Automatic hyperparameter optimisation and neural architecture search

Final Year Project Report

Author: Dobromir Marinov
Degree: BSc Digital and Technology Solutions
Supervisor: Dr Daniel Karapetyan
Date: June 2019

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor Dr Daniel Karapetyan. His continuous guidance and support have made this project possible and have also helped me discover the world of science and research. Secondly, I would like to thank my colleagues, Bob Komara and Laurence Smith, for always being available to help and to answer all of my questions related to data science and software engineering. Last but not least, I would like to thank my parents for their continuous support and patience and for helping me become the person that I am today.

Abstract

In recent years, machine learning has massively grown in popularity. It is being used extensively for predictive analysis, automatic image recognition, scene detection, speech recognition and in many other areas. However, before a machine learning model can be used, it needs to be trained. The training process, and in many cases even the architecture of the models, is dictated by parameters, known as hyperparameters. Modern algorithms are complex and their hyper-parameter spaces are vast. Moreover, testing each hyper-parameter set may take CPU-years. We present a novel approach that reduces the time needed to train a model. This will enable better optimisation of hyperparameters and, as a result, higher performance of machine learning applications.

Table of contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Aims and objectives . . . 2

2 Background **3**
2.1 Machine Learning . . . 3
2.2 Ensemble methods . . . 4
2.3 Neural Networks . . . 5
2.4 Hyperparameter optimisation . . . 5
2.5 Neural architecture search . . . 6

3 Proposed algorithm **8**
3.1 Core idea . . . 8
3.2 Feature engineering . . . 11
3.3 Linear regression . . . 13
3.4 Clustering . . . 14
3.5 Selecting predicted models . . . 16

4 Evaluations **18**
4.1 Datasets . . . 18
4.2 Baseline performance . . . 19
4.3 Binary classification . . . 21
4.4 Image recognition . . . 22

5 Technology **24**
5.1 ML tools . . . 24
5.1.1 XGBoost . . . 24
5.1.2 Scikit-learn . . . 25
5.1.3 Keras and TensorFlow . . . 25
5.2 Data manipulation and visualisation . . . 26
5.2.1 Pandas . . . 26
5.2.2 Matplotlib . . . 26

5.3 Cloud computing 26
5.3.1 Google Colaboratory 27
5.3.2 Amazon Web Services 27
5.3.3 Microsoft Azure 28
5.4 Testing and integration tools 28
5.4.1 PyUnit and Travis CI 28
5.4.2 Packaging 28

6 Product development **29**
6.1 Design and architecture 29
6.1.1 Boar ML 29
6.1.2 Predictive hyperparameter optimisation 30
6.2 Additional tools developed 31
6.3 Testing 32

7 Project planning **33**
7.1 Overview 33
7.2 Risk management 34
7.2.1 Continuous integration 35

8 Conclusion **36**
8.1 Future work 36

9 Bibliography **37**

Chapter 1

Introduction

With the increase of computational power in recent years, a significant amount of researchers, scientists and engineers have turned their attention towards machine learning [1, 2]. It is particularly well suited for problems which may not have obvious solutions or require automatic discovery of optimal strategies to solve them. This has lead to the development of many new machine learning algorithms. It turns out, however, that the task of deciding what the best run-time settings for those algorithms should be, is not straightforward.

The discussion starts with an explanation of the motivation behind the project, followed by a clear definition of the aims and objectives. The rest of the project is organised as follow: Chapter 2 gives an overview of the current state of the machine learning field and introduces and summarises the concepts needed in our further discussion. Chapter 3 outlines the proposed novel approach for hyperparameter optimisation, explaining the intuition behind it. Chapter 4 shows the performance of the proposed algorithm on a number of different problems. Chapter 5 gives detailed information about the technology that has been used in order to implement, test and run the algorithm. Chapter 6 describes the software architecture of the project and goes into more in-depth analysis of the technical details of the project. Chapter 7 focuses on the management of the project and describes some of the challenges that were faced along the way. Chapter 8 provides a formal conclusion, as well as a description of the future work that can be undertaken.

1.1 Motivation

Machine learning algorithms can solve complex problems in a wide range of domains [3]. In some cases, they are even able to achieve better performance than the top human-experts in

a given field [4]. The performance of machine learning models, built using the algorithms, is determined to a large degree by the appropriate choice of parameters for the algorithms [5]. Those parameters are typically called *hyperparameters* and represent values that dictate the training process, such as the learning rate. The number of hyperparameters can vary quite drastically between algorithms, with some having as little as 3-5 hyperparameters, and other having hundreds of them. Additionally, the hyperparameters can take multiple values, leading to combinatorial explosions, and coupled with the fact that training can take significant amount of time, it makes simple brute-force techniques infeasible.

Artificial neural networks are a popular choice for solving machine learning problems [6]. They use different architectures for different tasks and selecting the best one for a given dataset is considered to be a non-trivial problem. The full architectures can be viewed as a combination of many hyperparameters, specifying the types of layers, their order, the connections between layers, etc. It is often referred to as 'Neural architecture search' [7] and with the advances in deep learning, which uses more and more layers to create state-of-the-art architectures, the importance of building 'smarter' algorithms for hyperparameter optimisation has increased massively. A good optimisation algorithm should not only be able to find suitable configurations of hyperparameters in a reasonable time, but it should also try to minimise the computational resources needed. On top of that, it should be consistent and able to generalise well, in order to work for different types of datasets.

1.2 Aims and objectives

The aim of the project is to develop a novel hyperparameter optimisation algorithm, which is capable of finding suitable configurations automatically. The goal is to minimise the computational time and resources needed, while achieving the best possible performance. The hypothesis is that there is a relation between data extracted from partially trained machine learning models and the final score that they can achieve. This should allow for the creation of a hyperparameter optimisation algorithm that can generate a large number of partially trained models and pick the most promising ones from them. By doing that, the algorithm will circumvent the need for full training of models and theoretically, produce results faster and more efficiently.

In the scope of the project, the optimisation of two different types of machine learning algorithms will be discussed. The initial discussion will be focused on binary classification using gradient boosting algorithms. Due to the relatively smaller complexity of this approach, compared to the second part of the project, this will serve as an introduction to the proposed algorithm. The second part of the project focuses on neural networks and more specifically on techniques for neural architecture search. The aim would be to show that the algorithm is able to generalise well and solve related problems from different domains.

Chapter 2

Background

Before diving into a thorough investigation and explanation of the proposed algorithm, the key aspects of the machine learning field should be discussed. The major topics of interest that are going to be covered, in regard to algorithms, are gradient boosting methods [8] and neural networks [9]. This is followed by a close look into the details of the most widely used approaches in hyperparameter optimisation and contrasting and comparing the strategies that they employ. The nature of neural architecture search will also be explained, including the similarities to hyperparameter optimisation. This will give us a strong foundation of knowledge, from which the newly proposed ideas should follow naturally.

2.1 Machine Learning

Machine learning is a sub-field in artificial intelligence which aims to produce algorithms that are able to automatically discover solutions to problems. It is characterised by the fact that the rules of the algorithms are not explicitly programmed but are instead learned from the data [10]. The learning is done through training, that represents a process in which a model is built by processing records from a training dataset and updating the internal values of that model. The final model should be able to generalise well for new data points and is used to predict on previously unseen data points [11]. Machine learning is typically divided in three separate sub-categories: supervised learning, unsupervised learning and reinforcement learning.

Supervised learning is used to learn the relationship between data points and their corresponding labels [12]. The data points can represent any type of information and can be used to predict the labels, which can be seen as the outcomes. For example, the data points for a weather forecast problem can be the information about the temperature and humidity

at any given day, and the labels can show what the actual weather was on this specific day. Using that data, supervised learning can be used in order to predict the weather of new days, given that the temperature and humidity for those days is known. It is a powerful tool for predictive analysis, regression problems and classification problems.

Unsupervised learning does not use labels and it is used to find similar patterns in the datasets. It can be used in order to cluster datasets into groups that exhibit similar characteristics. This type of learning is particularly useful in anomaly detection, because it is able to separate data points that are not similar to any of the others [13].

Reinforcement learning is concerned with learning optimal solutions to problems which require making decisions about selecting the most optimal actions in a given environment. It is used in the cases when it is required to develop a self learning algorithm for finding the best possible set of steps for a given problem [14]. This form of learning has been the most widely used approach behind the recent advancements in self-driving cars and smart robots that are able to beat human experts in games like chess [15] and ‘Go’ [16].

2.2 Ensemble methods

One common approach for building machine learning models is the use of ensemble of weak learners, for example, decision trees [17]. Each decision tree on its own is not capable of producing predictions with high quality, however when combined together, multiple trees can create extremely accurate models. The two most common ways of acquiring a single score from multiple trees are bagging and boosting [18].

Bagging can be seen as a method in which multiple decision trees are trained and then used to predict on a single data point. Their individual predictions are then combined, typically by averaging, and thus the final score is obtained [19]. The *random forest* algorithm takes it a step further by training the decision trees on only a subset of the available features, in order to avoid a high correlation between the decision trees [20].

Boosting also uses multiple decision trees, however instead of training them all on the features of a data point, only the first one is trained against the actual data and all others are trained sequentially on the residuals of the decision tree before them. This allows every new tree to minimise the error that the final model will have. This approach is not only capable of producing good models, but it has also achieved state-of-the-art performance on some tasks [21].

2.3 Neural Networks

Artificial neural networks are an approach in machine learning that was inspired by the biological structure of the human brain. They use simplified, digital version of neurons and synapses to process information and extract patterns and meaning from data [9]. The neurons are usually called nodes and are separated into different layers and connected by the synapses. The first and the last layer of the networks are known as input layer and output layer respectively. The layers between those are called *hidden layers* and a network can have anywhere from zero to hundreds of them, depending on its architecture. The value of each node is calculated by first summing the values of all nodes to which it is connected, multiplied by the weights of the synapses that connects them. The total sum is then passed through an *activation* function, which is just a non-linear transformation. The most widely used activation functions are the *Sigmoid* function, the *Rectified Linear Unit (ReLu)* function and the *tanh* function [22].

Deep neural networks refers to neural networks that have multiple hidden layers. Because they typically contain large number of layers, and thus require more calculations to be done, they have only recently started to grow in popularity with the increase in computational power [23]. Despite that, researchers have already managed to use their potential and have developed state-of-the-art solutions for not-trivial problems, such as image recognition, speech recognition and many more areas [15].

Convolutional neural networks (CNNs) are a type of deep neural networks commonly used in image processing and image recognition problems [24]. They are more suitable that traditional deep neural networks, because images are high-dimensional vectors and would require very high amount of nodes and synapses to be represented using a traditional approach. In order to reduce the dimensionality of the data but retain the information, special *Convolutional layers* are used. They consist of multiple filters with typical size of 3x3, that are slid across the input image and the dot product between the filters and the image are calculated [25, 26]. Other types of layers that are commonly associated with convolutional neural networks are: *Dropout layers*, *Batch Normalisation layers* and *Pooling layers*.

2.4 Hyperparameter optimisation

Hyperparameter optimisation is the process of finding the best possible configuration of parameters for a given algorithm, using which, it achieves the best score. It is represented by the equation:

$$\lambda^* = \underset{\lambda \in \Lambda}{arg\,min}\, f(\lambda) \quad (1)$$

In equation (1), $f(\lambda)$ is an objective function, typically calculating the mean squared error, and it is to be minimised. The parameter λ belongs to the search space Λ and it is a single hyperparameter configuration. The optimisation algorithms are trying to find λ^*, which represents the optimal hyperparameter set for which the function has the lowest value [27].

A simple and intuitive approach would be to just define a set of hyperparameter configurations and test all of them. This is the idea behind *Grid search*, which defines the hyperparameter configurations in a grid and then tests all possible configurations from it [28]. This is a form of a brute-force technique that can work for problems with limited number of parameters, but is typically infeasible for optimisation of algorithms with many parameters. It is also highly inefficient to test all possible configurations, compared to using 'smarter' approaches.

Random search is another brute-force algorithm which searches blindly for configurations, but instead of using a predefined grid, it samples, with uniform probability, different hyperparameter configurations from the entire search space [29]. It is particularly useful in situation when no assumptions can be made about the hyperparameter space and thus, the full space must be considered. The strength of this approach is that it focuses on exploration of the search space and while it is not guaranteed, due to its random nature, it usually finds hyperparameters that are close to optimal in reasonable time.

Bayesian search is a more complex approach compared to the ones discussed so far. The main concept behind it is the creation of a surrogate model, based on previous evaluations, and using it in order to predict the next best configuration to evaluate. It uses *Gaussian process regression* to quantify the uncertainty. The decision of where to sample next is made using an acquisition function applied to the surrogate [30, 31].

It is worth mentioning that it is also possible for people with experience in a specific domain, known as human-experts, to try and find the optimal hyperparameter configuration by hand. They usually rely on a try and error approaches, augmented by their experience and intuition, to choose a well-performing configuration This is typically impractical and in many cases can be outperformed by automatic systems [32].

2.5 Neural architecture search

Neural architecture search (NAS) is a form of hyperparameter optimisation, which aims to produce architectures of a neural networks that achieves high performance. One major difference is that the number of hyperparameters needed to specify an architecture is much bigger. This is due to the fact that the the neural network architectures contain multiple hidden layers, all the confections between those layers and additional hyperparameters for each individual layer. On top of that, the order of the layers is also important. Because

there is not upper bound to the number of hidden layers, the search space is effectively infinite. Because of that, it is infeasible to use random search, and more sophisticated approaches are used instead [33].

Many of the advanced methods are based on Reinforcement learning. In that approach, a recurrent neural network controller is used in order to generate variable-length strings that specify the architectures. The resulting architectures are trained and evaluated on the data and the controller is updated [34]. Even though the approach has been successfully used by companies such as Google, it requires high computational power and it is time-consuming.

Another popular approach is the use of evolutionary algorithms. They are inspired by biological evolution and more specifically, by natural selection. For this type of algorithms, a population of size N is created at random, and each individual is trained and evaluated. A new generation is then created from the previous, using different techniques such as crossover, cloning and mutation. For crossover, two parents are selected and parts of them are combined together in order to create an offspring. Cloning represents copying of an individual from the current generation directly to the new generation. Mutation uses random probability in order to change parts of the individual and can be applied either as a standalone operation, or as a final operation after one of the previous two. The selection of parents for crossover or individuals for cloning can be done using a selection algorithm based on the evaluated fitness of the individuals [35]. A major drawback of this technique for NAS are high computational costs related to the full training and evaluation of multiple neural networks for just a single evolutionary generation

It is important to note that in hyperparameter optimisation problems, including NAS, finding models that can outperform the current state-of-the-art models, by even a fraction of a percent, is still a considerably hard problem. Furthermore, machine learning models usually operate on billions of records, so even small improvements in performance result in significant gains that are extremely valuable.

Chapter 3

Proposed algorithm

The discussion in this chapter is focused around a new algorithm for hyperparameter optimisation. It starts with an explanation of the main principles on which the idea for the algorithm is based, followed by a discussion on the feature engineering that has been done. In the later sections, the different techniques for performance improvement that has been used are discussed, inducing the reasons behind their choice. The chapter concludes with an explanation of the selection strategy for choosing the most promising machine learning models from the predicted models.

3.1 Core idea

As seen from the previous discussion about the different approaches in hyperparameter optimisation, most of the existing algorithms treat the model as a black box and are only interested in the final score that it produces. The core idea behind the proposed approach is to instead, extract all available information from the model while it is being trained and use it to create features that would allow for an early prediction of the final score of that model. The hypothesis is that the final accuracy of the machine learning models can be predicted from their internal state in the early stages of training. This is based on the idea that high performing models should have some internal differences, compared to low performing models, that can be used to distinguish between the two.

In order to provide evidence for that proposition, Figure 1 shows a plot of the validation accuracies over 20 epochs of training, for two different models. The same training and validation datasets have been used in order to train and score the models, and the only difference between the two are the hyperparameters used. The top model is a high-performing one, achieving a final validation accuracy of about 85%, whereas the bottom model has a

lower final performance, with accuracy just above 67%. There is a clear distinction between the starting accuracies of the two models, and on average, the red model improves about 0.5% faster than the blue model between epochs.

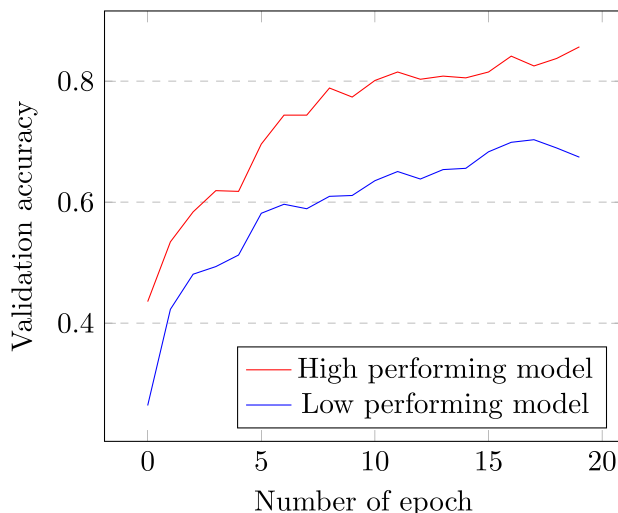

Figure 1: Comparison of the training curves of a high performing model and a low performing one. The difference between the two is present from the start, supporting the theory for using early prediction to identify the better performing model.

This gives an interesting insight into the nature of hyperparameter optimisation and shows the importance of choosing a good hyperparameter configuration. Furthermore, it supports the theory that there might be some characteristics early in training that are correlated with the final accuracy of the models.

Based on our observations, we propose a novel approach for hyperparameter optimisation, called *Predictive hyperparameter optimisation* (Predictive hyper-opt or PHO for short), which creates a pool of partially trained models, predicts the most promising ones and evaluates them, in order to find the top performer. The following pseudo code outlines the

proposed algorithm:

Parameters: H – Evaluation history
T – Total number of models to train
P_m – Pool of partially trained models
λ – Hyperparameter configuration
Λ – Hyperparameter search spaces
E – Number of epochs for fully trained models
ϵ – Number of epochs for partially trained models
T_f – Number of models to fully train for training set
T_p – Number of top predicted models evaluated

$H \leftarrow \emptyset$;
$P_m \leftarrow \emptyset$;

```
// Building a pool of partially trained models
```
for $i \leftarrow$ *1 to* T **do**
  select λ from Λ;
  create model μ with hyperparameters λ;
  train model μ for ϵ epochs;
  recording training loss at each iteration;
end

```
// Continue training some of the models until trained fully
```
for $i \leftarrow$ *1 to* T_f **do**
  select a partially trained model at random from P_m;
  train to E epochs;
  calculate and record final validation accuracy in H;
  remove current model from P_m;
end

```
// Build linear regression and make predictions
```
let x be a dataset containing the validation accuracies of the fully trained models;
let y be a dataset containing the validation accuracies of the partially trained models;
create linear regression model R using x;
predict on y using R;

```
// Evaluate predicted and produce final selection
```
for $i \leftarrow$ *1 to* T_p **do**
  get i-th best predicted model from R;
  train fully and calculate its final validation accuracy;
  record validation accuracy in H;
end

return configuration with highest validation accuracy from H;

Algorithm 1: Predictive hyperparameter optimisation

As shown in Algorithm 1, the process is split into 4 different steps. The first step is to create a pool of partially trained models and record their training loss in order for it to be used as a feature for predicting the final accuracies of the models. The second step selects partially trained models from the pool and trains them fully. This allows for the creation of a labelled dataset, where the partially trained loss is the feature and the final validation accuracy is the label. Using that, a linear regression model is built and it is applied to all partially trained models that are left in the pool. The models with highest predicted performance are then fully evaluated as well, and the top performer is selected from both them and the fully trained models that were trained in step 2.

3.2 Feature engineering

As seen from the previous section, the internal parameter that was used as a predictor for the final validation accuracy was the loss at a specific epoch. This feature was chosen initially, because it does not require additional processing in order to be obtained, and analysis showed that there is correlation between the training loss and the final validation accuracy.

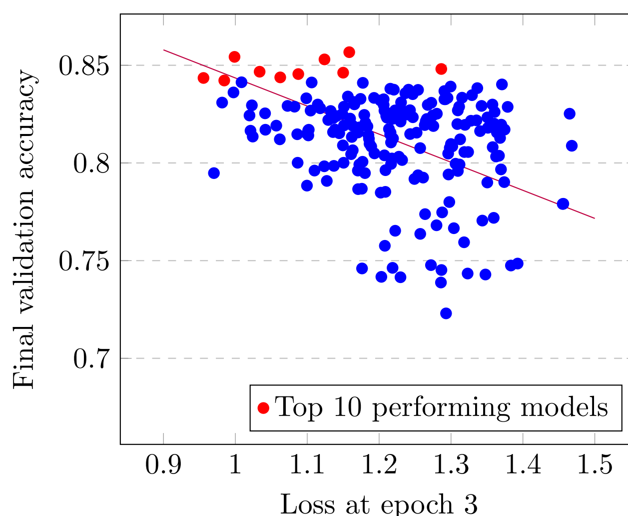

Figure 2: Correlation between the loss at an early epoch and the final accuracy of models. While the correlation might not be very strong, it still holds more predictive capability than choosing models at random.

However, as seen from Figure 2, the correlation is weak, with a correlation coefficient of -0.32. Experiments with building linear models for the loss at different epochs showed

that some epochs have stronger correlation with the final validation accuracy of the models. Unfortunately, the epochs which had loss with stronger correlation, were not always the same for all the models in the pool. This was a prime indicator that feature engineering should be used in order to create a more stable feature that will be able to generalise over the full pool of models.

The first approach was to use the losses at multiple epochs as predictors. Although an interesting idea, this resulted in highly correlated features which worsened the performance of the linear regression model. The next step was to try and average the values instead of using them as separate features. Unfortunately, this also led to a poor performance of the linear regression and to predictions that had low validation accuracies. It is worth mentioning that experiments were also done using the training accuracy instead of the training loss, as well as using combinations of the two, however this resulted in no noticeable differences as the two values are also highly correlated.

A major point of interest in model training, is the rate of improvement of a model. This proved to be a suitable new feature and it is calculated using the training losses of the models at different epochs. The calculation is done by computing the difference between the losses at the ends of an epochs range. In other words, if we are interested in the range from epoch 2 to epoch 0, we calculated the new feature as the difference between the loss at epoch 2 and the loss at epoch 0. From this point forward, the new feature will be refereed to as 'Improvement(n, m)', where n is the rank of the epoch at the upper bound of the range and m is the epoch at the lower bound. For example, Improvement(2, 0), denotes the difference between the loss at epoch 2 and the loss at epoch 0, for a specific model.

Shown bellow, is a plot of the correlation between the new 'Improvement' feature and the final validation accuracy.

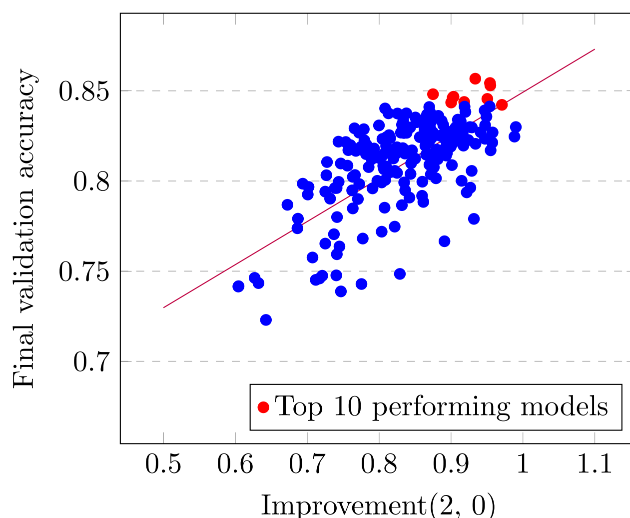

Figure 3: Correlation between the improvement of models from epoch 0 to 2 and their final accuracy. The newly engineered feature achieves a much stronger correlation and generalisation and can be created with only a simple computation.

As we can see from Figure 3, there is a much stronger correlation between the new metric and the final validation accuracy, with a correlation coefficient of 0.55. The increased stability and generalisation, coupled with the fact that it is easy to calculate, makes it a suitable choice for a predictor for the final validation accuracy.

3.3 Linear regression

The choice for building a simple linear regression model for the predictions of the predictive hyperparameter optimisation algorithm, stems from the fact that it is a straightforward approach, which is easy to explain and computationally cheaper to build and train, compared to more complex machine learning algorithms that can be used for regression. The following plot shows a linear regression model built from 10 initial models and used to predict the accuracies of 190 other partially trained models.

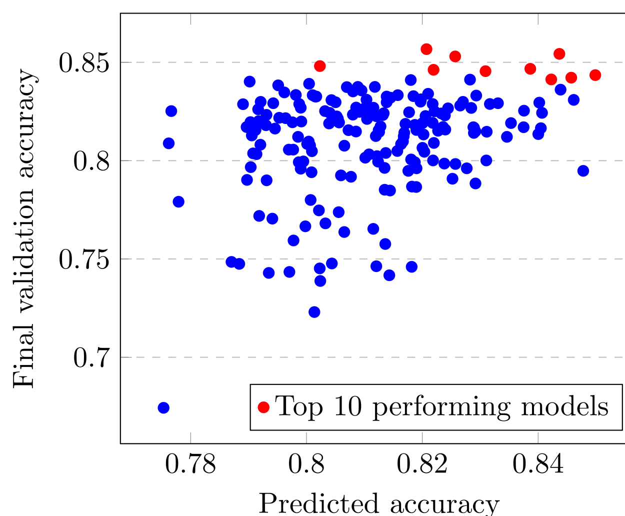

Figure 4: Models, plotted based on their predicted accuracy and their actual accuracy. Four of the top ten predicted models, correctly identify models from the list of top performers.

All models have been trained on the same train and test datasets. They have been initialised with different hyperparameter configurations, sampled without replacement from the full hyperparameter search space. On Figure 4, the data points furthest to the right represent models that have achieved the highest final accuracies. The data points that are near the top, represent models that have been predicted from our linear regression model to have the highest accuracies. Looking at the plot, we can see that the second best model is in the top 6 predicted. Furthermore, almost all of the top predicted models are close to the top performance.

It is worth noting that additional experiments were also preformed using other machine learning techniques. For example, an attempt was made to train a neural network as the regression model, however its accuracy suffered from overfitting and the training times increased drastically. Furthermore, it introduced additional complexity, namely the tuning of its hyperparameters.

3.4 Clustering

One problem encountered when testing, was that the models selected for full evaluation and used to build the training dataset were not always representative of the full population of models. This was due to the fact that they were being selected at random and it was

not possible to avoid cases in which only one type of models was being selected (Only high performing models or only low performing models). Building a linear regression model without samples from all types of models was leading to poor general performance of the algorithm and it was making it unreliable.

In order to ensure that a wide range of different types of models was being selected, a clustering model was introduced. Algorithms for clustering do not need labelled data, but instead try to separate the data into different categories (clusters) based on their features. Using a simple K-means clustering algorithm, which builds clusters by placing similar data points next to each other, a clustering model was created, shown in Figure 5 bellow.

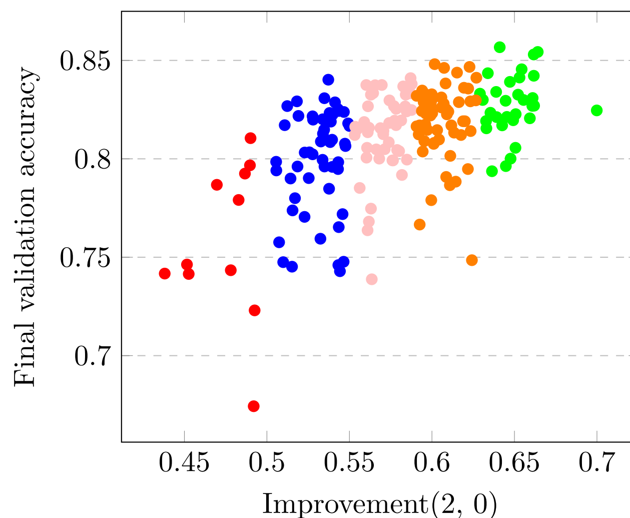

Figure 5: Clusters of models created based on their Improvement(2, 0) metric. The clusters with higher improvement generally contain models with higher final accuracy. Sampling each of the clusters in order to build the dataset for the linear regression, ensures that the linear regression model will have examples representative of the full population.

The clustering model is trained on the Improvement(2, 0) metric of a full pool of partially trained models. After that, it is used to obtain models from each cluster that are then fully evaluated and used to build the training set for the linear regression model. The models that are being selected from each cluster are selected at random, however, given that the clustering algorithm is working correctly, the random choice should not affect the final accuracy drastically.

Figure 6, bellow, shows a linear regression model created under the same conditions as the one in Figure 4, but the training models were selected with clustering.

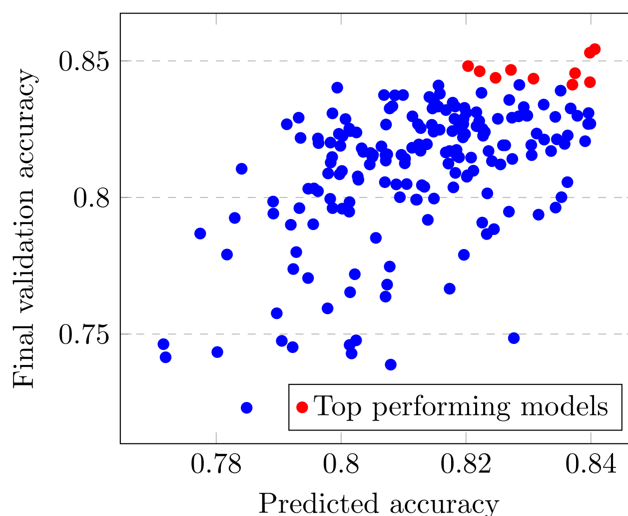

Figure 6: Models, plotted based on their predicted accuracy and their actual accuracy, with the training data selected with clustering. The top predicted model is the actual highest performing model from the pool.

It is clear that the approach produces much more accurate regression models. The top predicted model is the actual highest performing model from the pool and the second best is in the top 5 predicted. Furthermore, this approach achieves a higher stability when creating the linear models because it is consistently picking models from every type, in order to build the training dataset, instead of relying on random chance.

3.5 Selecting predicted models

The final point of discussion, in regard to our newly developed algorithm, is the strategy for selecting which of the predicted models to fully evaluate in the end. Even though it is not as complex as the previously discussed parts of the algorithm, this is one of the most important steps, because it is directly responsible for the selection of the models that are to be considered.

At the early stages of development, the algorithm was only considering the top predicted model. This approach was relying heavily on the premise that the linear regression model will be able to always predict the top model, which was not always the case and led to unstable performance. It was clear that a more sophisticated method for selection was needed to be developed, however, it was also important to not jump straight into complex approaches before thoroughly testing the simpler ideas.

Naturally, the simplest solution was to consider the top N models instead of only the top 1. Given that the linear regression, in combination with the clustering model, were consistently placing the actual top performing model in the top 15-20 predicted models, it was only natural to experiment with evaluating the top 20 predicted models. This approach quickly resulted in the algorithm being able to find the top performer in almost all of the cases. A reasonable heuristic that was established from conducting experiments with the algorithm, was that the number, N, of top models that should be evaluated for best performance is equal to 10% of the total number of models in the pool.

Even thought the previously described approach for selection is the one that was found to be most successful, it is worth mentioning the other strategies that were tried. Some of them were only considered but were not tested, due to time constraints.

The idea behind incremental selection was to evaluate only one model at a time and use it to retrain the linear regression model. The next model was then selected from the prediction of the newly trained linear regression model. This method proved to be inefficient without significant return in performance. Furthermore, when clustering was being used, it was actually skewing the distribution of model types and was making the performance worse.

Bottom elimination was a similar approach that was, again, incrementally improving the linear regression model, but it was also eliminating the bottom M models with worst predicted performance. Even though this approach has been proven to work in optimisation problems, the small amount of test that were performed with it were unsuccessful. It will be interesting to revisit this method in future works, because it was not explored thoroughly due to time constraints.

The final method that was also not explored due to time constraints, was the predicted rank proportionate selection. The main idea of this approach was to evaluate the top N models by selecting them at random, with each having a probability of being selected, proportionate to its predicted rank or its predicted accuracy. That means that models with higher predicted accuracy will have a higher chance of being evaluated but it will not be guaranteed. In theory, it will allow for the recovery of some of the models that were wrongly predicted to have a lower accuracy compared to their actual performance. This is also an approach that would be worth investigating in the future, because it is widely used as a selection method for evolutionary and genetic algorithms.

Chapter 4

Evaluations

The newly proposed approach has been evaluated against multiple different datasets and applied to different problems, in order to measure its performance. This chapter gives a detailed description of all the datasets that have been used, and discusses the different problems to which the algorithm has been applied. Additionally, detailed comparison is made between the performance of random search and the new algorithm. The chapter also explains the difficulties that were encountered while trying to evaluate and compare machine learning models.

4.1 Datasets

The choices of the datasets are directly related to the project development timeline and the project focus at the different stages of development. Brief description of the project timeline and iterations is given where necessary, in order to give context for the choices that have been made.

The first iteration of the project was focused on hyperparameter optimisation for gradient boosting machine learning models. The aim was to create a prototype of the algorithm and to evaluate it on models that are not too computationally heavy, allowing for more rapid development. It was chosen to work on a binary classification problems, due to their relatively lower difficulty. Two datasets were selected: a smaller one, for faster training and experimentation and a larger one for final validation of the algorithm.

The first dataset that was selected, was a bank marketing dataset. It contained data for a marketing campaign with the goal of predicting whether or not a customer will subscribe to a term deposit [36]. The data consisted of 17 attributes and a total of 45,211 data points.

The second dataset was for solving a physics problem, more specifically, whether or not a process will produce a Higgs boson. It contained data produced using Monte Carlo simulations with 28 attributes and 11,000,000 data points [37].

The second iteration of the project was focused on applying the predictive hyperparameter optimisation algorithm for neural architecture search for convolutional neural networks. Convolutional neural networks are best suited for solving image classification problems. Because of that, the two datasets that were chosen for this iteration were for multiclass image classification.

The first choice was the MNIST dataset [38], due to its major popularity and wide use in the field. It is a dataset that contains images of handwritten digits with 70,000 examples. The size of the images is 28x28 and they are in black and white. It is a relatively small dataset that is easy to use, well documented and it is well suited for initial testing and rapid prototyping of the convolutional neural networks.

The second choice was the CIFAR-10 dataset. It contained images of 10 different classes with a size of 32x32. The total number of data points is 60,000 and it is a more challenging dataset because the images use colour [39].

4.2 Baseline performance

The baseline performance against which the newly developed algorithm has been compared to, has been obtained by using random search and recording its performance. For the gradient boosting models, it was possible to train and evaluate models when needed, however, for the convolutional neural networks it was extremely time consuming to retrain and evaluate the models each time. Because of that, a real time comparison was performed for the gradient boosting, whereas for the convolutional neural networks, the performances of all 200 models that were used were precomputed and the random search was simulated from them.

Two different approaches have been used in order to calculated the performance of the random search algorithm for the precomputed values. The first one is based on Monte Carlo simulations in order to approximate the performance and the second one is using combinatorics to calculate the performance. In both of the cases, the comparison between random search and predictive hyperparameter optimisation, is based on training the same number of models from the model pool. For predictive hyperparameter optimisation, all of the models are partially trained and some are never fully trained. In order to compare them to the fully trained models from random search, the total sum of epochs between all trained models is kept the same between the two algorithms. For example, if predictive hyperparameter optimisation has trained 10 models fully to epoch 10 and 100 models

partially, to epoch 1, we compare it with 20 (10 + 100 / 10) models from random search. The general formula is:

$$F + P/T \tag{2}$$

Where F is the number of fully trained models. P is the number of partially trained models. T is the total number of epochs to which the fully trained models have been trained. It is worth noting that this is just an approximation that might not be consistent with the truth depending on the model pool, because different models might have different training times.

An experiment was performed by recording the training times for all models in the pool and using that to keep random search and predictive hyperparameter optimisation consistent. It was discovered that for the pool of models that has been described in this report, the training times were very consistent between models and thus, the sum of epochs was appropriate to use and it was faster to compute.

The idea behind the Monte Carlo simulations approach for comparison is to first calculate how many models have been trained by the predictive hyperparameter optimisation algorithm, using the method that was described in the previous paragraphs, and to sample without replacement the same number of precomputed model accuracies. The final performance of that episode is then the maximum value found in the sample. The Monte Carlo simulation is then performed for large number of episodes (100,000) and the values of all episodes are averaged. The averaged value is the approximation of the performance of random search. While this approach is straightforward to implement and achieves reasonable approximations, it is extremely time consuming and computationally expensive.

The second approach is purely mathematical and it uses combinatorics in order to calculate the exact accuracy that can be achieved by random search. In simple terms, the problem can be described as such: Given a total of N values, what is the average value of an algorithm that samples n values without replacement and returns only the highest value from the sample?

Lets assume that the values N are ordered from lowest to highest in a list. Because the sampling is without replacement, that means that lowest value that this algorithm can have is the value of the n$^{\text{th}}$ element in the list. Furthermore, that value can be obtained in only one way - by selecting the value n and selecting $n-1$ values that are lower than n. And because there are exactly $n-1$ values bellow n, that means that there is only one way for them to be selected. This is due to the fact that the order of selection is not important, as we are only interested in the element with the highest value from the sample. In mathematical notation, the selection of the $n-1$ values can be expressed as $\binom{n-1}{n-1}$. The selection of the value at place $n+1$ by the algorithm, follows the same reasoning. In order for the algorithm to select it, that value must be the highest in the sample. Because the elements are ordered by value and the sampling is without replacement, the algorithm must

select the value at position $n+1$ and another $n-1$ number of values from the values up to and including n. The total number of unique samples that can be obtained by picking $n-1$ numbers from the n values, will be equal to the number of different choices by which the value at position $n+1$ can be returned by the algorithm. In mathematical notation, that can be written as $\binom{n}{n-1}$.

From the previous explanation, the general case can be deducted. For an element in the ordered list, the number of different ways that it can be selected by the algorithm is equal to $\binom{n-1+i}{n-1}$, where i is the number of elements that are above n + 1, but bellow the element that we want the algorithm to select. If we then take the average of the weighted sum of the values of the elements and the number of ways that they can be selected, we end up with the average performance of the random search algorithm.

$$\frac{\sum_{i=n}^{N} \binom{n-1+i}{n-1} \cdot L(n-1+i)}{N} \tag{3}$$

In Equation 3, $L(x)$ is a function that returns the value of the element at position x.

4.3 Binary classification

One of the major challenges in regard to binary classification was that the datasets were imbalanced. This led to classifiers that learned to predict the class that had more data points, which was not the desired behaviour. In order to resolve that problem, the scoring needed to be changed as it was not enough to simply look at the true positives and true negatives. The approach that was taken for scoring the models was to create their receiver operating characteristic curves and find the area under those curves. This new metric was much more suitable and it was displaying the actual performance of the models.

The dataset imbalance, however, was also affecting the training of the models and they were having a low final performance. The solution to this problem was to use undersampling or oversampling. Both techniques were tested. In oversampling, additional data points are created synthetically for the under-represented class before training. In undersampling, some of the data points from the over-represented class are removed until the two classes have the same number of observations. Because the performance of both methods was almost identical, the undersampling technique was chosen as it required less computational power.

The comparison between predictive hyperparameter optimisation and random search was performed by running both algorithms 100 times and comparing the final results that they have obtained. Figure 7 (reproduced from the poster created for the project), shows the results of the experiment.

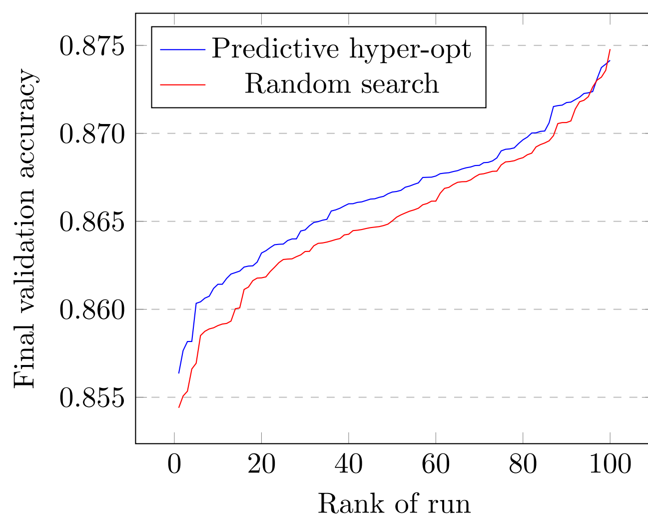

Figure 7: Compassion between the final accuracies of one hundred models, chosen by predictive hyperparameter optimisation and random search respectively.

The results for both algorithms are ordered in ascending order by the accuracies. The plot shows that for predictive hyperparameter optimisation, half of the selected models have achieved performance above 86.7%. In comparison, only 36 of the models chosen by random search has achieved performance above 86.7%.

4.4 Image recognition

For the image recognition problem, the hyperparameter optimisation algorithms were trying to find the most suitable architecture. Different CNN architectures were reviewed and the one that was chosen for the experiments was 'All Convolutional Net' [40]. It was chosen due to the fact that it achieved reasonable performance and it was faster to train and evaluate compared to other convolutional neural networks that had more layers. The experiment was performed on the CIFAR-10 dataset because it is more complex and data augmentation techniques were used for improving the training performance.

In order for the algorithms to test different architectures, a special module for generating random architectures was created. This module takes a base architecture and applies different types of mutations to it, in order to produce new architectures. The created architectures are stored in a history in order for the module to not repeat them. The mutations that the module can apply are duplicate, remove, or augment parts of the

architecture. The probabilities of the mutations were set so that the average size of the architectures stayed the same.

For the experiment, a total of 200 models were trained using different mutated versions of the ‘All convolutional Net’. The models for each experiment have been trained to a different epoch. The statistics for the 200 models were recorded and were used for simulating random and predictive hyperparameter optimisation. Figure 8, shows the performance of the two hyperparameter optimisation methods side by side.

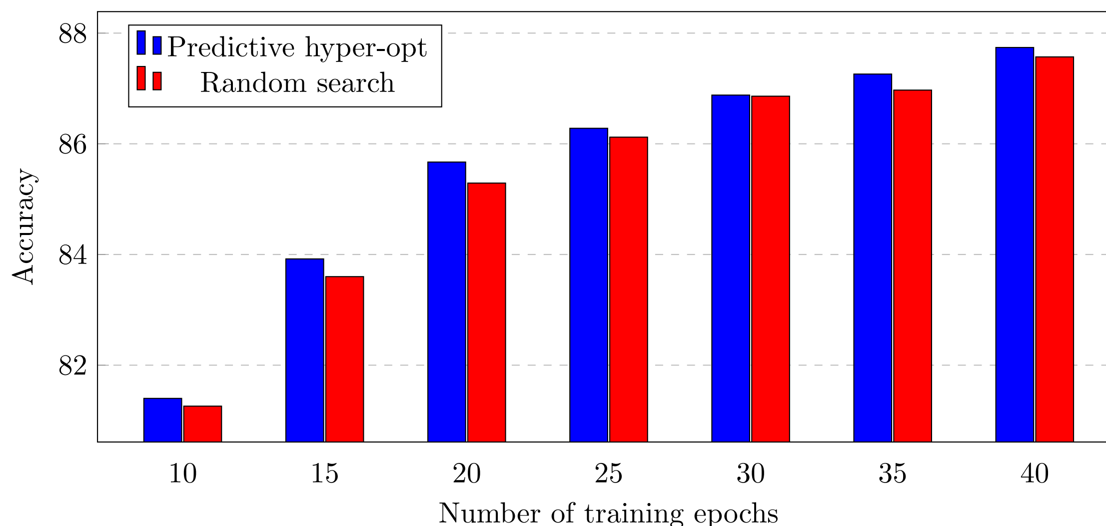

Figure 8: Comparison of predictive hyperparameter optimisation and random search, evaluated for models at different epochs, created using mutated versions of the ‘All convolutional Net’ architecture. Predictive hyperparameter optimisation achieves consistently better performance than random search.

Initially, random search was performed using Monte Carlo simulation. However, due to the higher computational time of this method, the performance was instead calculated following Equation 3. The difference in accuracy between simulation and calculation is negligible, at only 0.001%.

For predictive hyperparameter optimisation, a Monte Carlo simulation has been used with 100,000 episodes, in order to find a good approximation of its performance. Predictive hyperparameter optimisation is consistently better than random search, and in 6 of the 7 cases, finds the best possible performing model from the pool.

Chapter 5

Technology

Before moving forward, the programming tools that have been used in the implementation of the proposed algorithm must be explained. In addition, the chapter contains a review of the tools used for data visualisation and manipulations, the most popular platforms for cloud computing, and the testing and deployment tools used in the project development. Some of the difficulties that were faced when using the different technologies are also described.

5.1 ML tools

There is a wide range of machine learning tools that are being developed and that are being used in the field. The focus of this discussion will be based around libraries that are built for the programming language *Python*. Most of the selected libraries have been chosen based on their availability for open use, how well they are documented and how much support is provided by the development team or the community. For all of the tools, the discussion is based on their most current version as of the date of writing of this report.

5.1.1 XGBoost

XGBoost is a machine learning library for building, training and evaluating gradient boosting models [41]. It is highly efficient because its core is written in C++. Distributions are available for both Linux and Windows, however, under Windows it is important to use a 64 bit version of python 3.7 and in some cases it might be require to compile it directly from source. It is much easier to setup directly under Linux or to use with a different python distribution, such as Anaconda. The package has been used extensively to train the models

in the first iteration of the project and additional modifications have been made to it, in order to support the needs of the project.

5.1.2 Scikit-learn

Scikit-learn is a library that contains various different machine learning algorithms for both supervised and unsupervised learning [42]. In the scope of the project, it has been used extensively to create many of the different components that were needed for both the core algorithm and for the performed experiments. Bellow is a brief summary of the main sub-modules from the library that have been used in the project:

ML algorithms sub-module – Contains implementation of almost all of the most popular machine learning algorithms. It has been used in the project for building the linear regression and clustering models.

Hyperparameter optimisation sub-module – Contains implementations for both random and grid search that have been used in the early stages of testing and development of the project.

Imbalanced datasets sub-module – Contains algorithms for undersampling and oversampling that have been used in the first iteration.

Metrics sub-module – Contains implementation of different metrics for scoring machine learning algorithms. For example, the ROC-AUC score used for the experiments in the first iteration of the project.

5.1.3 Keras and TensorFlow

Keras is an application programming interface (API) for working with neural networks that can use different libraries as its backend [43]. In this project, TensorFlow has been used as the backend due to its wide popularity and support for training models on a GPUs [44]. TensorFlow has separate distributions for training on CPUs and GPUs, so it is important to select the correct one at installation. The process for training using a GPU is also more complex, requiring the installation of the CUDA toolkit, provided by NVidia, which is only available for specific types of GPUs.

Keras has been used for all of the model training in the second iteration of the project, and additional modules have been developed for allowing architecture mutations. It also has implementations for data augmentation that has been used, and a sub-module containing the different optimisers for the neural networks.

5.2 Data manipulation and visualisation

The tools for data manipulation have been an important part of all the stages of development. They have been used extensively, both in the development of the final product, and in the analysis of the data for the different problems. The data visualisation library that has been used mainly for analysis of results, however, some of the plots produced from it has been used in the initial report of the project.

There are many libraries that have been either partially used, or that provide the functionality, behind some of the functions of the main libraries discussed in this section. We will focus only on the main ones, as it will be unnecessary to cover all the dependencies in detail.

5.2.1 Pandas

Pandas is an open source library, used for data analytics, providing a wide range of optimised data structures [45]. It has been the core tool used in order to prepare the datasets by splitting them in different parts for training and testing. It supports encoding of categorical variables and easier selection of rows and columns of the datasets. Another part of the library that has been used extensively is the reading from, and writing to CSV files. The library represents the data using objects called *Data frames*, which are heavily optimised and have a lot of built-in functionality, such as sorting by column, applying a function to all elements in a row or a column, filtering and many more. Pandas has also been used for calculating the mean values of series of numbers and for random sampling.

5.2.2 Matplotlib

Matplotlib is the library that has been used for producing 2D plots for analysis [46]. It has an extensive collection of functions for producing a large variety of statistical diagrams including scatter plots, histograms, bar charts, box plots and many more. It allows for full customisation of the different elements of the diagrams, including colour, size, position, labels, markers, etc. The diagrams that are produced can be saved in all of the major file formats: png, jpeg, pdf, svg, etc.

5.3 Cloud computing

Cloud computing has recently gained massive popularity and have become more accessible. The benefits of using such service is that it allows for massive computational power when-

ever it is needed. For short experiments, it is cheaper to rent the hardware that you need for a day and perform your experiments, as opposed to buying, building and supporting you own high-power machine. Most of the major providers also allow users to try their platform for free, but the use is limited. The discussion of this section will cover the major providers and their platforms for cloud computing and will discuss their features and limitations in regard to this project.

5.3.1 Google Colaboratory

Google Colaboratory is a cloud based Jupyter Notebook that comes preinstalled with a wide range of libraries for machine learning [47]. It is suitable for running experiments fast and it also allows for a GPU to be used (this might be limited for some users). It is best suited for interactive running and it is highly discouraged to use the platform for long-ruining background computations. It easily integrates with Git repositories and Google Drive can also be used for data storage. One of the major drawbacks is that the user is limited to only using python trough the Jupyter notebook. In many cases, this leads to difficulties with resolving paths, executing custom scripts and using a debugger. It is well suited for collaborative work, but due to the nature of this project, this is not applicable in the current context.

5.3.2 Amazon Web Services

Amazon Web Services (AWS) is arguably one of the largest cloud computing platforms available at the moment [48]. It contains an extremely large collection of cloud services for different tasks, but the focus of our discussion will be only their Elastic Cloud Computing (EC2) service and their Simple Storage Service (S3). The EC2 service allows for the creation and use of virtual instances. It is highly customisable, allowing the user to specify the exact machine that they need, the operating system and the storage. It is also possible to create machines from custom images that are specifically suited for machine learning development. In general, it is harder to use compared to Google Colaboratory as it requires the user to connect to the remote machine via SSH, but it is much more powerful and is not limited to any specific tool or library. The simple storage service, provides a good place to store the data that is needed and the data that is generated by the machine learning tasks. It is supported directly by the Pandas library, which can read and write to files on S3, as easily as it can do on the local file system. The drawbacks of this platform are that it is expensive and the free trial allows for only extremely slow virtual machines to be used. It is possible to obtain free credits through their educational program, but in order to use them for their GPU instances, an additional request must be submitted through their support team, before access can be granted.

5.3.3 Microsoft Azure

Microsoft Azure is another major platform for cloud computing [49]. It is very similar to AWS in the way that it operates. The platform shares almost all of the features that have been described in the previous section for AWS. One difference that was encountered when it was tested during this project, was the fact that it was highly-unreliable. The creation of virtual machines was timing-out at random, the server was unresponsive and the error messages were unhelpful. It is unclear whether this was caused due to limitations of the basic account or due to the service experiencing a heavy load at the time.

5.4 Testing and integration tools

Testing and integration play a major role in any well developed project. The following sections will cover the main tools used in order to ensure that the project is kept in a working order. The final section covers the packaging and distribution of the project that has been employed.

5.4.1 PyUnit and Travis CI

PyUnit is a testing framework for python, used for writing unit tests [50]. It is the python version of the popular unit testing library JUnit, for Java. It allows for fast execution of tests and it is easy to use. The library has supports for stubs and mocks and it allows for function to be executed before or after a specific test or a group of tests. It is a powerful tool for testing and tests can be executed directly from the command line.

Travis CI is a service for continuous integration that can be used to build and tests projects from GitHub [51]. It is simple to use and it only requires a configuration file to be created in the repository in order to start working. The configuration files are written in YAML and specify the steps for the continuous integration. It has support for builds with different configurations depending on the branch, for ignoring specific branches of the repository and for an optional phase for deployment.

5.4.2 Packaging

The packaging of Python code is handled by the library Setuptools [52]. The library builds the package as source archives and a Python *Wheel*. The created package can then be uploaded to the Python Package Index (PyPI) and can be installed on any machine directly by using the package manager for Python – PIP.

Chapter 6

Product development

The main focus of this chapter is to explain in details the product development stage of this research project. Because of the nature of the project, the development can be roughly divided into two stages. In the first stage, most of the tools that have been developed were needed for the experiments that were being conducted. In the second stage, the findings of the research, have been developed into their own packages that allow for easier use. It is important to note that while the products developed in the second stage are production ready systems that adhere to all of the requirements that a user might have, the code that was written for the first stage is strictly experimental and should be treated as such.

6.1 Design and architecture

As mentioned above, the research products have been implemented into two separate Python libraries. Each library is a different product and has its own repository. The repositories have been made publicly available on GitHub and has been connected to Travis CI in order to have continuous integration. More information about the CI is given in the project planning chapter of this report.

6.1.1 Boar ML

Build Once And Run ML (BOAR ML) is the first library that has been created. The aim of this library is to allow the user to define a machine learning model once and then to compile it with different popular machine learning libraries. The idea behind it stems from the fact that as a researcher, if you want to test your neural network architecture with multiple different libraries, you have to use their specific syntax and create it separately for

all of them. The library also implements additional modules for applying mutations to an architecture. This is something that is not directly supported by any of the major providers of machine learning libraries and in many cases is very time consuming to implement, because of the limitations that they have. The library also has support for writing an architecture to a file and reading architectures from a file in a user readable format, another feature that is not present in all popular machine learning libraries. The advantages are that by using this library, the format will also be consistent across all the different models that the user creates, regardless of the underlying library that they choose to compile it with.

Boar ML uses a special type of object called *unit*. A single unit represents a block of layers from a neural networks. The units are defined by the user and the mutation operations are performed per unit, instead of per layer. The library defines the most popular types of layers that are used in convolutional neural networks. All of the layers inherit an abstract base layer that defines the common functionality between all of them. The layers implemented by Boar ML are used only for creating the representation of the architecture of the neural networks, and must be compiled using a special object called *builder* before use.

A base architecture class is provided that takes as parameters the number of input and output nodes of the network and creates a new blank architecture to which the user can add units. The units are created by adding layers to them and the constructed units can then be added to the architecture. The generator class contains the functionality for mutating architectures. Once the architecture has been completed, it can be fed into a generator object which will produce unique mutated offspring from it. The final step is handled by the builder module, which defines builder classes that can compile the original architecture into actual machine learning models using third-party ML libraries. The library is available at: `https://github.com/DobromirM/BoarML`

6.1.2 Predictive hyperparameter optimisation

The second library is the predictive hyperparameter optimisation library, which encapsulates the novel algorithm for hyperparameter optimisation. It is a simple and clean implementation of the main idea of the project that allows for execution with different parameters, such as the number of models in the pool, the number of partially trained models, the number of predicted models evaluated at the end, etc.

The models sub-module contains the implementations of the linear regression model and the clustering models that are used by the algorithm. There are also different interfaces for running the algorithm in real time or for models that have been precomputed. While running on precomputed models might not be useful for real case scenarios, it is particularly useful for short demonstrations and analysis. The library is available at:
`https://github.com/DobromirM/Predictive-Hyperparameter-Optimisation`

6.2 Additional tools developed

The additional tool that have been developed cover the tools that were created as part of the research but for different reasons did not end up in the final libraries. All of the code for them can be found under the *hyper-optimization* repository in the university GitLab website.

XGBoost modifier – A program that modifies the XGBoost library by inserting additional code in the training function, in order to collect the internal states of the models while they are being trained. This was only used in the first iteration of the project, because the second was about neural networks.

Utilities – This module contains functions for recording both the machine learning models and their internal states. It uses a specific folder structure, file names and formats, in order to provide a generic way for feeding the recorded internal data into the different machine learning models independent of the structure of the original datasets that they came from. In addition, the functions that are writing to files are making sure that no data is overwritten.

Data preparation – The module contains functionality related to the processing of the original datasets. It has functions for removing headers from CSV files, splitting data into features and labels based on a column index and prepossessing image data.

Models – The models sub-module contains implementation of some machine learning models that have been used only for experimentation. That includes a logistic regression model a simple neural network model used for regression and the XGBoost model used in the first iteration.

Jupyter – The Jupyter sub-module contains a Jupyter notebook that can be used to train models on Google Colaboratory. It contains commands for connecting to a Google drive, setting up environmental variables and executing scripts from the project. There are also additional branches that have been created in the repository for both AWS and Microsoft Azure. They amend some of the configuration in the project, in order to allow for easier execution on remote machines.

Hyperparameter optimisation – This sub-module contains implementation of some of the most popular algorithms for hyperparameter optimisation. It contains a base class that is inherited by all of the other search algorithms. All of the classes implement a *run* function that executes the search and some of the classes implement additional functionality specific to their type. The implemented methods are: grid search, random search, Bayesian search and the new algorithm for predictive hyperparameter optimisation.

Extractors – The functions defined under this sub-module are used to generate training and testing sets, from all of the files with internal states information, extracted by the utilities

package. They can also generate statistical data, such as lists with all the accuracies of a pool of models. It has been used heavily for generating data for analysis.

Analysis – This sub-module contains all the functions related to plotting data and simulating different types of hyperparameter searches based on precomputed values. It contains the implementation of the Monte Carlo simulations, for both random search and predictive hyperparameter optimisation and the implementation of the equation for calculating random search. In addition to that, it contains the functions for plotting the different statistics of a pool of models.

CNN training – There are two files for training of convolutional neural networks. They provide support for pausing and resuming training runs and allow for saved models to be trained for additional steps.

Autokeras – This sub-module contains an experimental setup for testing the library Autokeras. It has not been used extensively due to the number of different bugs that were encountered.

6.3 Testing

It is important to have a suite of tests that can be executed regularly, in order to spot bugs as early as possible and to give confidence that the project is working as intended. Because of that, both of the libraries that have been created for this project have been thoroughly tested. For each sub-module of the original libraries, a test sub-module has been created. The names of the test files correspond to the names of the files that are being tested and the test functions use the same names as the functions being tested with the addition of the word *test*. The full unit test suits provide full coverage of the packages and are designed to test for both trivial and non-trivial test cases, such as edge cases, values that result in errors and values that might cause numeric overflows.

The tests have been deliberately written without the use of any conditional operators and loops in order to minimise their complexity and avoid bugs in the tests. This is also done in order to increase the readability and to make them easier to change in the future. Whenever a test requires a long string of text, the text is placed in a separate text file in the resource directory, with the name of the text file being similar or even matching the name of the function that requires it.

In addition to the unit tests, end to end tests have also been created. They can be used as a tool for verification that the full system is in good working order, but can also be useful to new users, as examples of how the system can be used. Some of the end to end tests will be used in the demonstration of the project in order to provide the audience with a complete tour of the libraries.

Chapter 7

Project planning

The following chapter serves as a reflective review of the project planning and risk management techniques that were employed during the course of the project. It tries to provide an objective overview that can be both useful for putting the project timeline into perspective, as well as for avoiding mistakes in the future.

7.1 Overview

The project followed an agile software development approach that is particularly suitable for projects which have requirements that change over time. The main development process was split into two different cycles. Each cycle represented a different version of the project with each version having its own code-name.

The first version called *Labrador* was developed between October 2018 and December 2018. The main focus at that time was to establish a better understanding of the field and to produce a proof of concept version of the predictive hyperparameter optimisation algorithm. The project cycle was divided into 4 parts, each defined by a separate *epic* story on the online Kanban board that was used. Each epic story represented an overview of the job to be done for the two week period that it spanned. Each of them defined 4 tasks for the duration of the period, with each task having multiple sub-tasks to be completed. The first iteration ended successfully, with a working implementation of the first version of the predictive hyperparameter optimisation algorithm and a demonstration of it. In addition to that, a lot of new knowledge have been acquired about hyperparameter optimisation and the machine learning field in general.

The second version of the project, called *Border Collie*, had an original span similar to the previous version. The original time-frame was from January 2019 to April 2019. This

was increased to include the full month of May and the first week of June. The change allowed for more experiments to be conducted and for further development of the project. It was particularly helpful because of the long computational times needed for training convolutional neural networks. All of the additional work was facilitated by increasing the length of the epic stories that were defined for that iteration and by adding more tasks for each of them. It was a successful iteration, resulting in major increase of understanding in the field of convolutional neural networks and image processing. The final version of the predictive hyperparameter optimisation algorithm and the Boar ML library had also been created.

It is important to mention that weekly meetings have been held during the development of the project, allowing for regular reviews and changes of direction whenever necessary. This has been a major reason for the success of the project and the the timely completion.

Jira has been used as a virtual board for managing the different stories using tickets, separated into lanes. It has also been used in order to monitor the progress of the project and to generate cumulative flow diagrams. The diagrams have been used in the first demonstration of the project. *GitLab* repositories were used to store the code for both the main research code and the two libraries that have been created. The main repository contains additional branches for some of the experiments that were performed, while the main branch contains the latest changes. The two libraries that were created, were migrated to *GitHub* and the old repositories have been updated with links to the new ones. This was done in order to use Travis CI, which can be integrated only with repositories that are available on *GitHub*.

7.2 Risk management

Project risks are an important aspect of the project management process, regardless of the scope of the tasks. The main strategies that were used in order to mitigate risks were the weekly meetings on which a summary of the work done so far was presented, as well as a mutual agreement on what to work next. In addition to that, the *Expedite* column in Jira was used in order to flag any tasks that may require urgent attention and are blocking the further development of the project. The main risks were identified by looking at the future work and trying to identify the potential bottlenecks. This was extremely useful in order to maintain the momentum of the project and to deal with risks before they have become a major thread. Overall, the agile methodology, coupled with the consistent and timely work, proved to be suitable for completing the expected project.

7.2.1 Continuous integration

Following the practices from extreme programming, a system for continuous integration has been used to automate the process of verification after new changes are added to the project. This reduces the risk of regression and acts as an early indicator for spotting potential problems. The system that has been used is Travis CI, which works with GitHub. Once turned on for a specific repository, the system can allocate workers to execute an integration that is defined by the user. The workers can be allocated manually, by pressing a button to build the project, or automatically, after a successful push to the repository. The system requires YAML file named *.travis.yaml* to be placed in the root directory of the repository for each branch that needs to be tracked. This allows for different build instructions to be specified for the different branches.

The YAML files that have been used for testing the two libraries for this project, contain configuration that specifies that Python is used as the programming language, it specifies the version of Python that should be used. Additionally, the configuration file has an *install* stage, on which it is configured to install with PIP everything that is placed in the *requirements.txt* file in the root directory. Lastly, the continuous integration worker executes all the available unit tests in the repository and reports the status of the project as either *passing* or *failing*. The workers are configured to execute the integration on every push to the master branches of the repositories and the latest status of the build is displayed in the main page of the repositories. The system is also configured to send emails whenever a build starts failing, so that the problem can be resolved as soon as possible.

Chapter 8

Conclusion

In conclusion, the discussion started with the importance of hyperparameters and their optimisation. It continued with an explanation of the idea behind the newly proposed algorithm, called predictive hyperparameter optimisation, and discussed the reasoning behind it. It showed that using internal states of partially trained model can be a viable strategy, that can outperform random search on specific tasks. The technology that was used, the product development and the creation of the additional tools were also discussed. The architecture and design behind them were explained, and also the process that was used for planning and managing the project and the associated risks.

8.1 Future work

In the feature, it will be interesting to apply the proposed approach for optimisation to more and different types of architectures for neural networks. It will also be interesting to try it for additional machine learning problems and datasets, in order to evaluate its ability to generalise across different tasks. Another point of interest will be to further push the idea and to try and achieve even better performance with lower computational times. This could be done by incorporating ideas from Bayesian optimisation or by changing the linear regression model, used for predictions, with a more complex regression model.

Chapter 9

Bibliography

[1] K. R. Foster, R. Koprowski, and J. D. Skufca, “Machine learning, medical diagnosis, and biomedical engineering research - commentary,” *BioMedical Engineering OnLine*, vol. 13, no. 1, 2014.

[2] K. P. Murphy, *Machine Learning: A Probabilistic Perspective.* MIT Press, Sept. 2012.

[3] P. Langley and H. A. Simon, “Applications of Machine Learning and Rule Induction,” *Commun. ACM*, vol. 38, Nov. 1995.

[4] A. Buetti-Dinh, V. Galli, S. Bellenberg, O. Ilie, M. Herold, S. Christel, M. Boretska, I. V. Pivkin, P. Wilmes, W. Sand, M. Vera, and M. Dopson, “Deep neural networks outperform human expert’s capacity in characterizing bioleaching bacterial biofilm composition,” *Biotechnology Reports*, vol. 22, June 2019.

[5] P. Probst, B. Bischl, and A.-L. Boulesteix, “Tunability: Importance of Hyperparameters of Machine Learning Algorithms,” *arXiv:1802.09596 [stat]*, Feb. 2018. arXiv: 1802.09596.

[6] W. S. Sarle, *Neural Networks and Statistical Models.* 1994.

[7] T. Elsken, J. H. Metzen, and F. Hutter, “Neural Architecture Search: A Survey,” *arXiv:1808.05377 [cs, stat]*, Aug. 2018. arXiv: 1808.05377.

[8] R. E. Schapire, “The Boosting Approach to Machine Learning: An Overview,” in *Nonlinear Estimation and Classification* (P. Bickel, P. Diggle, S. Fienberg, K. Krickeberg, I. Olkin, N. Wermuth, S. Zeger, D. D. Denison, M. H. Hansen, C. C. Holmes, B. Mallick, and B. Yu, eds.), vol. 171, New York, NY: Springer New York, 2003.

[9] M. H. Hassoun, *Fundamentals of Artificial Neural Networks.* MIT Press, 1995.

[10] E. Alpaydin, *Machine Learning: The New AI.* MIT Press, Oct. 2016.

[11] E. Alpaydin, *Introduction to Machine Learning.* MIT Press, Dec. 2009.

[12] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Malaysia; Pearson Education Limited,, 2016.

[13] Z. Ghahramani, "Unsupervised Learning," in *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures* (O. Bousquet, U. von Luxburg, and G. Rätsch, eds.), Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[14] R. S. Sutton, A. G. Barto, and F. Bach, *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[15] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv:1712.01815 [cs]*, Dec. 2017. arXiv: 1712.01815.

[16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. v. d. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, 2016.

[17] T. G. Dietterich, "Ensemble Methods in Machine Learning," Springer, Berlin, Heidelberg, 2000.

[18] C. D. Sutton, *Classification and Regression Trees, Bagging, and Boosting.* 2005.

[19] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, Aug. 1996.

[20] A. Liaw and M. Wiener, "Classification and Regression by randomForest," vol. 2, 2002.

[21] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," tech. rep.

[22] R. Rojas, *Neural Networks: A Systematic Introduction.* Springer Science & Business Media, June 2013.

[23] I. Arel, D. C. Rose, and T. P. Karnowski, "Deep Machine Learning - A New Frontier in Artificial Intelligence Research," *IEEE Computational Intelligence Magazine*, 2010.

[24] S. Lawrence, C. Giles, Ah Chung Tsoi, and A. Back, "Face recognition: a convolutional neural-network approach," *IEEE Transactions on Neural Networks*, vol. 8, Jan. 1997.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), Curran Associates, Inc., 2012.

[26] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," 1998.

[27] M. Feurer and F. Hutter, "Hyperparameter Optimization,"

[28] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for Hyper-Parameter Optimization," *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[29] J. Bergstra and U. Yoshua Bengio, "Random Search for Hyper-Parameter Optimization," *Journal of Machine Learning Research*, vol. 13, 2012.

[30] P. I. Frazier, "A Tutorial on Bayesian Optimization," *arXiv:1807.02811 [cs, math, stat]*, July 2018. arXiv: 1807.02811.

[31] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," *Adv. Neural Inf. Process. Syst. 25*, 2012.

[32] M. L ' Opez-Ibáñez, J. Dubois-Lacoste, T. Stützle, S. Stützle, M. Birattari, and M. López-Ibáñez, "The irace Package: Iterated Racing for Automatic Algorithm Configuration," 2011.

[33] Y. Chen, G. Meng, Q. Zhang, S. Xiang, C. Huang, L. Mu, and X. Wang, "Reinforced Evolutionary Neural Architecture Search," *arXiv:1808.00193 [cs]*, Aug. 2018. arXiv: 1808.00193.

[34] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," *arXiv:1611.01578 [cs]*, Nov. 2016. arXiv: 1611.01578.

[35] G. J. van Wyk and A. S. Bosman, "Evolutionary Neural Architecture Search for Image Restoration," *arXiv:1812.05866 [cs]*, Dec. 2018. arXiv: 1812.05866.

[36] S. Moro, P. Cortez, and P. Rita, "A data-driven approach to predict the success of bank telemarketing," *Decision Support Systems*, vol. 62, pp. 22–31, June 2014.

[37] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for Exotic Particles in High-Energy Physics with Deep Learning," *Nature Communications*, vol. 5, p. 4308, Sept. 2014. arXiv: 1402.4735.

[38] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[39] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," p. 60, 2009.

[40] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, “Striving for Simplicity: The All Convolutional Net,” *arXiv:1412.6806 [cs]*, Dec. 2014. arXiv: 1412.6806.

[41] T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System,” 2016.

[42] “Documentation scikit-learn: machine learning in Python — scikit-learn 0.21.2 documentation.” Available at https://scikit-learn.org/stable/documentation.html (accessed on 26/05/2019).

[43] “Home - Keras Documentation.” Available at https://keras.io/ (accessed on 26/05/2019).

[44] “Api - Documentaion Tnsorflow.” Available at https://www.tensorflow.org/api_docs (accessed on 26/05/2019).

[45] “pandas: powerful Python data analysis toolkit — pandas 0.24.2 documentation.” Available at http://pandas.pydata.org/pandas-docs/stable/ (accessed on 26/05/2019).

[46] “Matplotlib - plotting 3.1.0 documentation.” Available at https://matplotlib.org/ (accessed on 26/05/2019).

[47] “Google - Colaboratory.” Available at https://colab.research.google.com/overview.ipynb (accessed on 26/05/2019).

[48] “AWS Documentation.” Available at https://docs.aws.amazon.com/ (accessed on 26/05/2019).

[49] “Microsoft - Azure Documentation.” Available at https://docs.microsoft.com/en-us/azure/ (accessed on 26/05/2019).

[50] “unittest — Unit testing framework — Python 3.7.3 documentation.” Available at https://docs.python.org/3.7/library/unittest.html (accessed on 26/05/2019).

[51] “Travis CI User Documentation.” Available at https://docs.travis-ci.com/ (accessed on 26/05/2019).

[52] “Welcome to Setuptools’ documentation! — setuptools 41.0.1 documentation.” Available at https://setuptools.readthedocs.io/en/latest/ (accessed on 26/05/2019).