] Ruben Martinez-Cantin, Kevin Tee, and Michael McCourt. Practical Bayesian optimization in the presence of outliers. dec 2017.

[85] Masakazu Matsugu, Katsuhiko Mori, Yusuke Mitari, and Yuji Kaneda. Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural networks : the official journal of the International Neural Network Society*, 16(5-6):555–9, 2003.

[86] Vinod Nair and Geoffrey E Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. *Proc. of ML*, (3):807–814, 2010.

[87] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.

[88] Travis E. Oliphant. *A guide to NumPy*. Trelgol Publishing, 2006.

[89] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and Transferring Mid-level Image Representations Using Convolutional Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1717–1724. IEEE, jun 2014.

[90] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning, oct 2010.

[91] PDQ Screening and Prevention Editorial Board. *Colorectal Cancer Screening (PDQ®): Health Professional Version*, volume 103. 2002.

[92] Anne F Peery, Evan S Dellon, Jennifer Lund, Seth D Crockett, Christopher E McGowan, William J Bulsiewicz, Lisa M Gangarosa, Michelle T Thiny, Karyn Stizenberg, Douglas R Morgan, Yehuda Ringel, Hannah P Kim, Marco Dacosta Dibonaventura, Charlotte F Carroll, Jeffery K Allen, Suzanne F Cook, Robert S Sandler, Michael D Kappelman, and Nicholas J Shaheen. Burden of gastrointestinal disease in the United States: 2012 update. *Gastroenterology*, 143(5):1179–87.e1–3, nov 2012.

[93] Tomaso Poggio and Fabio Anselmi. *Visual Cortex and Deep networks: Learning Invariant Representations*. 2016.

[94] Konstantin Pogorelov, Baerum Hospital, Norway Thomas de Lange, Carsten Griwodz, Kristin Ranheim Randel, Håkon Kvale Stensland, Duc-Tien Dang-Nguyen, Concetto Spampinato, Dag Johansen, Michael Riegler, Pål Halvorsen, Sigrun Losada Eskeland, and Thomas de Lange. A Holistic Multimedia System for Gastrointestinal Tract Disease Detection Sigrun Losada Eskeland ACM Reference format.

[95] Konstantin Pogorelov, Sigrun Losada, Carsten Griwodz, Thomas de Lange, Kristin Ranheim Randel, Duc Tien Dang Nguyen, Håkon Kvale Stensland, Francesco De Natale, Dag Johansen, Michael Riegler, and Pål Halvorsen. A holistic multimedia system for gastrointestinal tract disease detection. In *Proc. of MMSys*, 2017.

[96] Konstantin Pogorelov, Olga Ostroukhova, Andreas Petlund, Pål Halvorsen, Thomas De Lange, Håvard Nygaard Espeland, Tomas Kupka, Carsten Griwodz, and Michael Riegler. Deep Learning and Handcrafted Feature Based Approaches for Automatic Detection of Angiectasia. In *Proc. of IEEE BHI*, 2018.

[97] Konstantin Pogorelov, Kristin Ranheim Randel, Thomas de Lange, Sigrun Losada Eskeland, Carsten Griwodz, Dag Johansen, Concetto Spampinato, Mario Taschwer, Mathias Lux, Peter Thelin Schmidt, Michael Riegler, and Pål Halvorsen. Nerthus: A bowel preparation quality video dataset. In *Proc. of ACM MMSys Conference*, MMSys'17, pages 170–174, New York, NY, USA, 2017. ACM.

[98] Konstantin Pogorelov, Kristin Ranheim Randel, Carsten Griwodz, Sigrun Losada Eskeland, Thomas de Lange, Dag Johansen, Concetto Spampinato, Duc-Tien Dang-Nguyen, Mathias Lux, Peter Thelin Schmidt, Michael Riegler, and Pål Halvorsen. Kvasir: A multi-class image dataset for computer aided gastrointestinal disease detection. In *Proc. of ACM MMSys*, MMSys'17, pages 164–169, New York, NY, USA, 2017. ACM.

[99] Konstantin Pogorelov, Michael Riegler, Sigrun Losada Eskeland, Thomas de Lange, Dag Johansen, Carsten Griwodz, Peter Thelin Schmidt, and Pål Halvorsen. Efficient disease detection in gastrointestinal videos – global features versus neural networks. *Multimedia Tools and Applications*, 76(21):22493–22525, 2017.

[100] Konstantin Pogorelov, Michael Riegler, Pal Halvorsen, Peter Thelin Schmidt, Carsten Griwodz, Dag Johansen, Sigrun Losada Eskeland, and Thomas de Lange. GPU-Accelerated Real-Time Gastrointestinal Diseases Detection. In *2016 IEEE 29th International Symposium on Computer-Based Medical Systems (CBMS)*, pages 185–190. IEEE, jun 2016.

[101] Chao Qin, Diego Klabjan, and Daniel Russo. Improving the Expected Improvement Algorithm. *CoRR*, abs/1705.1, 2017.

[102] Stuart H Ralston, Ian Penman, Mark Strachan, and Richard Hobson. *Davidson's Principles and Practice of Medicine E-Book*. 2014.

[103] Waseem Rawat and Zenghui Wang. Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review. *Neural Computation*, 29(9):2352–2449, 2017.

[104] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. CNN features off-the-shelf: An astounding baseline for recognition. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 512–519, 2014.

[105] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the Convergence of Adam and Beyond. In *International Conference on Learning Representations*, 2018.

[106] C.J. Rees, Roisin Bevan, Katharina Zimmermann-Fraedrich, M.D. Rutter, Douglas Rex, Evelien Dekker, Thierry Ponchon, Michael Bretthauer, Jaroslaw Regula, Brian Saunders, Cesare Hassan, M.J. Bourke, and T. Rösch. Expert opinions and scientific evidence for colonoscopy key performance indicators. *Gut*, 65(12), 2016.

[107] Douglas K Rex, C. Richard Boland, Jason A Dominitz, Francis M Giardiello, David A Johnson, Tonya Kaltenbach, Theodore R Levin, David Lieberman, and Douglas J Robertson. Colorectal Cancer Screening: Recommendations for Physicians and Patients from the U.S. Multi-Society Task Force on Colorectal Cancer, 2017.

[108] Eduardo Ribeiro, Andreas Uhl, Georg Wimmer, and Michael Häfner. Exploring Deep Learning and Transfer Learning for Colonic Polyp Classification. *Computational and Mathematical Methods in Medicine*, 2016:6584725, 2016.

[109] Michael Riegler, Martha Larson, Mathias Lux, and Christoph Kofler. How 'how' reflects what's what: Content-based exploitation of how users frame social images. In *Proc. of ACM MM*, pages 397–406, 2014.

[110] Michael Riegler, Mathias Lux, Carsten Griwodz, Concetto Spampinato, Thomas de Lange, Sigrun L. Eskeland, Konstantin Pogorelov, Wallapak Tavanapong, Peter T. Schmidt, Cathal Gurrin, Dag Johansen, Håvard Johansen, and Pål Halvorsen. Multimedia and medicine: Teammates for better disease detection and survival. In *Proc. of ACM MM*, pages 968–977, 2016.

[111] Michael Riegler, Konstantin Pogorelov, Sigrun Losada Eskeland, Peter Thelin Schmidt, Zeno Albisser, Dag Johansen, Carsten Griwodz, Pål Halvorsen, and Thomas De Lange. From Annotation to Computer-Aided Diagnosis. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 13(3):1–26, may 2017.

[112] Michael Riegler, Konstantin Pogorelov, Sigrun Losada Eskeland, Peter Thelin Schmidt, Zeno Albisser, Dag Johansen, Carsten Griwodz, Pål Halvorsen, and Thomas De Lange. From annotation to computer-aided diagnosis: Detailed evaluation of a medical multimedia system. *ACM Trans. Multimedia Comput. Commun. Appl.*, 13(3):26:1–26:26, May 2017.

[113] Michael Riegler, Konstantin Pogorelov, Pål Halvorsen, Thomas de Lange, Carsten Griwodz, Peter Thelin Schmidt, Sigrun Losada Eskeland, and Dag Johansen. Eir - efficient computer aided diagnosis framework for gastrointestinal endoscopies. In *Proc. of CBMI*, 2016.

[114] Alaa Rostom and Emilie Jolicoeur. Validation of a new scale for the assessment of bowel preparation quality. *Gastrointestinal endoscopy*, 59(4):482–6, apr 2004.

[115] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, sep 2016.

[116] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, dec 2015.

[117] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Third edit edition, 2010.

[118] Dennis Salguero. Keras API Proposal - EarlyBaselineStopping, 2018.

[119] Dennis Salguero. Pull Request: ”New Callback: EarlyBaselineStopping”, 2018.

[120] A. L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3):210–229, jul 1959.

[121] Klaus Schoeffmann, Bernd Münzer, Michael Riegler, and Pål Halvorsen. Medical Multimedia Information Systems (MMIS). In *Proc. of ACM MM*, MM '17, pages 1957–1958, New York, NY, USA, 2017. ACM.

[122] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proc. of ACM SIGKDD KDD*, KDD '16, page 2135, New York, NY, USA, 2016. ACM.

[123] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

[124] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations (ICRL)*, pages 1–14, 2015.

[125] P. Patrick Van Der Smagt, P. Patrick Van Der Smagt, Ben J. A. Kröse, Ben J. A. Krose, and P. Patrick Smagt. An introduction to Neural Networks. 1993.

[126] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. *Adv. Neural Inf. Process. Syst. 25*, pages 1–9, 2012.

[127] Jasper Snoek, Oren Rippel, and Ryan P Adams. Scalable Bayesian Optimization Using Deep Neural Networks. In *Proc. of ML*, 2015.

[128] American Cancer Society. American cancer society recommendations for colorectal cancer early detection, 2017. `https://www.cancer.org/cancer/colon-rectal-cancer/detection-diagnosis-staging/acs-recommendations.html` Accessed: 2017-11-06.

[129] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for Simplicity: The All Convolutional Net. 2014.

[130] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, (2010):8609–8613, 2013.

[131] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-Task Bayesian Optimization. In *Proc. of NIPS*, pages 2004–2012, 2013.

[132] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *CoRR*, abs/1602.07261, 2016.

[133] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. 2015.

[134] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *CoRR*, abs/1512.00567, 2015.

[135] Nima Tajbakhsh, Jae Y Shin, Suryakanth R Gurudu, R Todd Hurst, Christopher B Kendall, Michael B Gotway, and Jianming Liang. Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning? *IEEE Transactions on Medical Imaging*, 35(5):1299–1312, 2016.

[136] TensorFlow. TensorFlow, 2018. `https://www.tensorflow.org/`. Accessed: 2018-05-01.

[137] Vestre Viken Hospital Trust. Vestre Viken Hospital Trust, 2018. `https://vestreviken.no/vestre-viken-hospital-trust`. Accessed 2018-05-07.

[138] L. von Karsa, J. Patnick, and N. Segnan. European guidelines for quality assurance in colorectal cancer screening and diagnosis. first edition–executive summary. *Endoscopy*, 44 Suppl 3:SE1–8, 2012.

[139] Yi Wang, Wallapak Tavanapong, Johnny Wong, Jung Hwan Oh, and Piet C. de Groen. Polyp-Alert: Near real-time feedback during colonoscopy. *Computer Methods and Programs in Biomedicine*, 120(3):164–179, jul 2015.

[140] Karl Weiss, Taghi M. Khoshgoftaar, and Ding Ding Wang. A survey of transfer learning. *Journal of Big Data*, 3(1):9, dec 2016.

[141] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. The WEKA Workbench. In *Data Mining: Practical Machine Learning Tools and Techniques*, chapter Online App. Morgan Kaufmann, fourth edi edition, 2016.

[142] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On Early Stopping In Gradient Descent Learning. *MIT*, 2005.

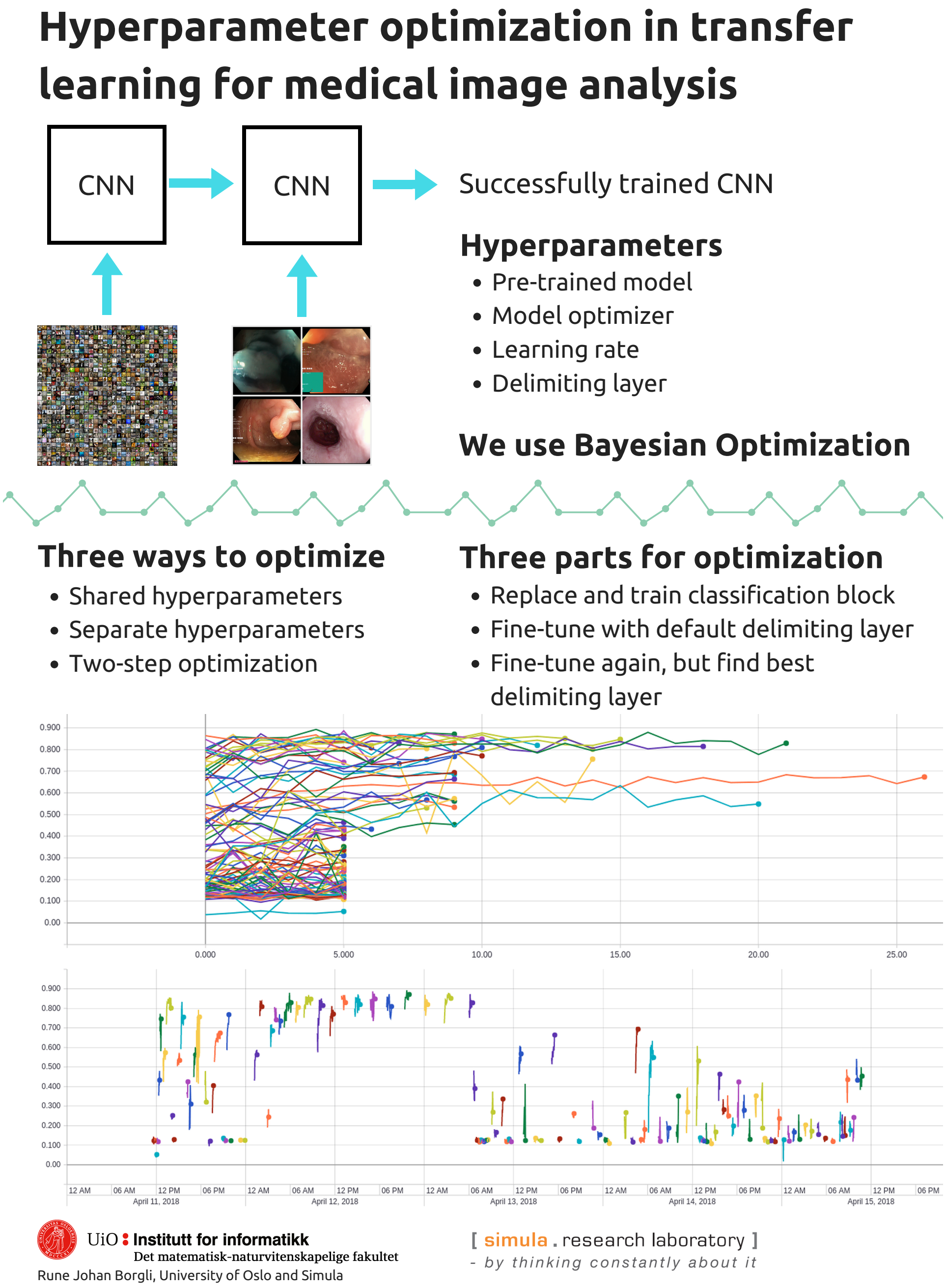
[143] Matthew D Zeiler. {ADADELTA:} An Adaptive Learning Rate Method. *CoRR*, abs/1212.5701, 2012.

[144] Ruikai Zhang, Yali Zheng, Tony Wing Chung Mak, Ruoxi Yu, Sunny H. Wong, James Y.W. Lau, and Carmen C.Y. Poon. Automatic Detection and Classification of Colorectal Polyps by Transferring Low-Level CNN Features from Nonmedical Domain. *IEEE Journal of Biomedical and Health Informatics*, 21(1):41–47, jan 2017.

[145] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning Transferable Architectures for Scalable Image Recognition. *CoRR*, abs/1707.07012, 2017.

Appendix A

Winning Poster from Autonomy Day 2018



Hyperparameter optimization in transfer learning for medical image analysis

Successfully trained CNN

Hyperparameters

- Pre-trained model
- Model optimizer
- Learning rate
- Delimiting layer

We use Bayesian Optimization

Three ways to optimize

- Shared hyperparameters
- Separate hyperparameters
- Two-step optimization

Three parts for optimization

- Replace and train classification block
- Fine-tune with default delimiting layer
- Fine-tune again, but find best delimiting layer

UiO : **Institutt for informatikk**
Det matematisk-naturvitenskapelige fakultet
Rune Johan Borgli, University of Oslo and Simula

[simula . research laboratory]
- by thinking constantly about it

Figure A.1: Poster winning the Best Poster Award from Autonomy Day 2018. The poster was awarded the best poster award at Autonomy Day 2018: 2nd workshop on Autonomous and Adaptive Systems held at Oslo Metropolitan University, May 3rd 2018. The poster was one of many presented during the workshop. It briefly presents our work and results from the Kvasir dataset experiment. Additonally, the work was presented by a three-minute lightning talk at the same event.

Appendix B

Plots From the Nerthus Optimization

B.1 Shared Hyperparameters Optimization Strategy

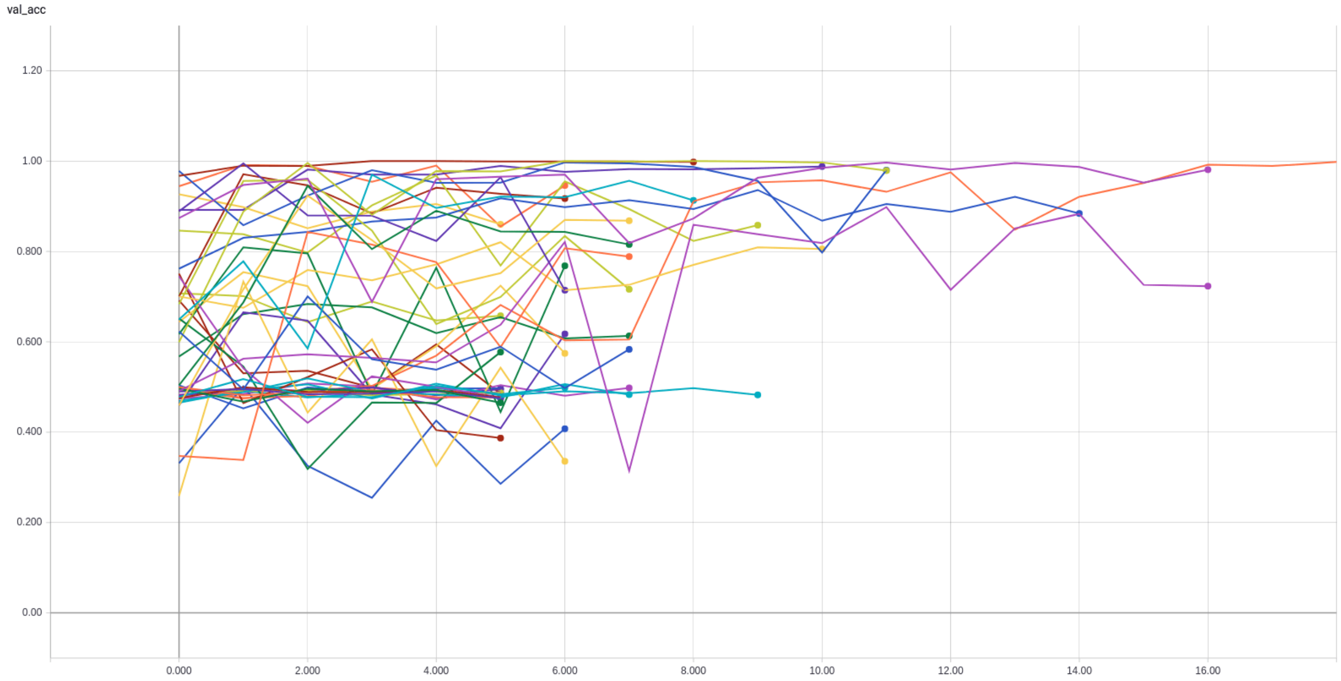


Figure B.1: Plot of Shared hyperparameters optimization strategy run in steps on the Kvasir dataset. X-axis is the number of epochs the training run has lasted and Y-axis is the attained validation accuracy. Each line represents a training run. One Bayesian optimization iteration is, therefore, segmented into several lines. The training of the classification block produces one line, while the training of the fine-tuning produces another line. The optimization of the delimiting layer will also produce a line. The plot is from the same run as figure 4.28, but with filering out every line that is not from the shared hyperparameter optimization strategy.

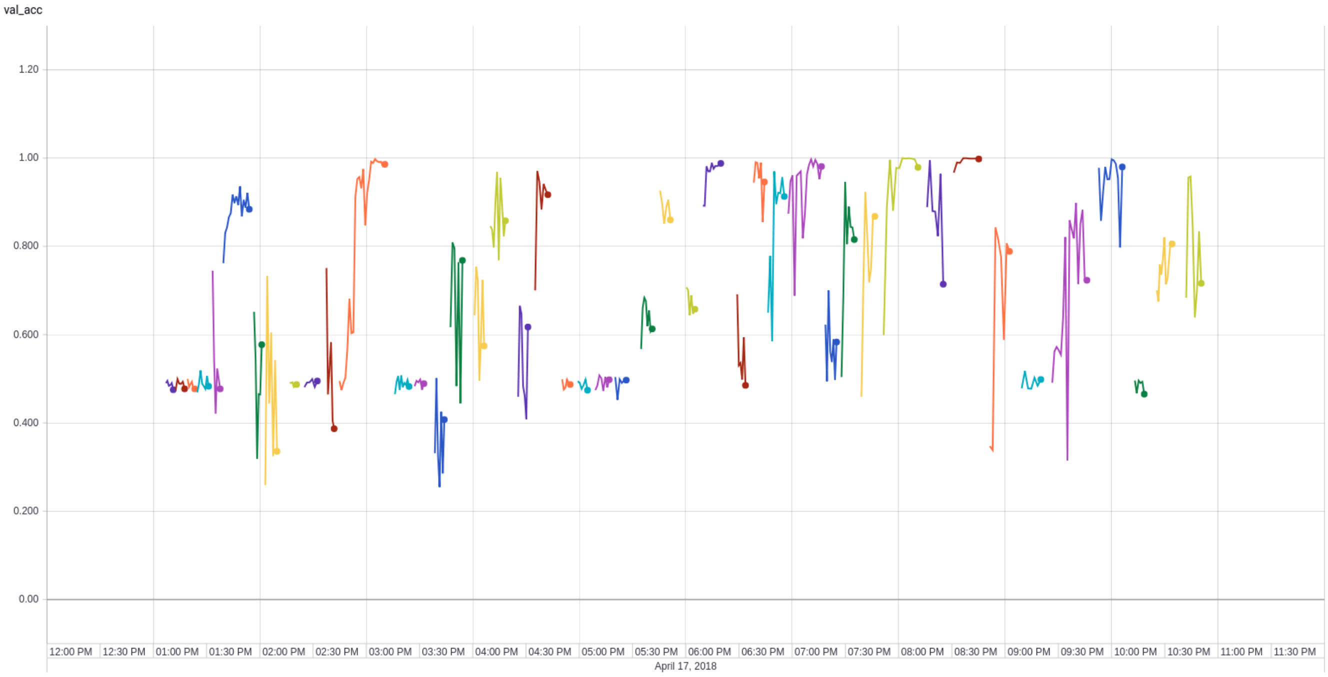


Figure B.2: Plot of Shared hyperparameters optimization strategy run in time on the Kvasir dataset. X-axis is timestamps for each finished epoch. Y-axis is the attained validation accuracy. Each line represents a training run. The plot is a subplot from figure 4.29 of the marked box of shared hyperparameters optimization strategy. We have added boxes with captions to show where each optimization step is plotted.

B.1.1 Model Optimization

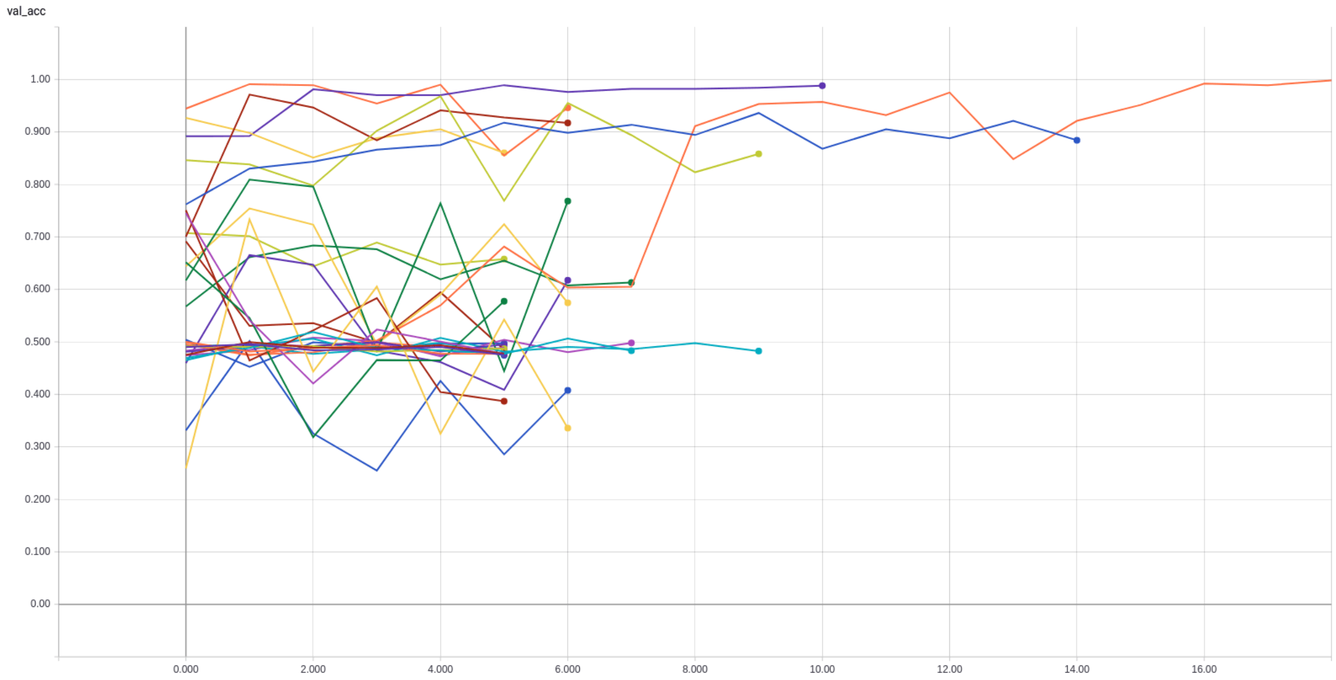


Figure B.3: The shared hyperparameters optimization strategy's model optimization run for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

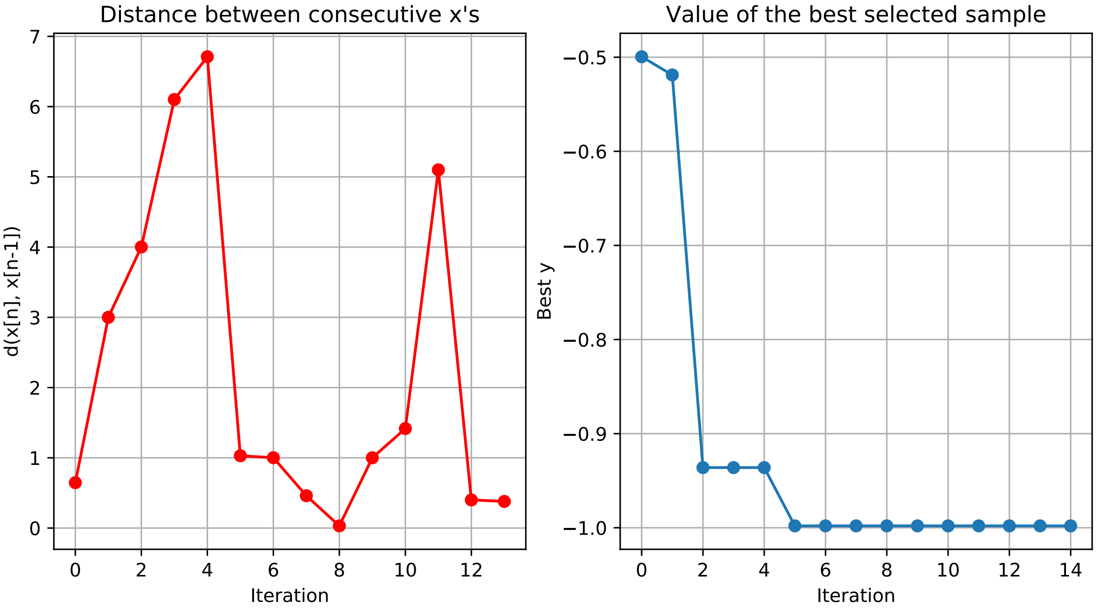


Figure B.4: Convergence plot of the shared hyperparameters optimization strategy's model optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

B.1.2 Layer Optimization

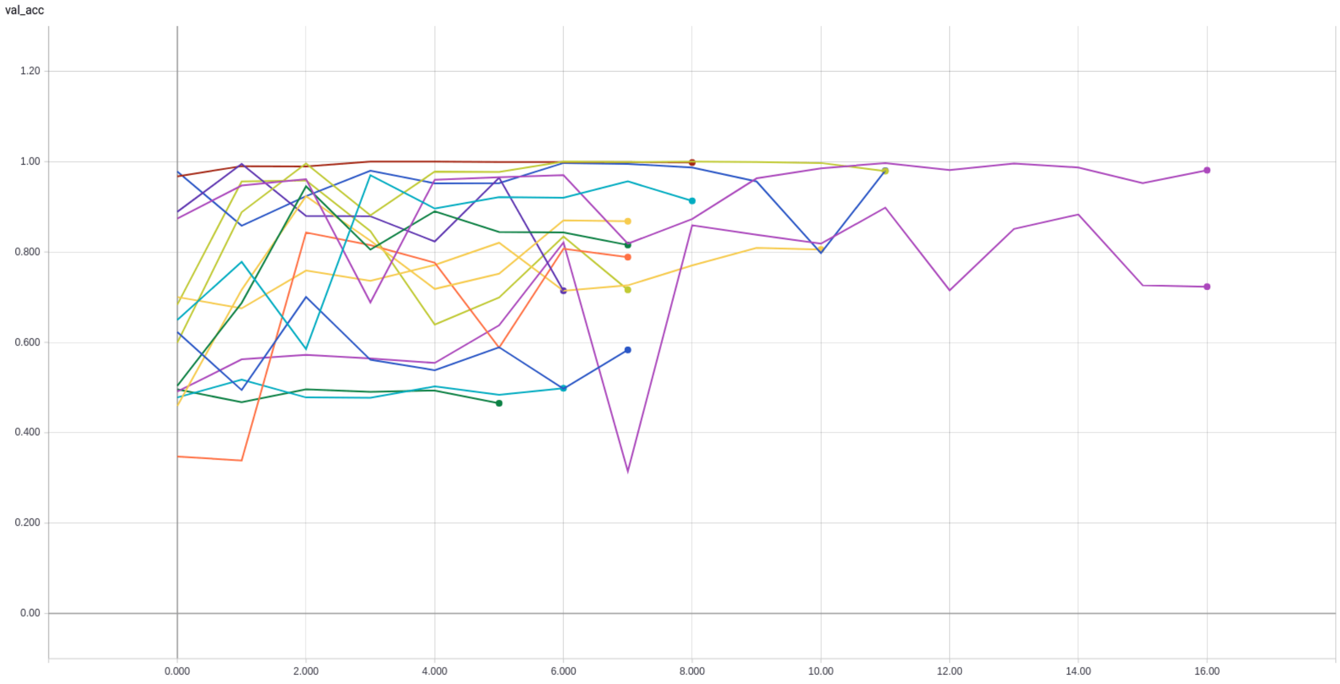


Figure B.5: The shared hyperparameters optimization strategy's layer optimization run for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

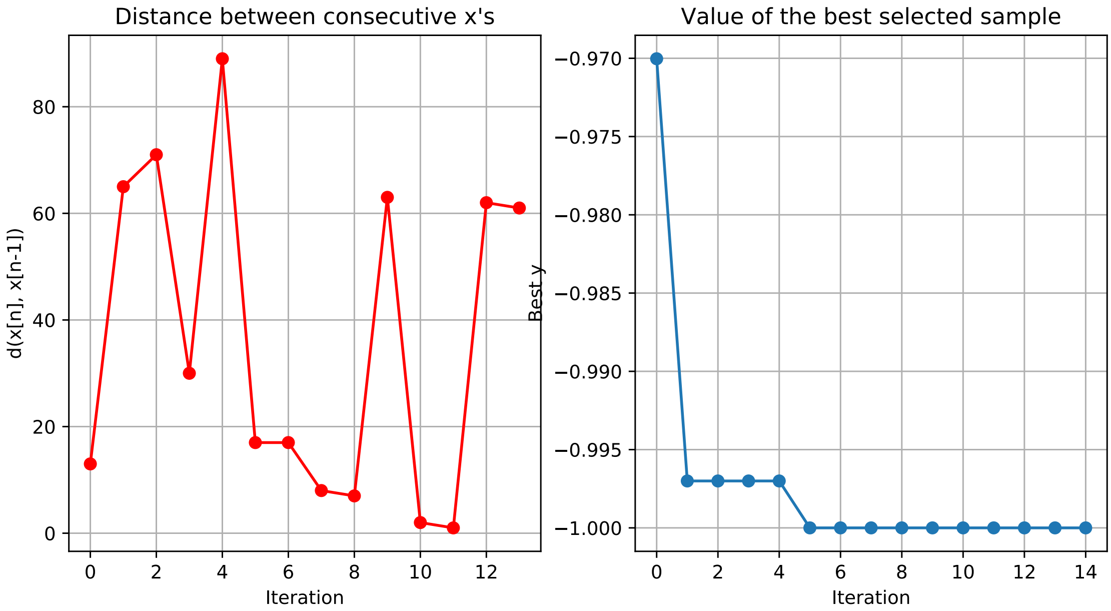


Figure B.6: Convergence plot of the shared hyperparameters optimization strategy's layer optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

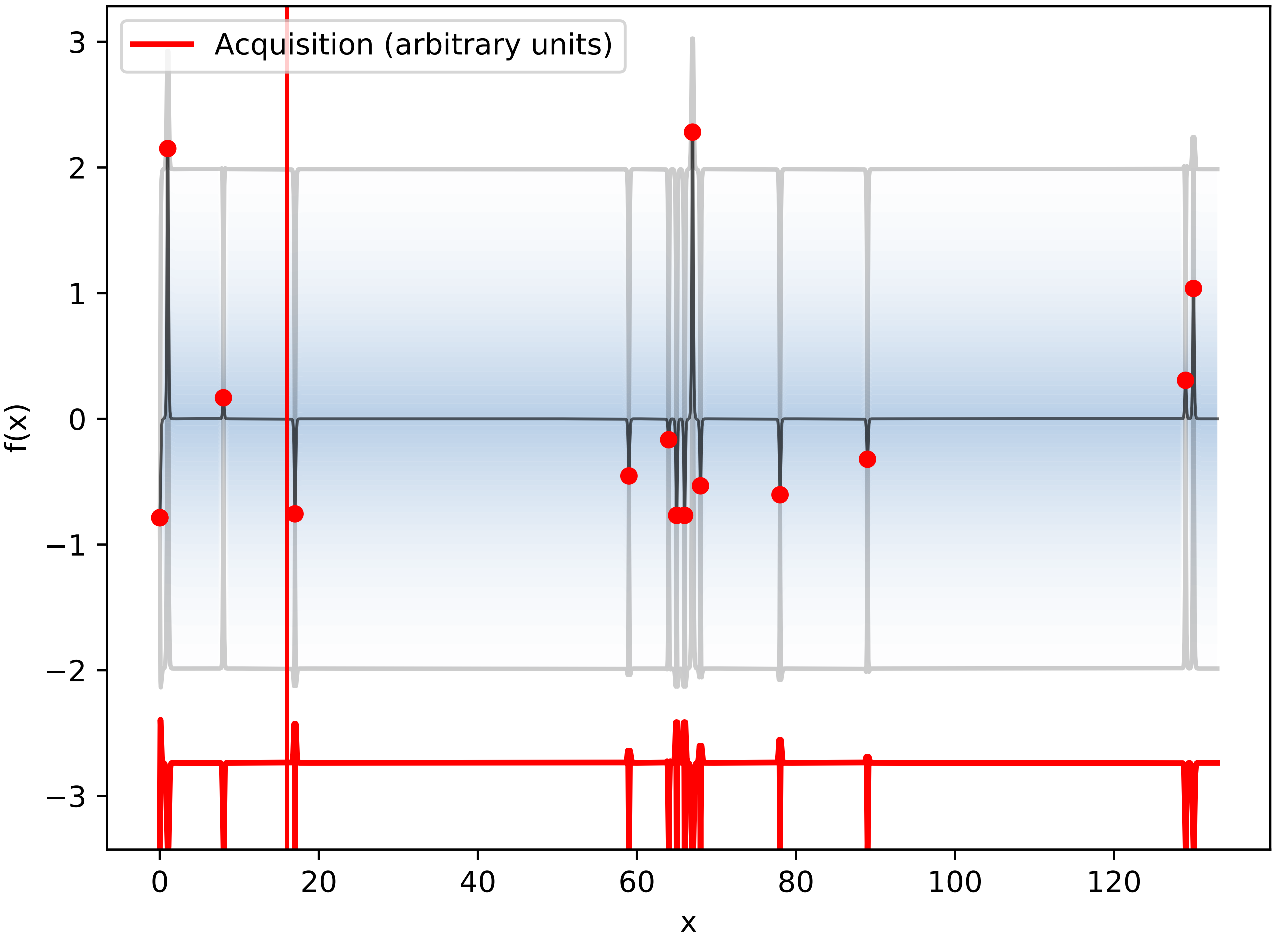


Figure B.7: Gaussian Process surrogate model and Expected Improvement acquisition function for the shared hyperparameters optimization strategy's layer optimization. The Y-axis is the expected result of a hyperparameter in the surrogate model. X-axis is the hyperparameter, which is this case is the delimiting layer. The black line represents the posterior mean, the gray lines represents the posterior uncertainty, the red dots are tested hyperparameters and their results on the surrogate model, the red vertical line is where the acquisition function would try the next test should it run for another iteration, and, finally, the horizontal red curve is the acquisition function in arbitrary units. A lower posterior mean is better, and a higher value for the acquisition function is where the next hyperparameter values will be taken from.

B.2 Separate Hyperparameters Optimization Strategy

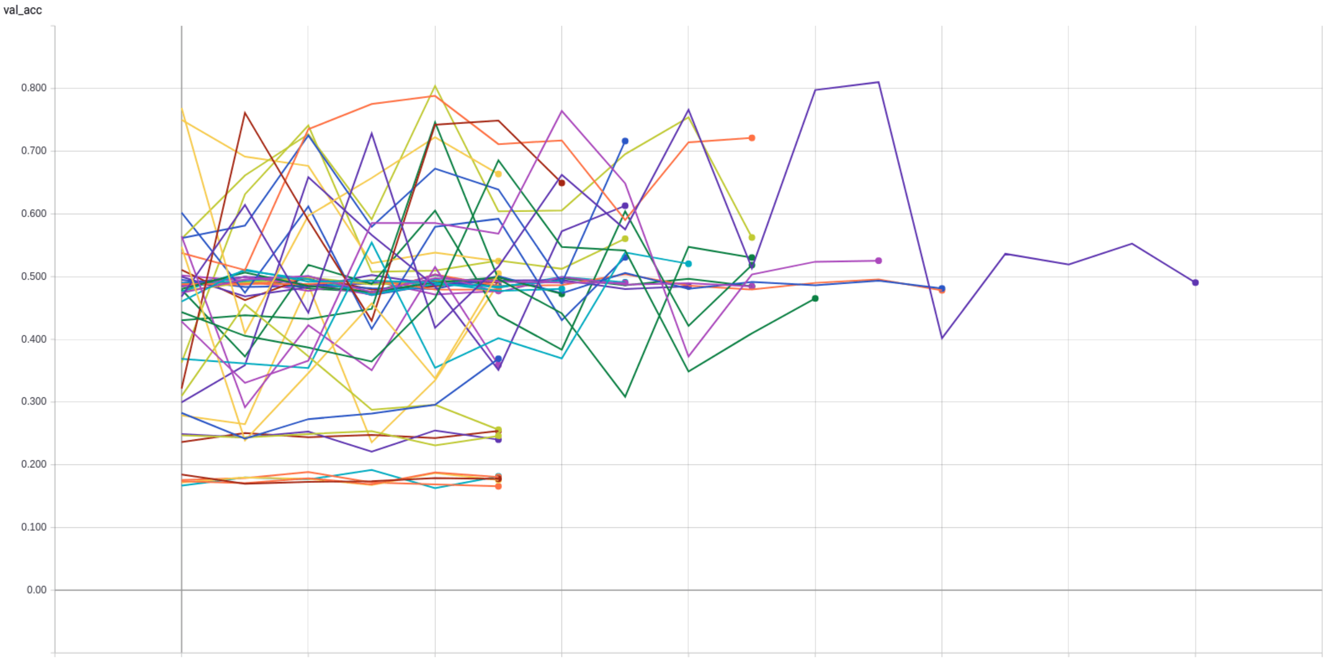


Figure B.8: The separate hyperparameters optimization strategy's plot of training runs for each epoch.

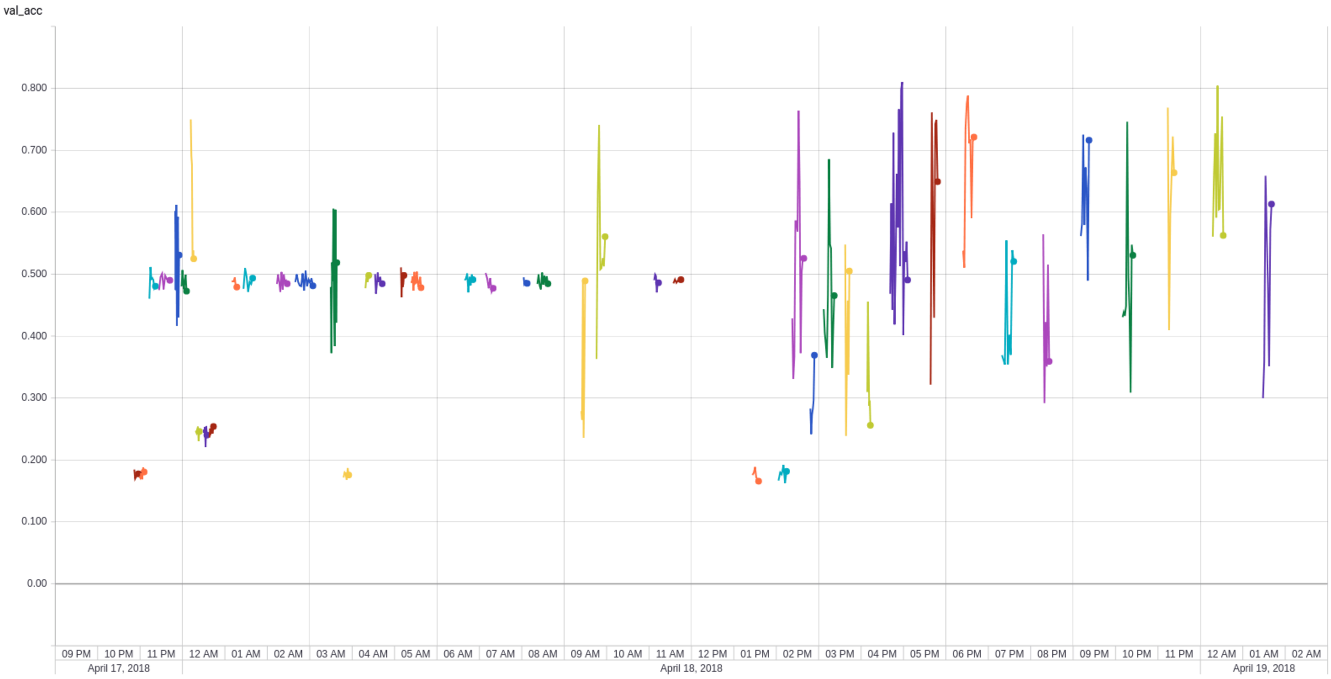


Figure B.9: The separate hyperparameters optimization strategy's plot of training runs for time. The model optimization and layer optimization has been marked with black boxes.

B.2.1 Model Optimization

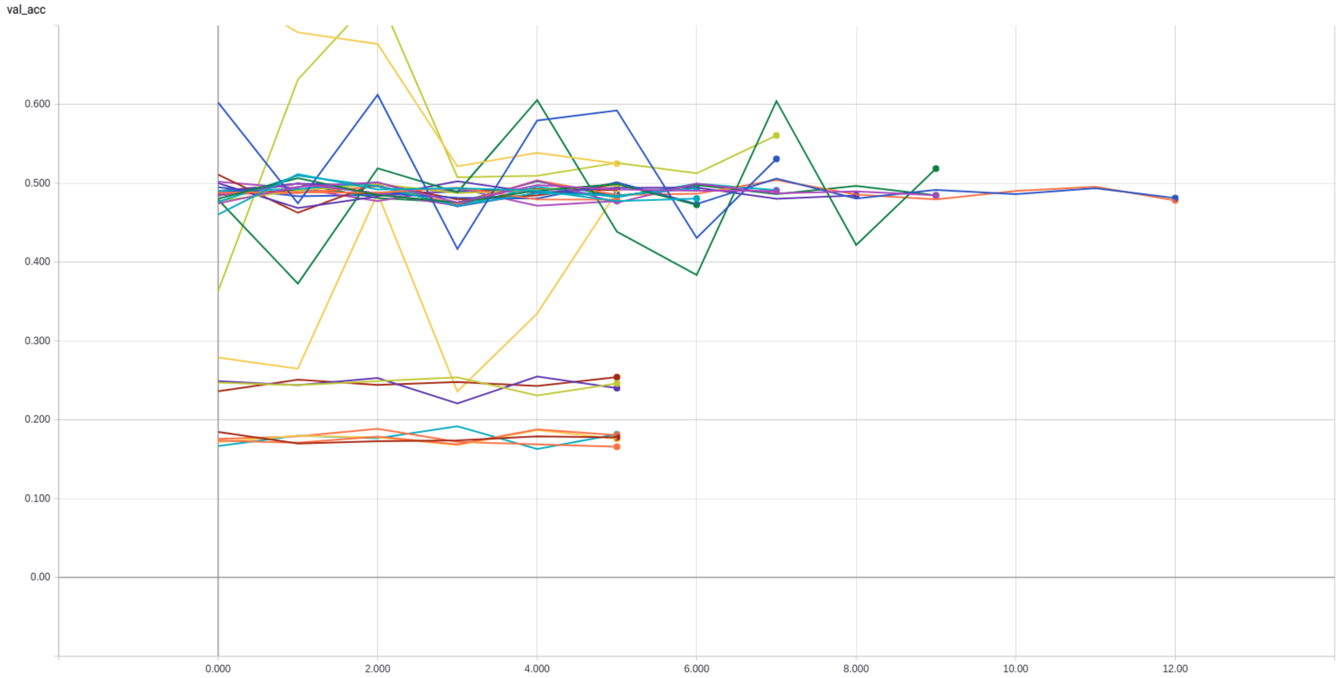


Figure B.10: The separate hyperparameters optimization strategy's plot of model optimization training runs for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

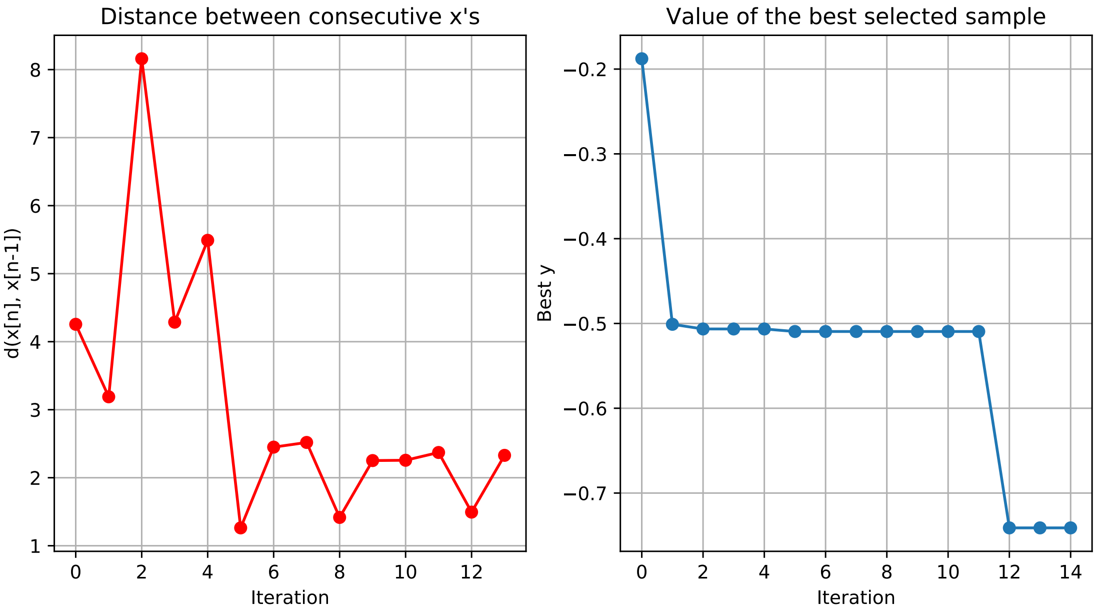


Figure B.11: Convergence plot of the separate hyperparameters optimization strategy's model optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

B.2.2 Layer Optimization

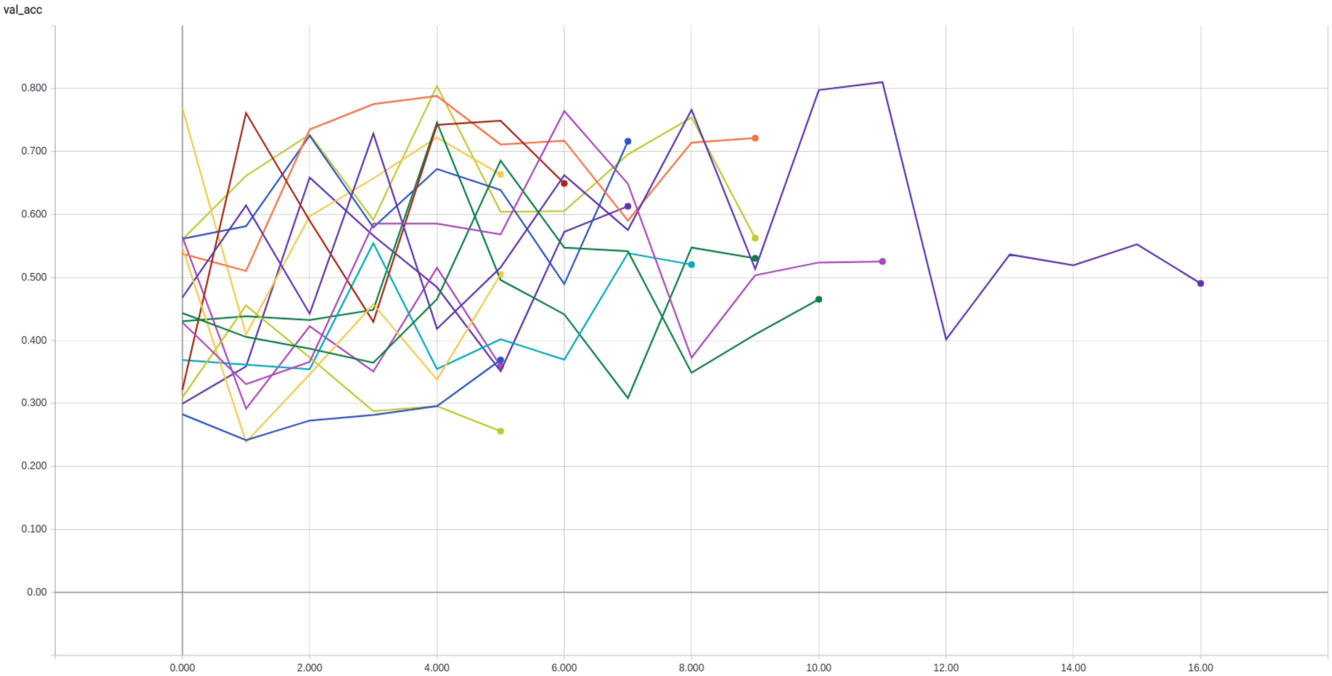


Figure B.12: The separate hyperparameters optimization strategy's plot of layer optimization training runs for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

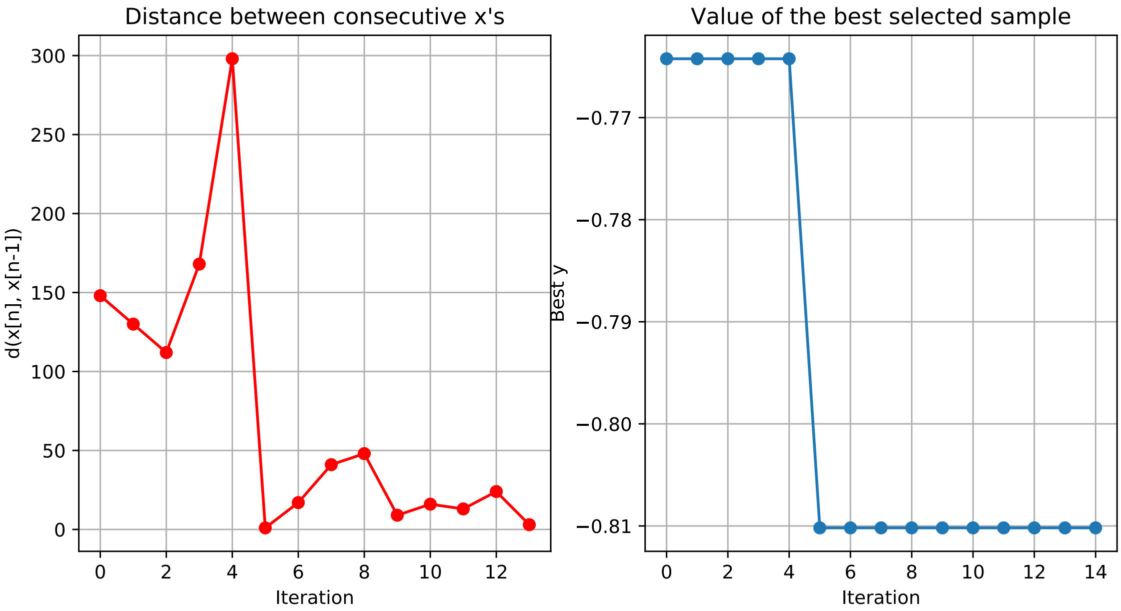


Figure B.13: Convergence plot of the separate hyperparameters optimization strategy's layer optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

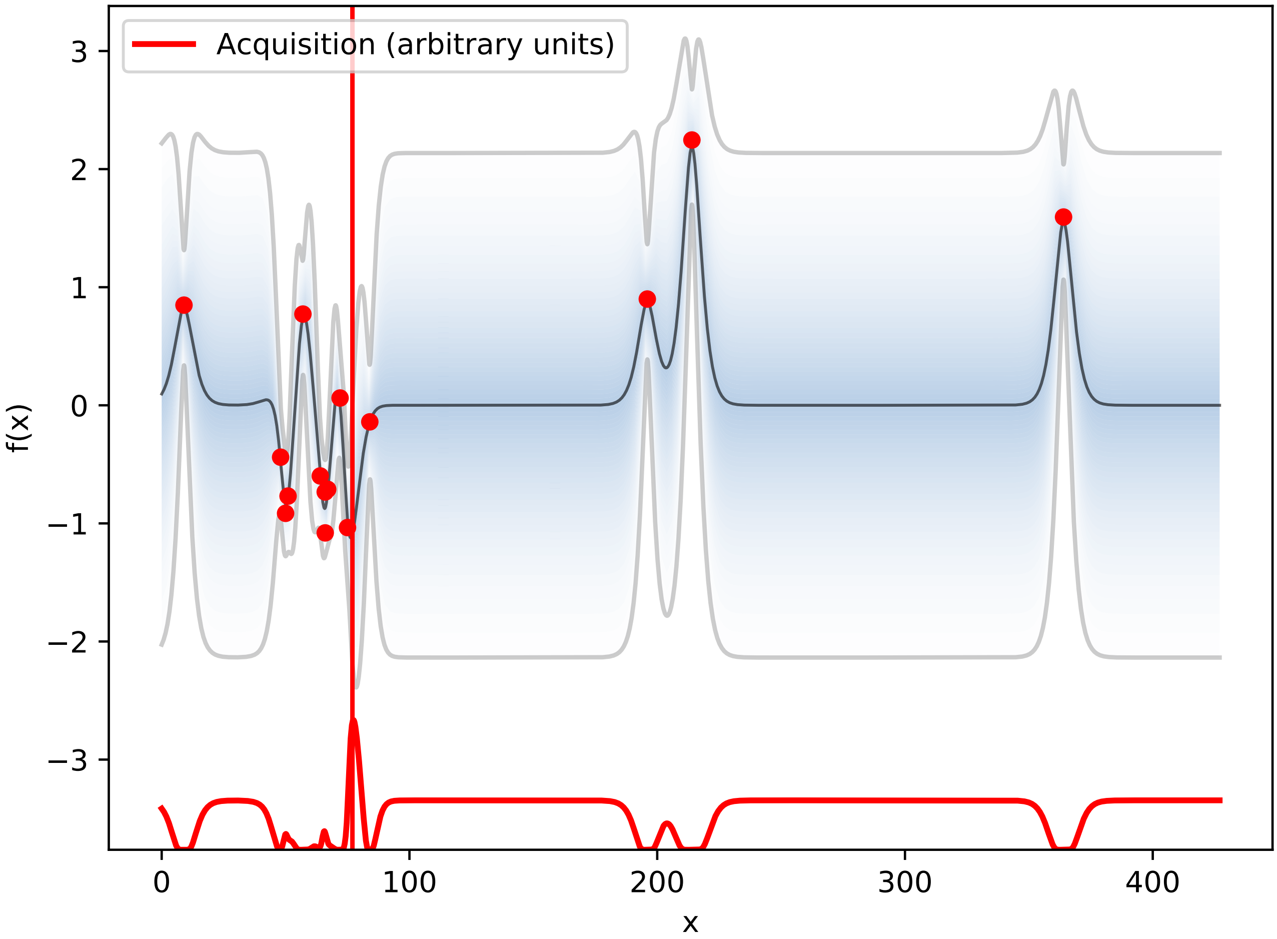


Figure B.14: Gaussian Process surrogate model and Expected Improvement acquisition function for the separate hyperparameters optimization strategy's layer optimization. The Y-axis is the expected result of a hyperparameter in the surrogate model. X-axis is the hyperparameter, which is this case is the delimiting layer. The black line represents the posterior mean, the gray lines represents the posterior uncertainty, the red dots are tested hyperparameters and their results on the surrogate model, the red vertical line is where the acquisition function would try the next test should it run for another iteration, and, finally, the horizontal red curve is the acquisition function in arbitrary units. A lower posterior mean is better, and a higher value for the acquisition function is where the next hyperparameter values will be taken from.

B.3 Separate Optimizations Optimization Strategy

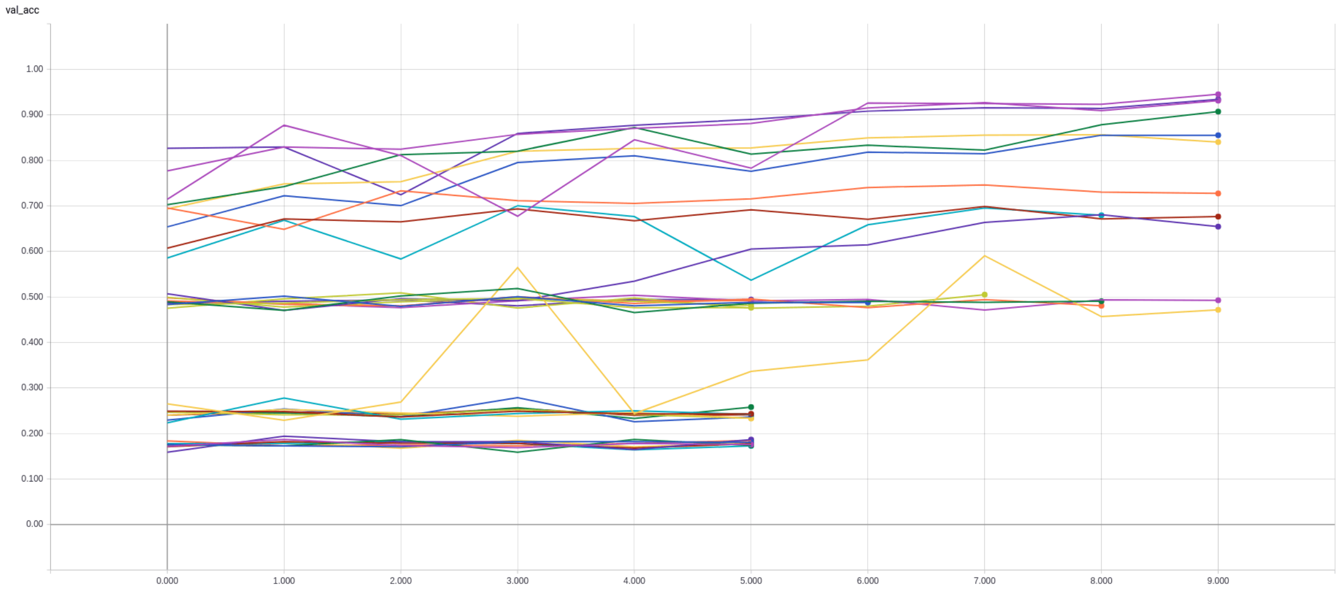


Figure B.15: The separate optimizations optimization strategy's plot of training runs for each epoch.

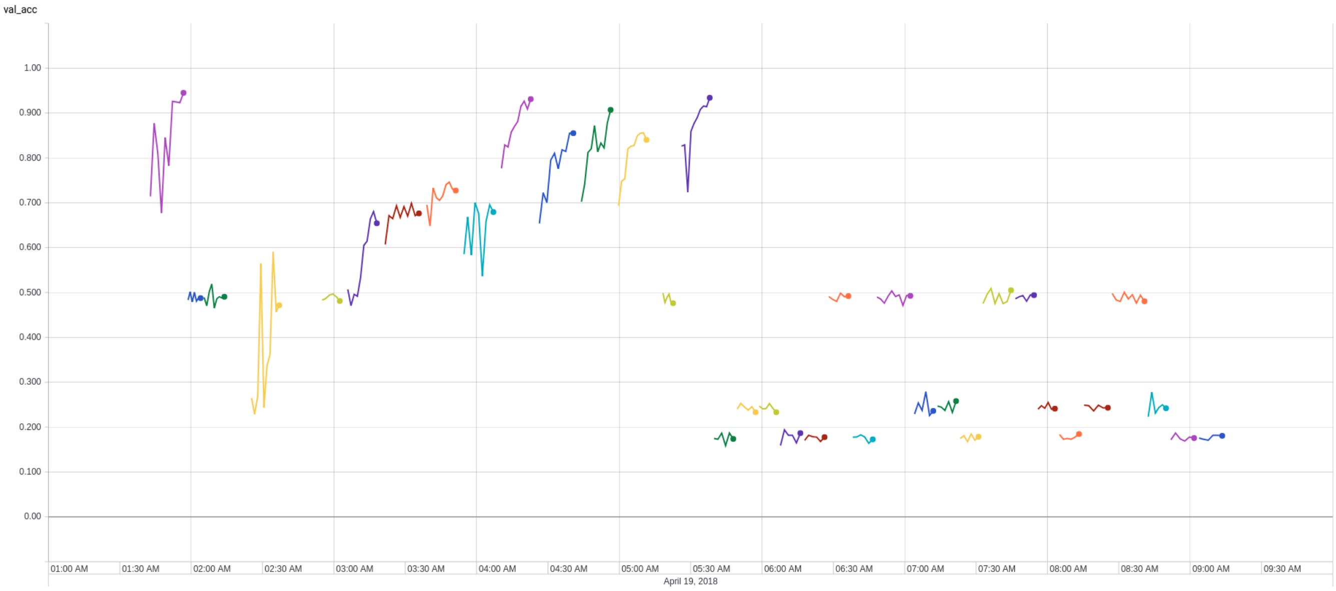


Figure B.16: The separate optimizations optimization strategy's plot of training runs for time. The classification block optimization, fine-tuning optimization, and layer optimizationa is marked with black boxes.

B.3.1 Classification Block Optimization

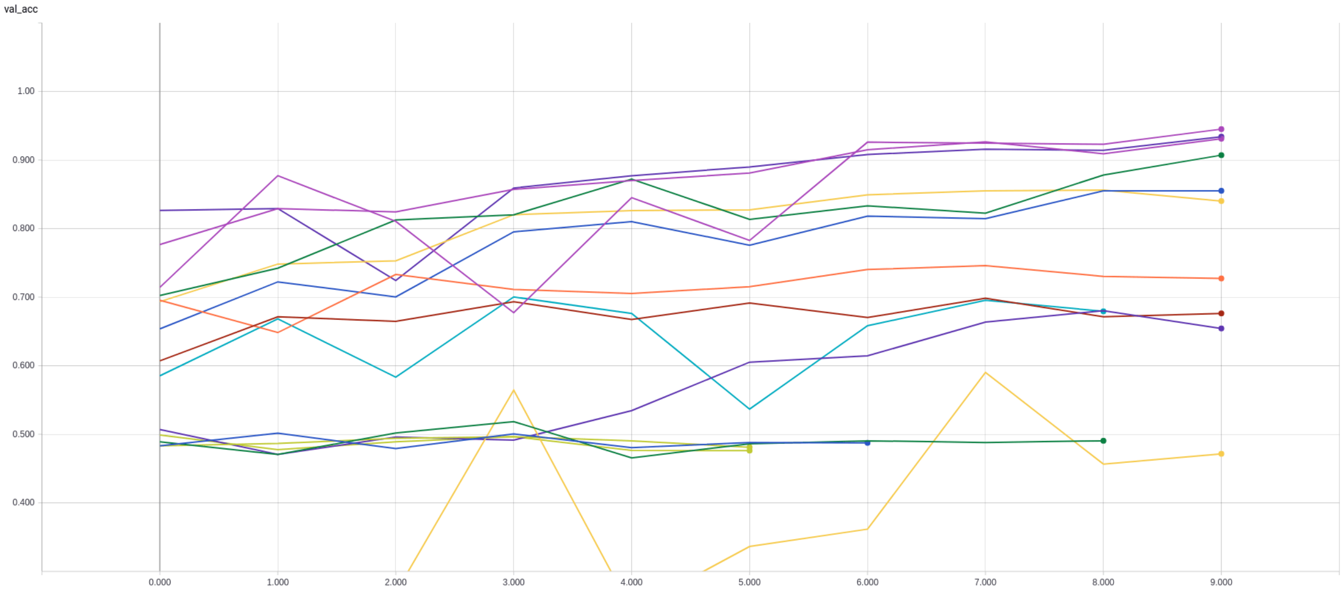


Figure B.17: The separate optimizations optimization strategy's classification block optimization run for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

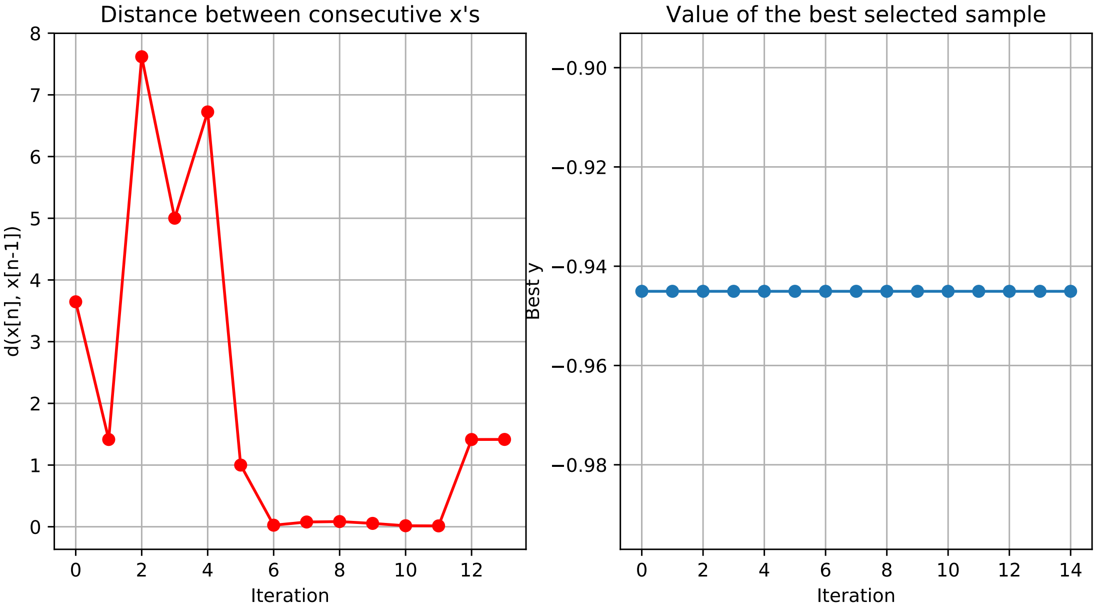


Figure B.18: Convergence plot of the separate optimizations optimization strategy's classification block optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

B.3.2 Fine-tuning Optimization

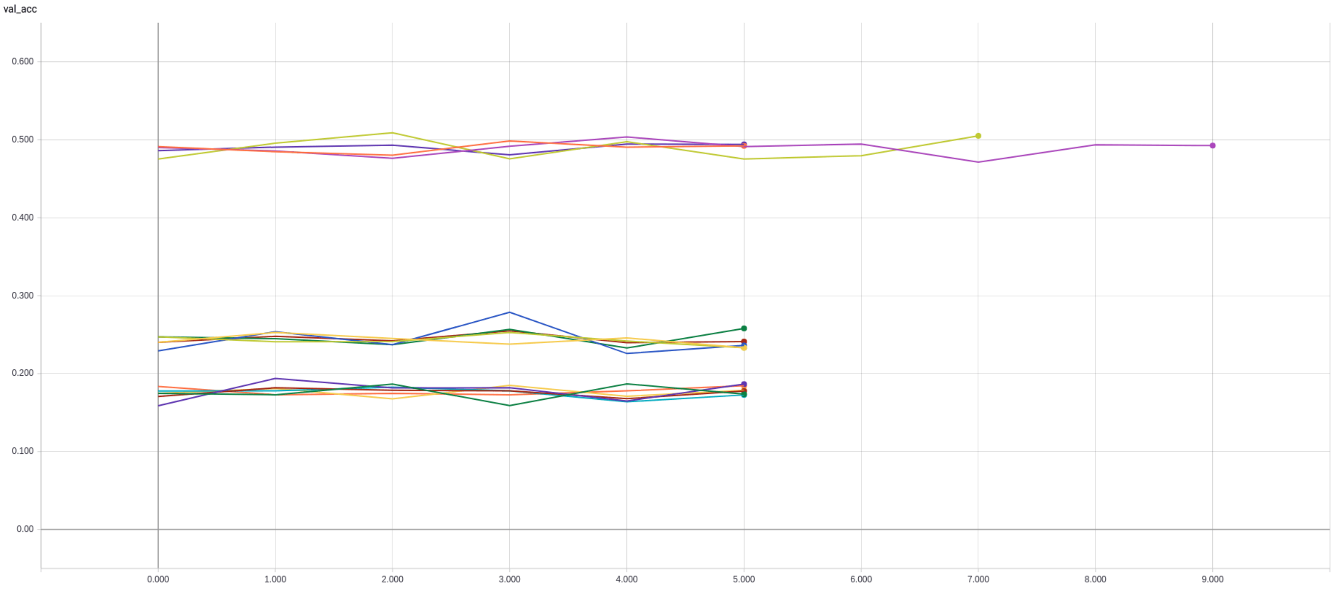


Figure B.19: The separate optimizations optimization strategy's fine-tuning optimization run for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

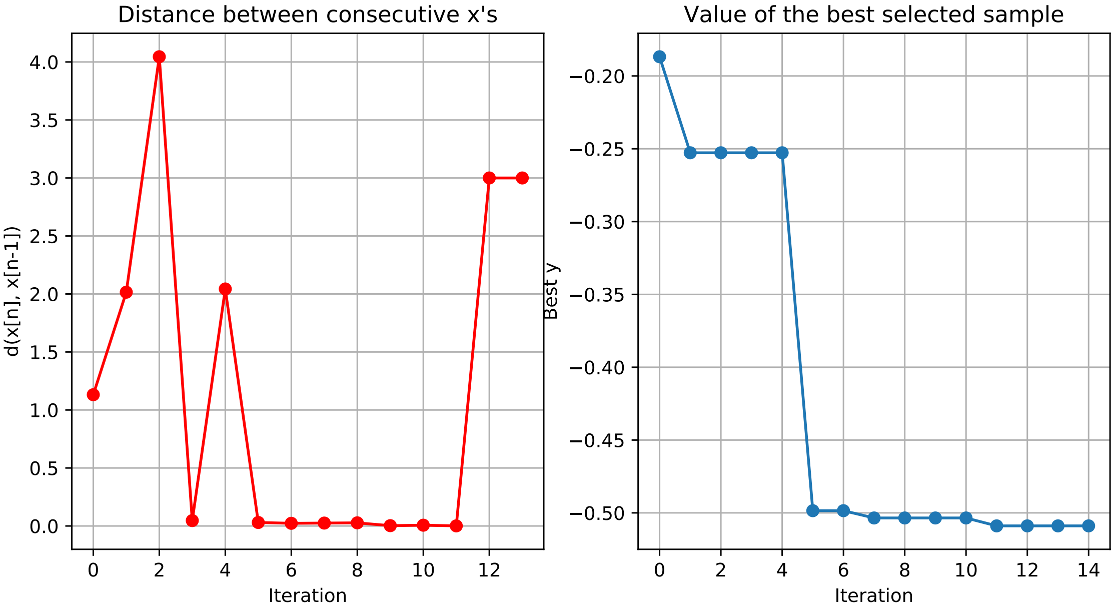


Figure B.20: Convergence plot of the separate optimizations optimization strategy's fine-tuning optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

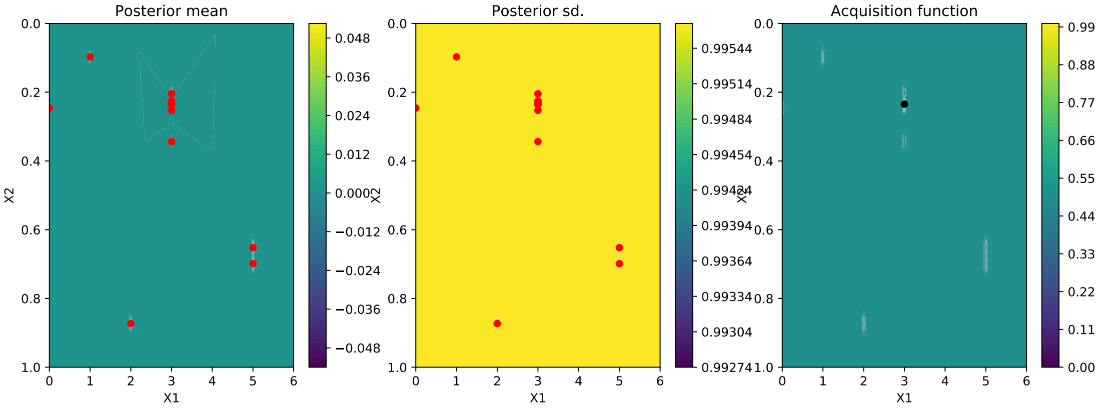


Figure B.21: Gaussian Process surrogate model and Expected Improvement acquisition function for the separate optimizations optimization strategy's fine-tuning optimization. The Y-axis is the normalized learning rate domain and the X-axis is the index of each gradient descent optimization function in a list. The color gradient reprentests the posterior mean in the first graph, the posterior standard deviation in the second graph, and the acquisition function in the final graph. The graph shows the model after the optimization. The red dots in the first two graphs are the tests that have been done during the optimization. The black dot in the acquisition function is where the next test would have occured should the optimization continue for another iteration.

B.3.3 Layer Optimization

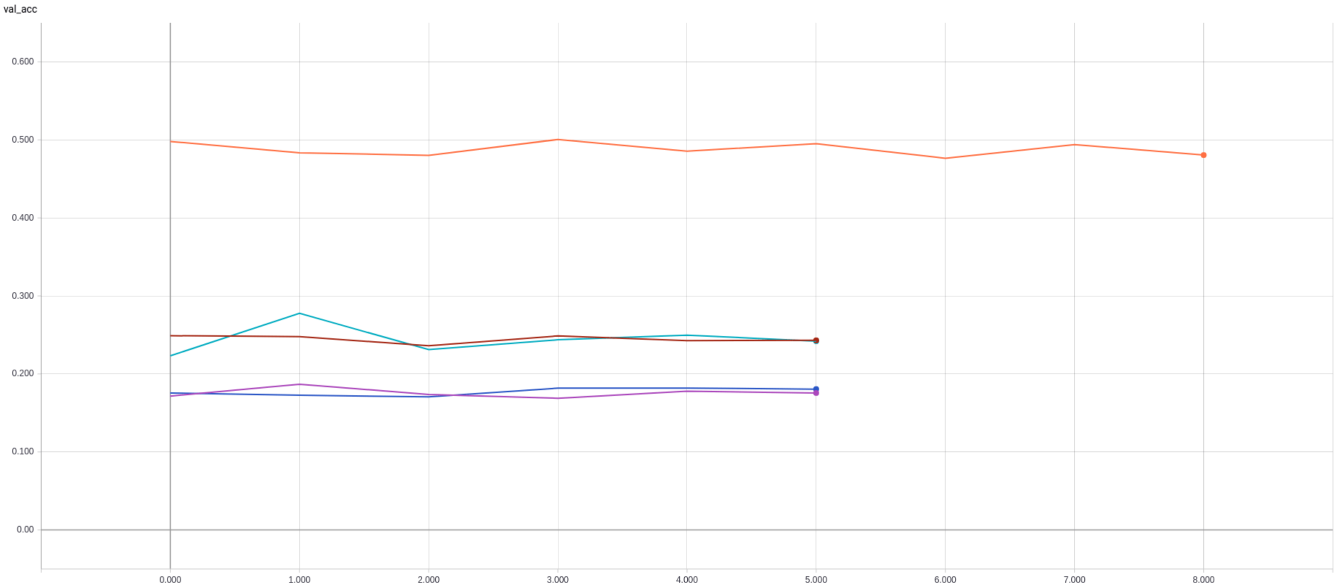


Figure B.22: The separate optimizations optimization strategy's layer optimization run for each epoch. Training runs from the model optimization is plotted in the graph for validation accuracy in Y-axis and epoch in X-axis. Classification block training and fine-tuning is plotted together.

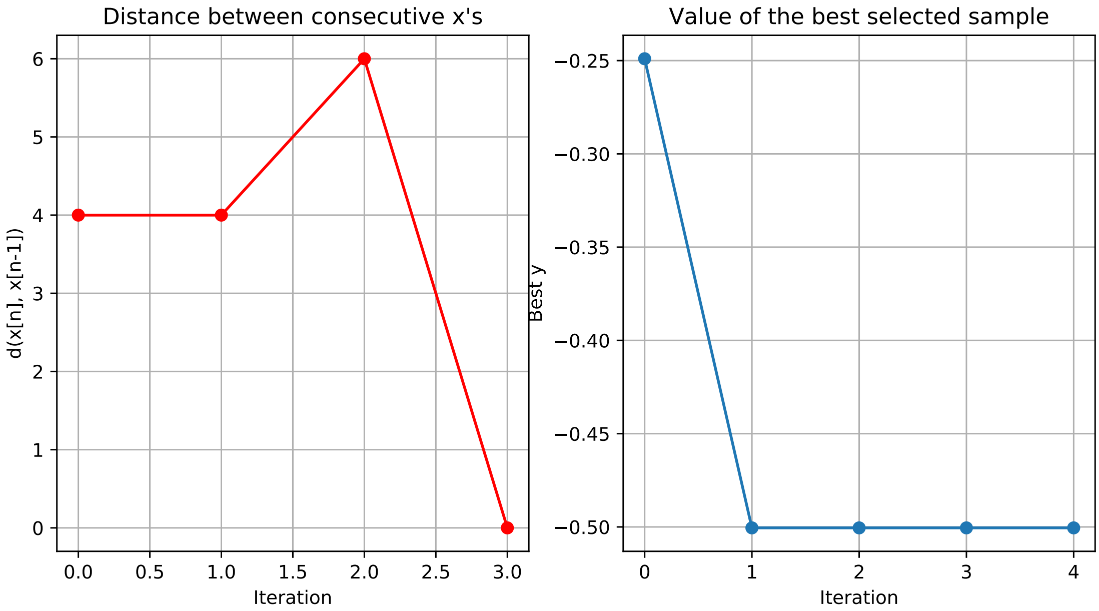


Figure B.23: Convergence plot of the separate optimizations optimization strategy's layer optimization run. The first plot shows the distance between each hyperparameter using Euclidean distance, where the Euclidean distance is plotted as Y-axis, and the iteration is plotted as X-axis. The iteration means the going from one iteration to another, so for example; iteration 0 means the distance between iteration 0 and 1. The second figure shows the highest attained validation accuracy, in Y-axis, for each iteration, in X-axis. The validation accuracy is negated as that is how it is handled in the GPyOpt library, which is used to create the plot. The iterations here are normal iterations.

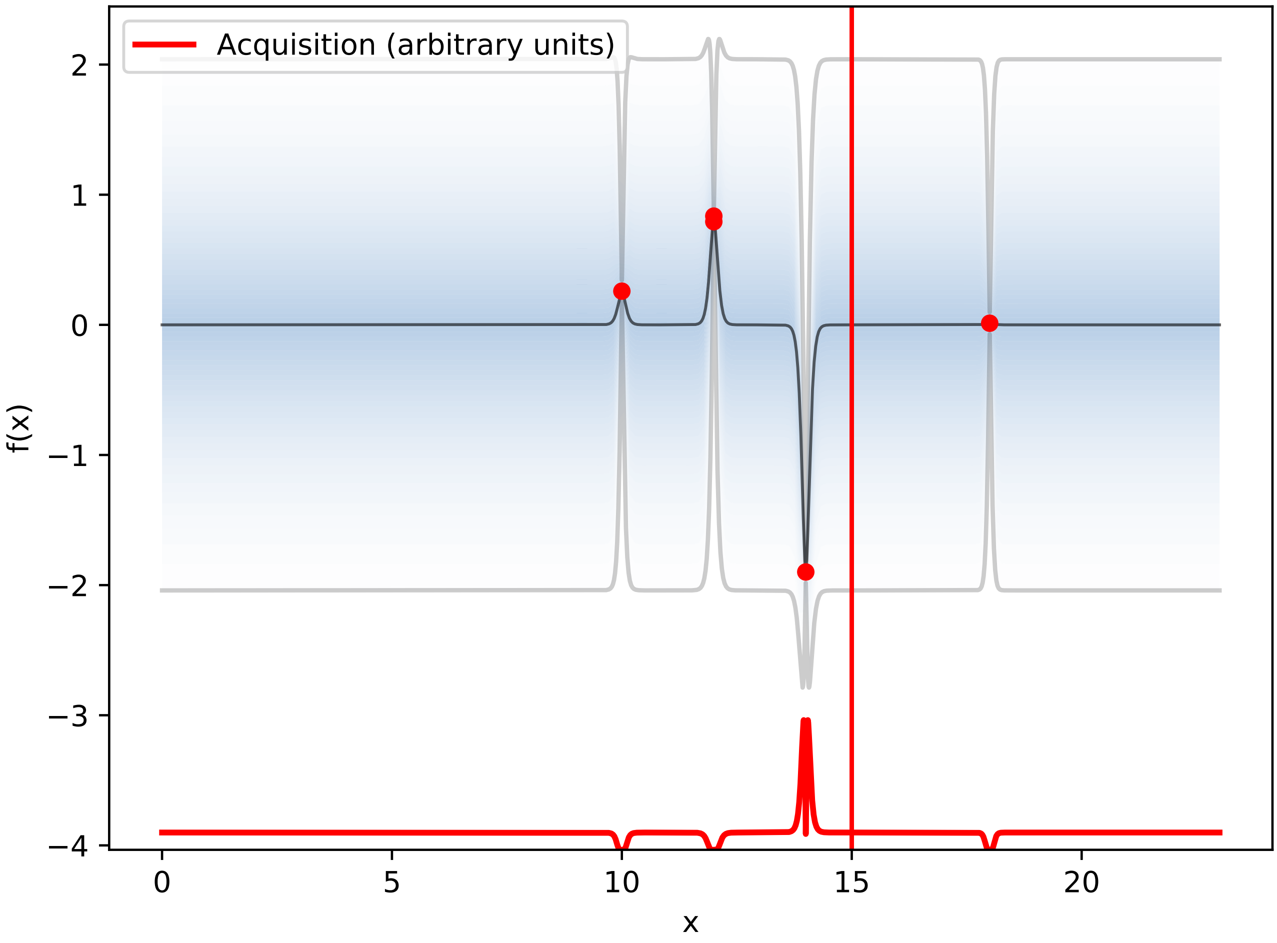
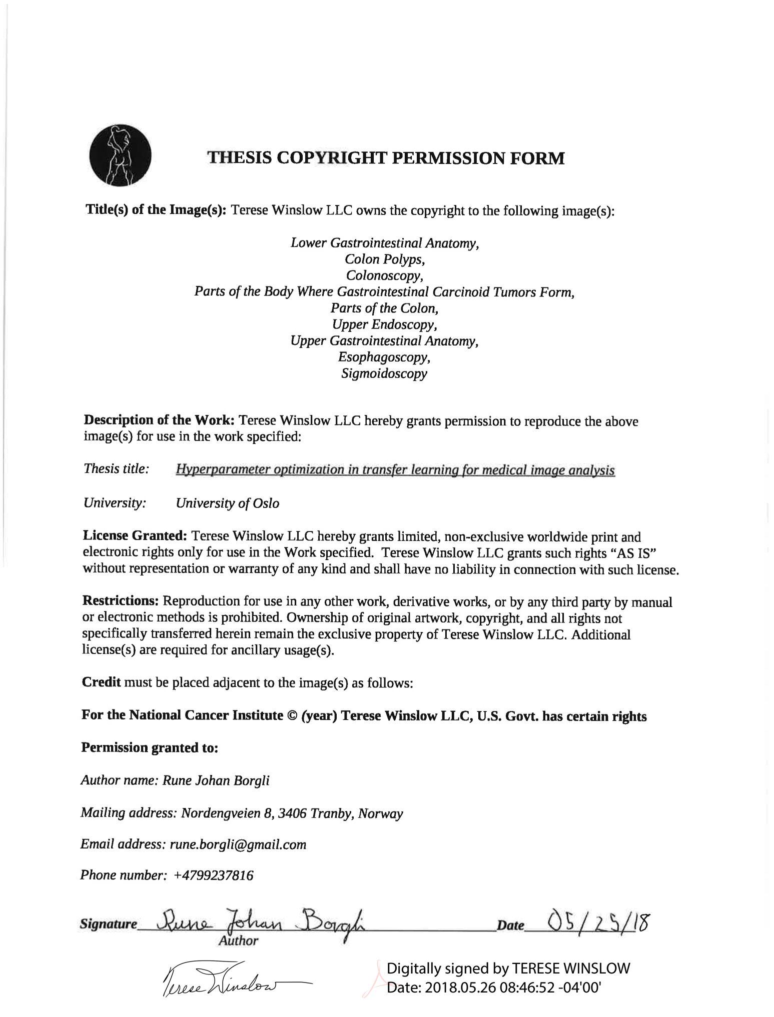


Figure B.24: Gaussian Process surrogate model and Expected Improvement acquisition function for the separate optimizations optimization strategy's layer optimization. The Y-axis is the expected result of a hyperparameter in the surrogate model. X-axis is the hyperparameter, which is this case is the delimiting layer. The black line represents the posterior mean, the gray lines represents the posterior uncertainty, the red dots are tested hyperparameters and their results on the surrogate model, the red vertical line is where the acquisition function would try the next test should it run for another iteration, and, finally, the horizontal red curve is the acquisition function in arbitrary units. A lower posterior mean is better, and a higher value for the acquisition function is where the next hyperparameter values will be taken from.

Appendix C

Permission To Use Illustrations



THESIS COPYRIGHT PERMISSION FORM

Title(s) of the Image(s): Terese Winslow LLC owns the copyright to the following image(s):

Lower Gastrointestinal Anatomy,
Colon Polyps,
Colonoscopy,
Parts of the Body Where Gastrointestinal Carcinoid Tumors Form,
Parts of the Colon,
Upper Endoscopy,
Upper Gastrointestinal Anatomy,
Esophagoscopy,
Sigmoidoscopy

Description of the Work: Terese Winslow LLC hereby grants permission to reproduce the above image(s) for use in the work specified:

Thesis title: *Hyperparameter optimization in transfer learning for medical image analysis*

University: *University of Oslo*

License Granted: Terese Winslow LLC hereby grants limited, non-exclusive worldwide print and electronic rights only for use in the Work specified. Terese Winslow LLC grants such rights "AS IS" without representation or warranty of any kind and shall have no liability in connection with such license.

Restrictions: Reproduction for use in any other work, derivative works, or by any third party by manual or electronic methods is prohibited. Ownership of original artwork, copyright, and all rights not specifically transferred herein remain the exclusive property of Terese Winslow LLC. Additional license(s) are required for ancillary usage(s).

Credit must be placed adjacent to the image(s) as follows:

For the National Cancer Institute © (year) Terese Winslow LLC, U.S. Govt. has certain rights

Permission granted to:

Author name: Rune Johan Borgli

Mailing address: Nordengveien 8, 3406 Tranby, Norway

Email address: rune.borgli@gmail.com

Phone number: +4799237816

Signature Rune Johan Borgli **Date** 05/25/18
Author

Terese Winslow

Digitally signed by TERESE WINSLOW
Date: 2018.05.26 08:46:52 -04'00'

Figure C.1: Signed Thesis Copyright Permission Form which gives consent to use the illustrations in the background showing a colonoscopy and gastrointestinal anatomy.