BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

DESIGN OF ACCURACY PREDICTORS FOR CONVOLUTIONAL NEURAL NETWORKS

NÁVRH PREDIKTORŮ PŘESNOSTI PRO KONVOLUČNÍ NEURONOVÉ SÍTĚ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ŠIMON ŠMÍDA**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148346

Institut:	Department of Computer Systems (UPSY)
Student:	**Šmída Šimon**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Design of Accuracy Predictors for Convolutional Neural Networks**
Category:	Artificial Intelligence
Academic year:	2022/23

Assignment:

1. Familiarize yourself with convolutional neural networks (CNNs), databases of trained CNNs, and machine learning methods used to construct CNN accuracy predictors.
2. Propose a method for constructing classification accuracy predictors for CNNs. Select an appropriate task for the CNN, construct a dataset, define appropriate features as input to the predictor, and select at least three machine learning methods for predictor training.
3. Using existing libraries, implement and train the accuracy predictors from the previous item.
4. Experimentally verify the functionality (accuracy) and performance of the constructed predictors.
5. Evaluate the results obtained.

Literature:

- According to the instructions of the supervisor.

Requirements for the semestral defence:

- Items 1 and 2 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Sekanina Lukáš, prof. Ing., Ph.D.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	31.10.2022

Abstract

The aim of this thesis is to present a method of constructing accuracy predictors for convolutional neural networks (CNNs) by leveraging databases of trained CNNs (NAS-Bench-101) and employing machine learning (ML) techniques as performance estimation strategies. The study begins with a description of various ML methods used in building CNN accuracy predictors, followed by an in-depth examination of CNNs and databases of pre-trained CNNs. The proposed method involves selecting a suitable task for the CNNs (image classification), assembling a dataset, defining relevant features for the predictor input, and choosing five ML methods for training the predictors. Using existing libraries, the accuracy predictors are implemented, trained, and experimentally validated to assess their functionality and performance. The results are thoroughly evaluated, providing insights into the effectiveness of the proposed method and the potential for further refinement in the field of CNN accuracy prediction.

Abstrakt

Cieľom tejto práce je predstaviť metódu na konštrukciu prediktorov presnosti pre konvolučné neurónové siete s využitím databáz natrénovaných konvolučných neurónových sietí (NAS-Bench-101) a uplatnením techník strojového učenia ako stratégií na odhad výkonnosti. Štúdia začína popisom rôznych metód strojového učenia použitých pri budovaní prediktorov presnosti, nasledujúc preskúmaním konvolučných neurónových sietí a databáz predtrénovaných konvolučných neurónových sietí. Navrhovaná metóda spočíva vo výbere vhodnej úlohy pre konvolučných neurónových sietí (klasifikácia obrázkov), zostavení dátovej sady, definovaní relevantných príznakov ako vstup prediktorov a vo výbere piatich metód strojového učenia na trénovanie prediktorov. S využitím existujúcich knižníc sú prediktory presnosti implementované, natrénované a experimentálne overené na posúdenie ich funkčnosti a výkonnosti. Výsledky sú dôkladne ohodnotené, validované a poskytujú pohľad do efektívnosti navrhovanej metódy a potenciál ďalšieho vylepšenia v oblasti predpovedania presnosti konvolučných neurónových sietí.

Keywords

convolutional neural networks, neural architecture search, performance estimation strategy, accuracy predictors

Klíčová slova

konvolučné neurónové siete, vyhľadávanie architektúr neurónových sietí, stratégie odhadu výkonnosti, prediktory presnosti

Reference

ŠMÍDA, Šimon. *Design of Accuracy Predictors for Convolutional Neural Networks.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Lukáš Sekanina, Ph.D.

Rozšírený abstrakt

Hlavným cieľom tejto práce je navrhnúť robustnú metódu pre konštrukciu prediktorov presnosti pre konvolučné neurónové siete. Metóda je postavená na využití existujúcej databázy natrénovaných konvolučných neurónových sietí a pracuje s rôznymi technikami strojového učenia slúžiacimi na odhad výkonnosti neurónových sietí. Práca má prevažne experimentálny charakter, pričom jej výsledkom je komplexné porovnanie a zhodnotenie rôznych metód strojového učenia použitých pri konštrukcii jednotlivých prediktorov.

V úvode ide o podrobný popis metód strojového učenia, ktoré sú použité pri konštrukcii prediktorov presnosti neurónových sietí. Táto časť poskytuje detailné porovnanie biologických neurónov s umelými, ktoré sú využívané pri tvorbe neurónových sietí, s dôrazom na ich využitie v praxi, ale aj potenciálne obmedzenia. Ďalej nasledujú základné princípy lineárnej regresie, algoritmov random forest a xgboost, ktoré sú založené na vyhľadávacích stromoch. Na záver kapitoly je predstavený koncept grafových konvolučných sietí, jedným z komplexnejších modelov strojového učenia, ktorý je schopný efektívnejšie a presnejšie odhaliť skryté štruktúry v dátach.

V rámci práce boli tiež podrobne popísané architektúry konvolučných neurónových sietí, od jednotlivých typov vrstiev týchto neurónových sietí až po typické operácie, ktoré sa v nich používajú. Konvolučné neurónové siete, ktoré boli inšpirované mechanizmami vizuálneho kortexu ľudského mozgu, sú kľúčové pri detekcii, rozpoznávaní a klasifikácii objektov.

Neurónové siete vo všeobecnosti trpia niekoľkými negatívnymi vlastnosťami, ako sú výpočetná a časová náročnosť pri trénovaní, ako aj komplexnosť pri budovaní konkrétnych architektúr. Tieto obmedzenia viedli k vzniku automatizovaného strojového učenia, ktorého integrálnou súčasťou je tzv. Neural Architecture Search. Ide o oblasť, ktorej cieľom je automatizovať proces tvorby neurónových sietí prostredníctvom techník, ako sú evolučné algoritmy a posilované učenie. Tieto techniky sú podporené takzvanými prediktormi presností neurónových sietí, ktoré majú za cieľ zefektívniť proces hľadania najsľubnejších architektúr neurónových sietí v procese vyhľadávania neurónových sietí. Táto časť bola nasledovaná popisom existujúcich databáz predtrénovaných konvolučných neurónových sietí, ktoré obsahujú jednotlivé architektúry spolu so zodpovedajúcimi metrikami, ktoré ich charakterizujú.

Navrhovaná metóda je založená na výbere relevantnej úlohy pre konvolučné neurónové siete (klasifikácia obrázkov), zostavení príslušného datasetu (podmnožina datasetu NAS-Bench-101), definovaní relevantných vstupných príznakov pre prediktory a vo výbere piatich metód strojového učenia pre trénovanie prediktorov.

Pomocou existujúcich knižníc boli jednotlivé prediktory implementované, natrénované a experimentálne bola overená ich funkčnosť a výkonnosť. Výsledky experimentov sú podrobne analyzované, validované a navzájom porovnané. Poskytujú pohľad na efektívnosť navrhovanej metódy a naznačujú potenciál pre ďalšie vylepšenia v oblasti predpovedania presnosti konvolučných neurónových sietí.

Design of Accuracy Predictors for Convolutional Neural Networks

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of prof. Ing. Lukáš Sekanina, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Šimon Šmída
May 17, 2023

Acknowledgements

I would like to express my sincere gratitude to my supervisor Lukáš Sekanina, whose invaluable guidance, support, and professional insight have been instrumental in the successful completion of this work.

Contents

1 Introduction **3**

2 Selected Machine Learning Models **4**
2.1 Perceptron 6
2.2 Multilayer Perceptron 7
2.3 Linear Regression 9
2.4 Random Forest 10
2.5 XGBoost 11
2.6 Graph Convolutional Networks 12

3 Convolutional Neural Networks **14**
3.1 Architecture 17
3.2 Benchmarks 21

4 Neural Architecture Search **23**
4.1 Search Space 24
4.2 Search Strategy 26
4.3 Performance Estimation Strategy 26
4.4 Benchmarks 29

5 Design of Accuracy Predictors **31**
5.1 Scikit-Learn 32
5.2 PyTorch 33
5.3 Performance Predictor Specification 34
5.4 Performance Predictor Structure 39

6 Implementation **42**
6.1 Program Structure 42
6.2 Dataset representation 42
6.3 Predictor Implementations 44
6.4 Accuracy predictors – analysis 46

7 Experimental results **47**
7.1 Experiment setup 48
7.2 Experiment Set 1: Conventional Approach 49
7.3 Experiment Set 2: Extended Feature Set 54
7.4 Experiment Set 3: Graph Convolutional Networks 55
7.5 Efficiency Gain Analysis of Accuracy Predictors in NAS 57

8 Conclusion **58**

Bibliography **60**

A Experiment Set 2 visualizations **65**

Chapter 1

Introduction

Deep learning has made significant advancements in various domains, such as natural language processing, computer vision, and speech recognition. Among deep learning techniques, Convolutional Neural Networks (CNNs) have revolutionized the field of computer vision by achieving state-of-the-art performance in a variety of tasks including image classification, object detection, and semantic segmentation.

The manual design of optimal artificial neural network architectures can be a time-consuming process that heavily relies on expert knowledge and often involves difficulty in identifying optimal hyperparameters. To address these challenges, Neural Architecture Search (NAS) has emerged as a promising approach to searching for the best architectures automatically. This eliminates the need for manual design and extensive expert knowledge.

In this context, developing accurate and resource-efficient predictors for the performance of CNNs becomes essential. These predictors can significantly reduce the computational demands of NAS by rapidly estimating the performance of various architectures, eliminating the need for time-consuming training and evaluation. Consequently, the search process is accelerated, and a broader search space can be explored, potentially discovering networks with superior performance. Furthermore, understanding the factors that impact the effectiveness of particular CNN architectures can provide valuable insights for designing future CNNs, promoting the development of more powerful and efficient models.

The main goal of this thesis is to develop, execute, and assess a technique for building classification accuracy predictors for CNNs. To achieve this, the study involves choosing a suitable task and dataset to evaluate the performance of the CNNs and the predictors, generating a dataset using current NAS benchmarks, identifying appropriate features used as input to the predictor, and selecting a minimum of three machine learning methods for training the predictor. Through the comparison of these methods, this thesis seeks to determine the most efficient approach for estimating the classification accuracy of CNNs, ultimately aiming to enable more streamlined and precise NAS.

The theoretical chapters of this bachelor thesis provide an overview of CNNs, NAS, relevant benchmarks, and machine learning predictors for accuracy estimation (Chapters 2-4). It is followed by the description of the design of the classification accuracy predictor, including task selection, choice of the dataset, feature extraction, and machine learning method selection (Chapter 5). The details of the implementation of the accuracy predictors using existing libraries and programming tools are provided in Chapter 6. Then the experimental setup, results, and statistical validation of the constructed predictors are presented, offering a comprehensive comparison of their performance (Chapter 7). The conclusion of the thesis summarizes the main findings and discusses their main implications (Chapter 8).

Chapter 2

Selected Machine Learning Models

Artificial intelligence (AI) has revolutionized the way we interact with the world, transforming various industries and enabling us to solve complex problems that were considered impossible. From self-driving cars [3], advanced robotics [1] and natural language processing [42], AI has become the cornerstone of modern technological advancements, promising to reshape industries, empower individuals, and create a more sustainable and equitable world. The power of AI not only lies in its ability to optimize processes, but also in its potential to unlock new avenues for creativity, collaboration, and progress.

Machine learning (ML) is a crucial component of AI that focuses on the development of algorithms that can learn from data and make predictions or decisions without being explicitly programmed [39]. By automating this process, ML enables systems to adapt to new and changing environments, making them more robust and reliable. ML techniques can handle the complexity and volume of massive datasets, which represents a challenge for humans in terms of processing and analysis [38], allowing for more efficient and accurate decision-making based on data-driven insights.

In this study, supervised learning methods are employed due to their relevance for tasks such as accuracy prediction of CNNs within the framework of NAS. Supervised learning involves learning from labeled data, where the model learns to map input features to output labels. Various ML models and algorithms are utilized, each with its own strengths and weaknesses.

Linear regression (Section 2.3) is a simple model that attempts to find the best-fitting linear relationship between input features and output labels. *Random forests* (Section 2.4), an extension of decision trees, build multiple trees and aggregate their results to improve prediction accuracy and reduce overfitting. Additionally, *XGBoost* (Section 2.5), an optimized distributed gradient boosting method, performs exceptionally well on structured or tabular datasets. For more complex tasks, *multilayer perceptrons* (MLPs) (Section 2.2) are employed. MLPs are feedforward neural networks with one or more hidden layers that can learn complex nonlinear relationships between input and output. Furthermore, *Graph Convolutional Networks* (GCNs) (Section 2.6), are a type of neural network designed to work directly on graphs and take advantage of their structural information.

In the context of neural networks, gaining a deeper understanding of their inner workings requires examining their fundamental building block: the *artificial neuron*. This component is essential for the network's ability to model and learn complex patterns and relationships in data.

Artificial Neuron An artificial neuron (also *node* or *unit*) is a fundamental building block of neural networks, inspired by the biological neurons present in the human brain [19]. These computational models attempt to simulate the behavior of biological neurons by receiving input signals, processing them, and generating an output signal. The input signals are weighted according to their importance, and the neuron computes the weighted sum of its inputs. This sum is then passed through an *activation function* that determines the neuron's output (Figure 2.1). The purpose of the activation function is to introduce *non-linearity* into the model, allowing it to learn complex patterns and relationships in the data.

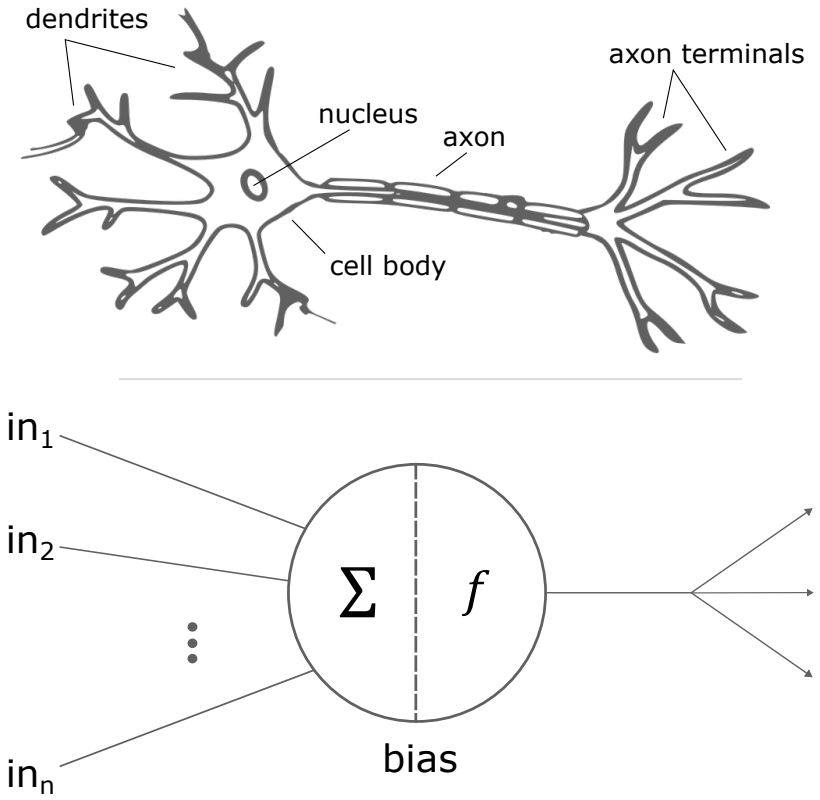


Figure 2.1: A comparison of a biological neuron (top) and an artificial neuron (bottom)[1]. The biological neuron consists of a cell body (soma), dendrites, and an axon. The artificial neuron consists of input features, weights, a bias, a summation function Σ, and an activation function f.

As neural networks evolved, researchers sought to create more advanced structures and algorithms to enhance their performance and enable them to model complex relationships in data. This pursuit led to the development of various types of artificial neuron models, each with their distinct characteristics and advantages. One of the earliest and most well-known examples of an artificial neuron model is the *perceptron* (Section 2.1), which was a groundbreaking innovation in its time. The perceptron introduced a simple yet effective method for binary classification, paving the way for more sophisticated neural network architectures that would emerge in the years to come.

[1] Adapted from: https://www.v7labs.com/blog/neural-networks-activation-functions

2.1 Perceptron

The *perceptron*, introduced by Frank Rosenblatt in the late 1950s [46], is a simple form of an artificial neuron. It is a binary linear classifier that uses a step function as its activation function (Figure 2.2). The perceptron can only learn linearly separable patterns, as it models a linear decision boundary (Figure 2.3). While the perceptron was a significant milestone in the development of artificial neural networks, its limited capacity to model complex relationships led to the development of more advanced neural networks with multiple layers (Section 2.2) and non-linear activation functions.

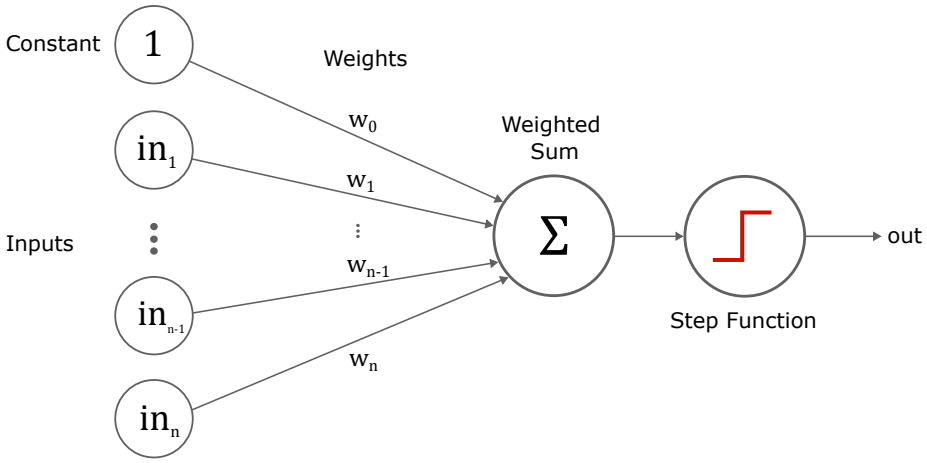


Figure 2.2: A typical perceptron consisting of n input neurons, each associated with a corresponding weight that represents the strength of the connection between two neurons. A bias term (represented as a constant 1) is added to the input layer to provide additional flexibility. The perceptron employs a step function as its activation function, determining the output based on the weighted sum of the inputs and the bias term[2].

Perceptron learning algorithm The perceptron learning algorithm, also known as the *delta rule*, is a specific algorithm used to update the weights of the perceptron during training [39]:

$$w_i \leftarrow w_i + \alpha(y - \hat{y})x_i \tag{2.1}$$

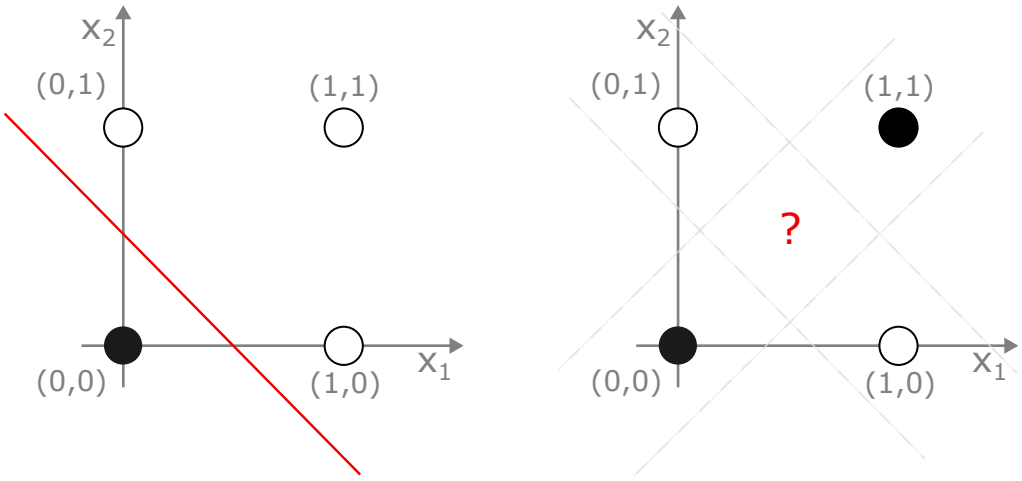
where w_i represents the weight associated with the i-th input feature, α is the learning rate, y is the true label, $\hat{y}$ is the predicted label, and x_i is the i-th input feature.

The delta rule updates each weight by adding the product of the learning rate, the error between the true and predicted labels, and the corresponding input feature. This process is iteratively applied to all the input features in order to minimize the error between the predicted and true labels.

The XOR Problem A major limitation of the perceptron is its inability to learn certain problems that are not linearly separable, such as the XOR (exclusive OR) problem. The XOR problem refers to a simple classification task where two input features, both binary,

[2]Inspired by: https://www.v7labs.com/blog/neural-network-architectures-guide

must be classified into one of two classes. The output is true (1) when the input features have different values (0 and 1 or 1 and 0), and false (0) when the input features have the same values (0 and 0 or 1 and 1). The XOR problem is not linearly separable, meaning that there is no single linear decision boundary that can accurately separate the two classes, as demonstrated in Figure 2.3.



(a) **OR** operation: The four points can be linearly separated using a single line (red).

(b) **XOR** operation: The four points cannot be linearly separated using a single line.

Figure 2.3: Comparison of OR and XOR operations in terms of linear separability.

The XOR problem and the perceptron's inability to solve it highlighted the need for more powerful learning models that could handle non-linearly separable problems. This realization led to the development of *multilayer perceptrons* (Section 2.2) with multiple layers of artificial neurons, allowing for the modeling of more complex, non-linear relationships between input and output [49].

2.2 Multilayer Perceptron

A Multilayer Perceptron (MLP) is an extension of the perceptron model and a type of *feedforward* neural network that consists of multiple layers of neurons, each connected to the neurons in the previous layer, as illustrated in Figure 2.4. Unlike perceptrons, MLPs can solve non-linearly separable problems, significantly expanding their applicability to a wider range of tasks [44]. The input data is fed into the network through the *input layer*, and then it is processed through one or more *hidden layers* before producing an output in the *final layer*. Each neuron in the hidden and output layers is associated with *bias* (additional parameter) and an *activation function* (introducing nonlinearity). The *bias* term serves to shift the activation function along the input axis, allowing the model to better fit the data by providing an additional degree of freedom.

The basic idea behind MLP is to use a combination of linear and nonlinear transformations to transform the input data into a form that can be used for prediction. Each neuron in the network applies a linear transformation to the input data and then applies a nonlinear activation function to the result, as shown in the following equation:

$$\text{output} = f\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right) \tag{2.2}$$

where f is an activation function, w_i and x_i represent the weights and inputs associated with the neuron, b is the bias term, and n is the number of inputs [46].

2.2.1 Structure

The structure of an MLP (depicted in Figure 2.4) begins with the *input layer*, consisting of neurons corresponding to the input features. Each input neuron is then connected to the neurons in the subsequent *hidden layer* through a set of *weights* (learnable parameters). The hidden layers are responsible for learning and representing the underlying structure of the data. The number of hidden layers and the number of neurons within each layer are adjustable parameters, significantly contributing to the model's performance. The final layer, the *output layer*, produces the predicted output values – either continuous (for regression tasks) or categorical (for classification tasks).

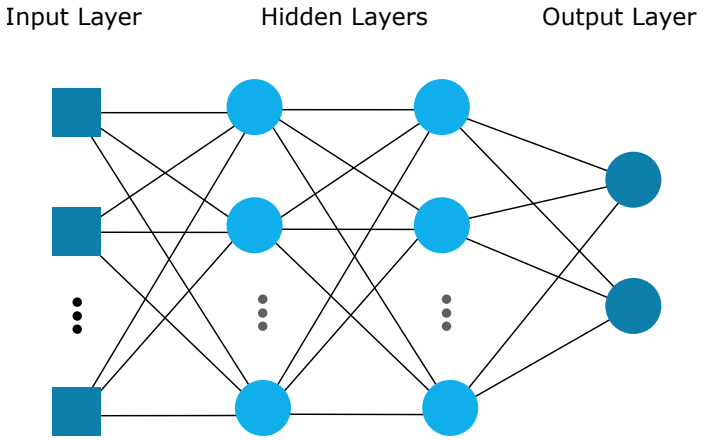


Figure 2.4: Illustration of Multilayer Perceptron architecture, with an input layer, 2 hidden layers, and an output layer. Connections between individual neurons represent the weights[3].

Training Training an MLP involves adjusting the weights and biases of the neurons to minimize the error (loss) between the predicted outputs and the actual output for a given input. This is usually achieved using an optimization algorithm such as *gradient descent* [15], in combination with a technique called *backpropagation* [49] to efficiently compute the gradients of the loss function with respect to the weights and biases of the neurons in the network.

Properties MLPs can learn complex, nonlinear relationships and offer flexibility through customizable activation functions, layers, and architectures. However, they can be computationally expensive and prone to *overfitting*, learning the training data too well (including the noise), if not regularized. Techniques like L1, L2 regularization, or dropout help prevent overfitting, but MLPs may still lack interpretability compared to simpler models [15].

[3]Image adapted from: https://www.javatpoint.com/multi-layer-perceptron-in-tensorflow

2.3 Linear Regression

Linear regression is a commonly used algorithm in supervised learning, where the goal is to learn a function that maps input variables to a continuous output variable [18]. It is a simple, yet powerful, statistical method that models the relationship between the input variables (predictors) and the output variable (response). The primary goal of linear regression is to find the best-fit line (Figure 2.5) that describes the relationship between the input variables and the output variable. This line is represented by a linear equation 2.3 that can be used to predict the value of the output variable given a set of input variables.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \epsilon \tag{2.3}$$

where y is the target variable, $x_1, x_2, \ldots, x_n$ are the input features, β_0 is the intercept, $\beta_1, \beta_2, \ldots, \beta_n$ are the coefficients (weights) associated with each feature, and ϵ is the error term – the difference between the actual value of the target variable and the predicted value obtained from the model [41].

To find the best-fit line, the task is to minimize the sum of squared residuals between the predicted values and the actual values. This is known as the *least squares criterion* and can be calculated as follows:

$$L(\boldsymbol{\beta}) = \sum_{i=1}^{N} (y_i - (\beta_0 + \beta_1 x_{1_i} + \beta_2 x_{2_i} + \cdots + \beta_n x_{n_i}))^2 \tag{2.4}$$

where $L(\boldsymbol{\beta})$ is the *objective function* we want to minimize, y_i is the actual value of the target variable for the i-th observation, β_0 is the intercept, $\beta_1, \beta_2, \ldots, \beta_n$ are the coefficients (weights) associated with each feature, and $x_{1_i}, x_{2_i}, \ldots x_{n_i}$ are the values of the input features for the i-th observation [41].

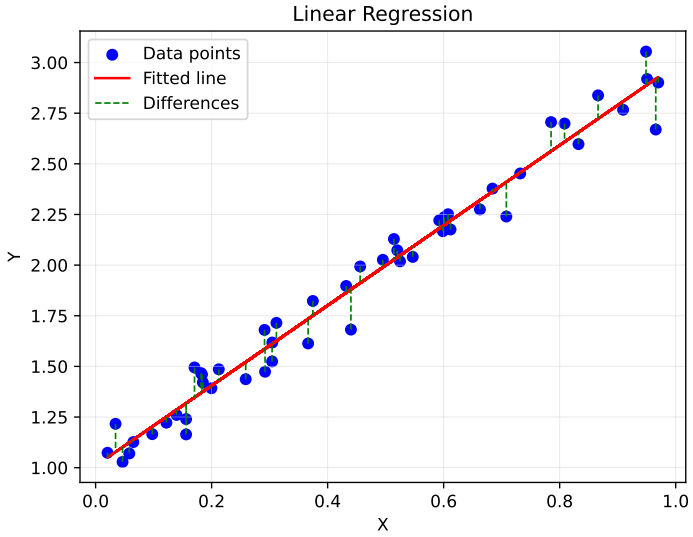


Figure 2.5: Depiction of Linear Regression Analysis: The blue points represent the data points, the red line is the fitted linear regression model, and the green dashed lines show the differences between the predictions and the actual values.

Properties The model is easy to understand and interpret, as the coefficients directly represent the contribution of each input feature to the target variable. Linear regression is also computationally efficient and has a quick training time, which makes it suitable for situations where resources are limited or where the relationship between input features and output labels can be reasonably approximated by a linear function [18].

Assumptions *Linearity* assumes that the relationship between the input features and the target variable is linear, while the *independence* of errors assumes that the errors are not correlated with each other. The *constant variance of errors* assumes that the errors have the same variance across all levels of the input features, and the *normal distribution of errors* assumes that the errors follow a normal distribution. Violations of these assumptions can lead to biased or inefficient estimates of the model parameters [18]. In cases where these assumptions do not hold, other machine learning methods, such as non-linear regression or tree-based models (Sections 2.4 and 2.5), may be more appropriate.

2.4 Random Forest

Random forest is a popular ML algorithm that belongs to the family of *ensemble methods*. Ensemble methods combine multiple models to improve predictive performance and reduce overfitting. Random forest achieves this by constructing multiple *decision trees* during the training phase and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees [18].

Decision trees are tree-like structures with internal nodes representing feature tests, branches representing the outcomes of these tests, and leaf nodes representing the final predictions. Random forest works by building a collection of decision trees, each trained on a random subset of the data and features (Figure 2.6).

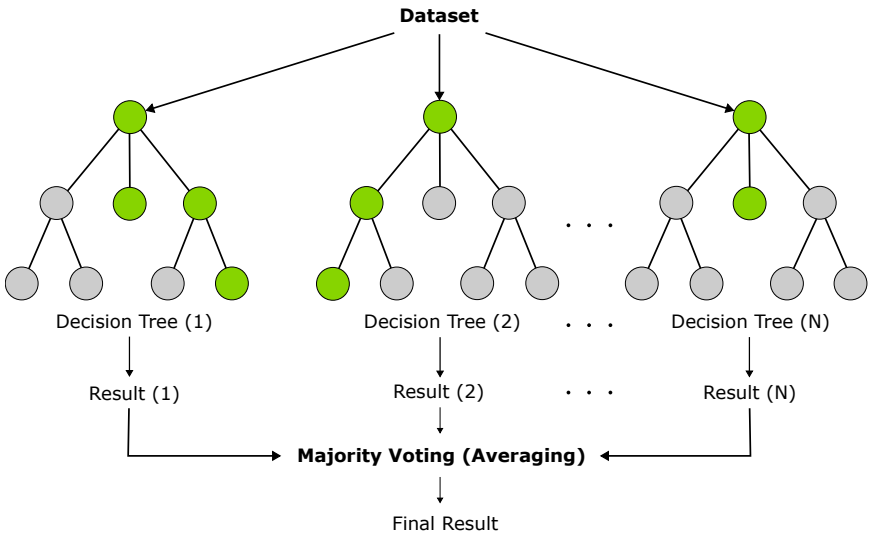


Figure 2.6: Depicting the Random Forest algorithm: a dataset feeds multiple decision trees, and their results are aggregated by majority voting, showcasing ensemble learning. Adapted from [28].

This process of sampling with replacement is known as *bootstrap aggregation* or *bagging* and introduces diversity among the trees. Additionally, a random subset of features is used at

each split in the decision trees, further increasing the diversity of the trees in the ensemble. The decision trees are then combined to form a random forest that can make predictions on new data. This approach reduces overfitting and increases generalization performance by reducing the variance of the models [4].

Properties Random forests have several advantages over other ML algorithms. Firstly, they can handle both classification and regression tasks, making it a versatile algorithm. They can handle missing data without imputation and are less sensitive to outliers and noise in the data compared to single decision trees or linear models. By averaging the results of multiple trees, random forests are less prone to overfitting than individual decision trees. It can also provide measures of feature importance, allowing the identification of the most important features in the data. Random forests, however, can also be computationally expensive, especially when working with large datasets or a high number of decision trees. This can cause longer training and inference time.

2.5 XGBoost

XGBoost, short for eXtreme Gradient Boosting, is a powerful ML algorithm that belongs to the family of gradient boosting methods. Gradient boosting algorithms are a type of ensemble method that combines multiple weak learners (typically decision trees) in a *sequential* manner to improve predictive performance and reduce overfitting [7].

Gradient boosting works by iteratively adding weak learners to the ensemble, with each new learner correcting the errors (residuals) made by the previous ones. The weak learners are decision trees that are added sequentially, and each tree learns to correct the *residuals* of the previous trees (Figure 2.7). The learning process is guided by the gradient of the loss function, optimized at each step to minimize the errors made by the ensemble. This results in an adaptive model that can accurately capture complex patterns in the data [7].

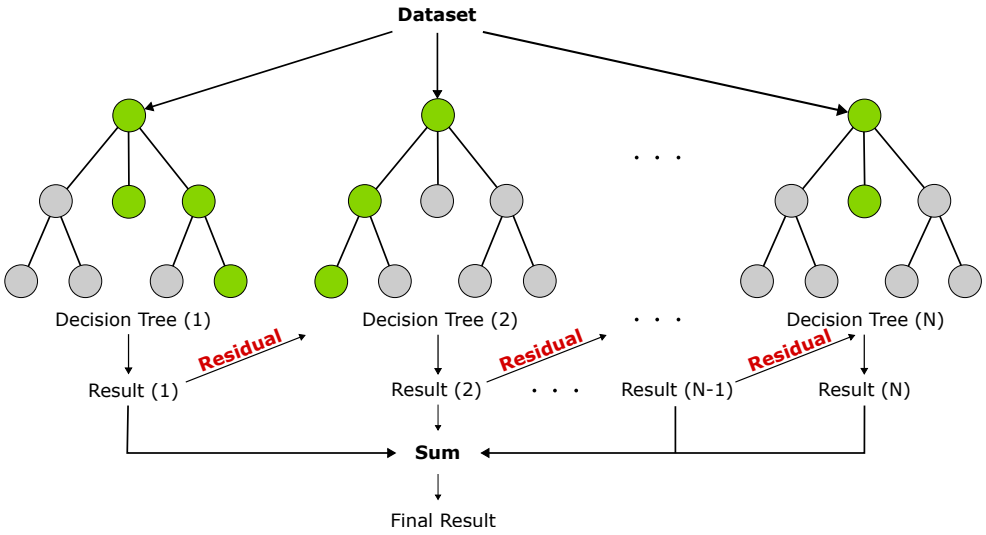


Figure 2.7: Depicting XGBoost's process: initial predictions are improved by decision trees learning from residuals, culminating in an aggregated output – a demonstration of gradient boosting. Adapted from [16].

Properties XGBoost offers several advantages over other gradient boosting methods and ML algorithms. Firstly, it is highly efficient and scalable, enabling the algorithm to handle large datasets and run on distributed computing systems. Secondly, XGBoost supports regularization, which helps reduce overfitting by penalizing more complex models. The algorithm incorporates both L1 and L2 regularization, allowing users to control the trade-off between model complexity and generalization performance. Additionally, XGBoost can handle missing data without requiring imputation, and it provides built-in support for feature importance analysis, making it easier for users to interpret the model and identify relevant features. XGBoost can also handle both classification and regression tasks, making it a versatile algorithm for various ML problems. XGBoost, however, like other tree-based methods, can be sensitive to noisy data and outliers, which may affect the model's performance.

2.6 Graph Convolutional Networks

Graph Convolutional Networks (GCNs) are a class of deep learning models designed to handle graph-structured data, as illustrated in Figure 2.8. Unlike traditional feedforward neural networks, which expect a fixed-size input, GCNs can process graphs with varying sizes and structures, making them well-suited for tasks involving graph-based data. In the context of NAS, GCNs can capture complex relationships between the nodes and edges in the search space, potentially leading to more accurate performance predictions.

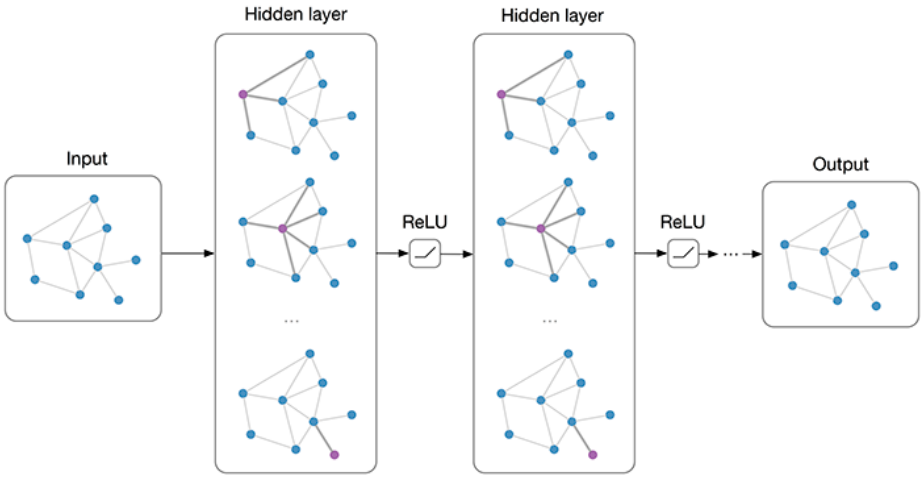


Figure 2.8: Illustration of Graph Convolutional Network architecture[4]. Nodes are connected to their neighbors through edges, and each node has a feature vector. The GCN learns to propagate information from neighbors to the central node, updating the feature vector of the central node.

2.6.1 Structure

A GCN consists of several graph convolutional layers, which operate on the nodes and their neighboring nodes in the graph. Each node in the graph is associated with a feature vector, and these feature vectors are updated throughout the graph convolutional layers. The

[4]Image from: https://tkipf.github.io/graph-convolutional-networks/

graph convolutional layers can capture local and global information in the graph, enabling the model to learn complex patterns and relationships between the nodes and edges.

Graph Convolutional Layer The fundamental process within a graph convolutional layer is the *neighborhood aggregation*, which involves gathering and combining information from the adjacent nodes. For a given node, the model learns a weighted combination of the neighboring nodes' feature vectors, updating the feature vector of the central node. This operation can be seen as a *convolution* on the graph, where the weights are learned by the model during training.

Training Training a GCN involves adjusting the weights and biases of the graph convolutional layers to minimize the error (*loss function*) between the predicted outputs and the actual outputs for a given graph input. This is usually achieved using an optimization algorithm such as *gradient descent* [48] or *Adam* [29], in combination with techniques such as *backpropagation* [49] or *graph attention mechanisms* [61] to efficiently compute the gradients of the loss function with respect to the weights and biases of the graph convolutional layers.

Properties GCNs are able to learn complex relationships between nodes and edges in graph-structured data, making them particularly suitable for NAS performance prediction. They are flexible and customizable through the selection of different layer types, architectures, and attention mechanisms. GCNs can be computationally expensive, especially when processing large graphs or when using deep architectures. They may also be prone to overfitting if not properly regularized. Regularization techniques, such as L1 and L2 regularization or graph pooling, can be used to prevent overfitting by adding constraints to the model parameters or by reducing the model's complexity during training [15]. GCNs, like other deep learning models, often lack the interpretability of simpler models like linear regression and random forests.

Chapter 3

Convolutional Neural Networks

In this chapter, the key components and principles of CNNs are explored. The chapter begins by presenting an overview of the basic architecture of CNNs, including their layers and operations, in Section 3.1. Section 3.2 discusses benchmarks for CNNs.

CNNs are a type of deep neural networks that have become increasingly popular in the field of AI, particularly in image and video processing tasks. They have shown remarkable performance in a range of applications, from image recognition and segmentation to object detection and tracking.

Convolutional networks have had a pivotal role in the history of deep learning. The unique design of CNNs allowed them to be more efficient than *fully connected networks* which were prone to overfitting and computationally more complex [33]. *Fully connected networks* were also less effective at learning spatial features and patterns in data, which is critical for tasks such as image and speech recognition [15]. They are typically trained using backpropagation, stochastic gradient descent, and other optimization techniques to minimize a loss function that measures the difference between the predicted outputs and the actual outputs. Regularization techniques, such as dropout, weight decay, and early stopping, are often used to avoid overfitting the model to the training data [15, 55].

As already mentioned, CNNs have been successfully used in a variety of applications, such as image recognition, object detection, and segmentation. They have also been used in real-world applications, such as self-driving cars, medical diagnosis, and video analysis. Recent advancements in the field of CNNs include the use of transfer learning, attention mechanisms, and adversarial training. However, there are also challenges and limitations of CNNs, such as their interpretability, scalability, and bias. The *interpretability* of CNNs has been a long-standing issue in the deep learning community, as it is often difficult to understand the inner workings of these models and how they arrive at their prediction [40]. This lack of transparency can pose problems in applications where model interpretability is crucial, such as medical diagnosis or financial decision-making. *Scalability* is another challenge for CNNs, especially when working with high-resolution images or large-scale datasets [56]. Training deep convolutional networks requires significant computational resources and can be time-consuming, which can be a bottleneck in certain applications. CNNs, like other machine learning models, can be susceptible to *biases* present in the training data [39]. This can lead to biased predictions or even perpetuate existing biases when these models are deployed in real-world settings. The deep learning community is actively working on developing methods to mitigate these issues and improve the fairness and robustness of CNNs and other deep learning models.

Biological brain as an inspiration for CNNs Convolutional networks were inspired by biological brains. Their history can be traced back to neuroscientific experiments conducted long before the corresponding computational models were developed. The work of neurophysiologists David Hubel and Torsten Wiesel greatly contributed to our understanding of how the visual system of mammals works [15]. The most influential findings of their work, which have heavily impacted current deep learning models, were obtained by monitoring the activity of individual neurons in cats [23]. Their research involved projecting images onto a screen in front of the cat and recording the response of the neurons in its brain. The researchers discovered that neurons in the primary visual cortex of mammals were sensitive to local regions in the visual field and responded to specific orientations of edges and bars. This research inspired the development of computational models that mimicked the local receptive fields and hierarchical structure of the visual cortex.

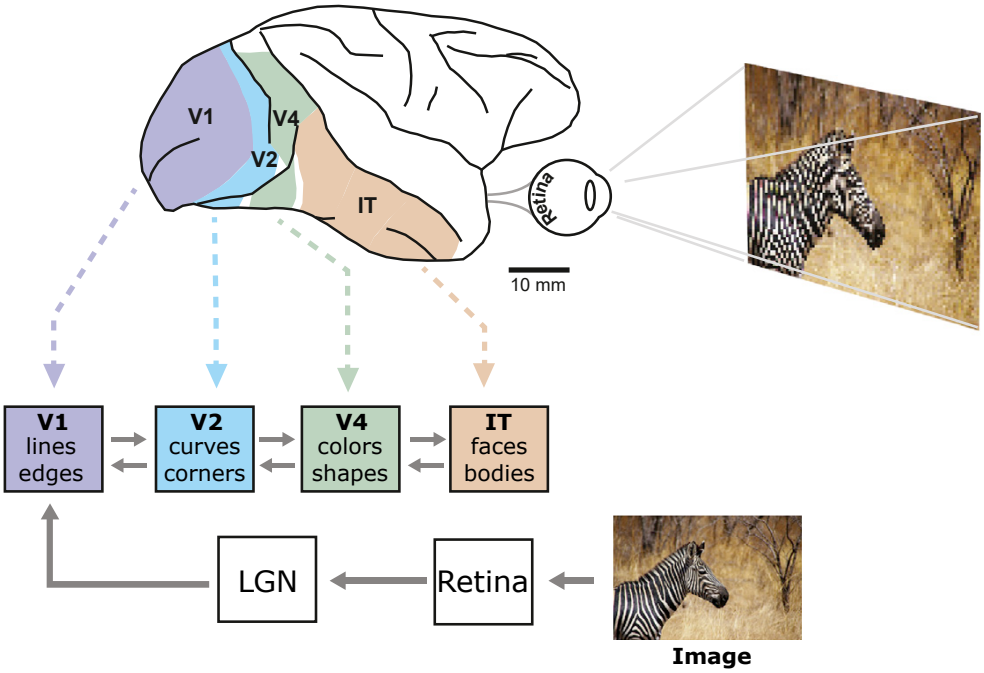


Figure 3.1: The ventral visual stream (V1, V2, V4, and IT) receives inputs from the lateral geniculate nucleus (LGN) of the thalamus, which receives input from the retina. The LGN projects to V1, the primary visual cortex, which is the first cortical area involved in processing visual information. From V1, the processed visual information is then transmitted to the other areas in the ventral visual stream, such as V2, V4, and IT, which are responsible for progressively more complex visual processing. The connections between these areas are both *feedforward*, where information flows from lower to higher processing areas, and *feedback*, where information flows from higher to lower processing areas. Adapted from [8].

Convolutional networks are designed to extract features from visual data by applying convolutional filters to the input image. These filters are designed to detect edges and patterns at different scales and orientations, much like the receptive fields in the visual cortex (Figure 3.1). The hierarchical structure of CNNs (multiple layers of filters) is also inspired by the hierarchical processing of visual information in the brain. The filters in the earlier layers

detect simple features like edges and corners, while the filters in the deeper layers detect more complex patterns and objects, as shown in Figure 3.2.

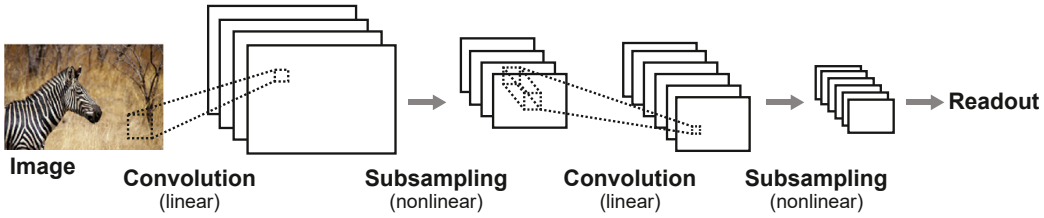


Figure 3.2: A simple feedforward convolutional network can be compared to the hierarchy of the biological visual system, where the two pairs of convolution operator and pooling layer have a similar function. Adapted from [8].

Neocognitron One of the first computational models that drew inspiration from the hierarchical organization of the visual cortex was proposed by K. Fukushima with the *Neocognitron* [14]. It consisted of alternating layers of simple cells (*S-cells*) and complex cells (*C-cells*). These cells were designed to extract local features, such as edges, corners, and texture. They provide invariance to scale (*S-cells)* and rotation (*C-cells*). Even though the Neocognitron was not a fully-fledged CNN, it laid the groundwork for future research in the field.

LeNet-5 In 1998, Yann LeCun and his team developed *LeNet-5* network [34] – pioneering CNN architecture designed for handwritten digit recognition and machine-printed character recognition. It showed its real-world usage when it was in applied the United States Postal Service for processing and recognizing handwritten zip codes on envelopes, improving the efficiency of mail sorting and routing, reducing manual work, and speeding up the delivery process. LeNet-5 employed a combination of convolutional layers, pooling layers, and fully connected layers to extract features and make predictions. It demonstrated impressive performance on the now well-known MNIST dataset [34].

Over the years, CNNs have continuously evolved and improved. Some of the most influential architectures that have emerged include *AlexNet* in 2012 [33], which significantly boosted CNN performance in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). *VGGNet* [53], developed in 2014, featured a deeper and more uniform structure, leading to even better performance in the ILSVRC. *ResNet* [21], introduced in 2015, employed residual connections that enabled the training of extremely deep networks. *Inception* (also known as GoogLeNet) [60], developed in 2014, utilized inception modules to allow the network to learn multi-scale features while reducing the number of parameters. This innovation resulted in improved efficiency and performance across various computer vision tasks.

3.1 Architecture

The typical structure of a CNN consists of multiple layers, including convolutional layers (extracting features using filters), pooling layers (reducing the spatial size), and fully-connected layers (flattening, classification/regression). The order and number of layers may vary depending on the specific architecture, designed for a specific task [58].

An architecture of a CNN, depicted in Figure 3.3, can be visualized as a flowchart, where the input image is fed into the network, and each layer transforms the input into a higher-level representation of the image. The final layer can output a vector of scores that represent the likelihood of the input image belonging to each of the possible classes.

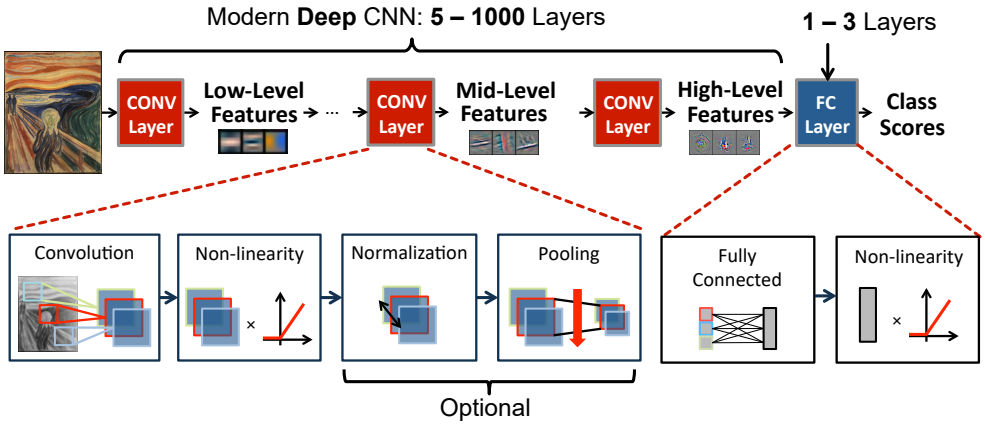


Figure 3.3: A general architecture of a CNN, adapted from [58].

3.1.1 Convolutional layer

A *convolutional layer* is one of the most fundamental building blocks of a CNN. The layer consists of a set of learnable *filters* (also called *kernels*) that are convolved with the input data to produce a set of feature maps. Each filter, a small matrix of numbers (weights), moves across the input image in a sliding-window fashion (Figure 3.5), performing a dot product operation with the local region of the input image that it covers.

The neurons in the initial convolutional layer have selective connections to particular clusters of pixels within the input image, referred to as their *receptive fields*. As a result, they are not influenced by every pixel in the image, but rather by specific receptive fields (refer to Figure 3.4). Subsequently, each neuron in the second convolutional layer establishes connections exclusively with neurons situated within a compact rectangular region of the first layer [17]. *Receptive field* refers to the region of the input image that affects the activation of a particular neuron in the convolutional layer. Its size is determined by the size of the convolutional filter (kernel) and the stride of the convolution operation. The *stride* is a parameter that controls how the filter moves across the input image during the convolution process. It defines the number of pixels the filter moves horizontally and vertically after each convolution operation. A larger stride results in a smaller output feature map, while a smaller stride produces a larger output feature map. As we move deeper into the network, the receptive fields of the neurons become larger, and more abstract features can

be learned. Consequently, CNNs work well for image recognition due to their ability to handle the hierarchical structure commonly found in real-world images.

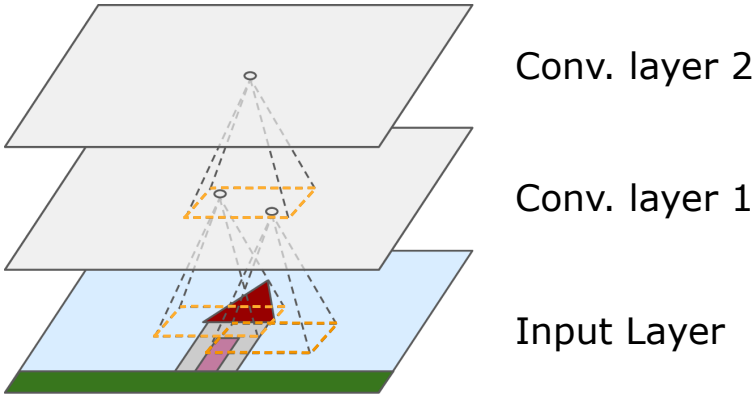


Figure 3.4: Illustration of the first convolutional layers in a CNN, showcasing local receptive fields (highlighted in orange) that are responsible for capturing specific features within the input image. Adapted from [17].

Convolutional layers are designed to be *translation-invariant*, meaning that they can recognize features regardless of their position in the input image. This makes them well-suited for image recognition tasks, where objects can appear at different positions in an image.

Convolution and cross-correlation As the name of this layer suggests, a convolutional layer is closely related to a mathematical operation called *convolution* (Equation 3.1). It involves sliding one function over another and calculates the sum of the product of two functions after one of the functions has been flipped and shifted. The result of the convolution operation is a new function that represents how the shape of one function affects the shape of another. The convolution operation is commonly used in signal processing, image processing, and many other areas of science and engineering.

$$(X * K)_{i,j} = \sum_m \sum_n X_{i+m,j+n} K_{m,n}, \tag{3.1}$$

where $X_{i+m,j+n}$ is the element in the $(i+m)$-th row and $(j+n)$-th column of the input matrix X, and $K_{m,n}$ is the element in the m-th row and n-th column of the *flipped* kernel K.

In practice, convolutional layers typically use a similar operation to convolution called *cross-correlation* (Equation 3.2). Both operations involve sliding a filter (kernel) over an input signal, computing the dot product between the filter and the overlapping part of the input, and producing an output signal. However, in convolution, the filter is flipped horizontally and vertically before being slid over the input signal, while in cross-correlation, the filter is not flipped [37].

$$(X \star K)_{i,j} = \sum_m \sum_n X_{i+m,j+n} K_{-m,-n}, \tag{3.2}$$

where $X_{i+m,j+n}$ is the element in the $(i+m)$-th row and $(j+n)$-th column of the input matrix X, and $K_{-m,-n}$ is the element in the $(-m)$-th row and $(-n)$-th column of the *unflipped* kernel K.

Although the terms *convolution* and *cross-correlation* have slightly different mathematical definitions, they are often used interchangeably in the context of CNNs, because the kernel is adjusted during training to optimize the desired output, regardless of whether it is implemented as a convolution or cross-correlation operation.

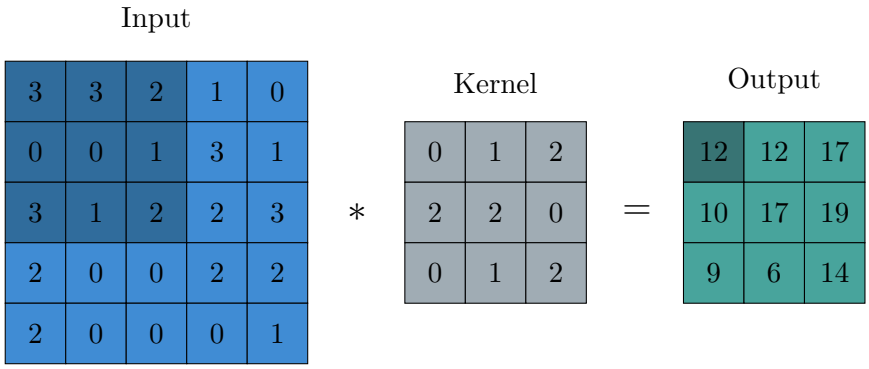


Figure 3.5: Example of the convolution operation with a `5x5` input matrix, a `3x3` kernel matrix, and a `3x3` output matrix. The convolution is performed with a stride of 1. The input and kernel matrices are element-wise multiplied and then summed to compute the corresponding entries in the output matrix. Inspired by [12].

3.1.2 Activation layer

Activation layers are typically used after each convolutional layer. The purpose of these layers is to introduce *non-linearity* into the network. Without non-linearity, the network would simply be a series of linear transformations, which is not sufficient for learning complex patterns in data. The activation function is applied element-wise to the output of the convolutional layer, producing a new feature map. A common activation function used in CNNs is ReLU depicted in Figure 3.6, which sets negative values to zero [33].

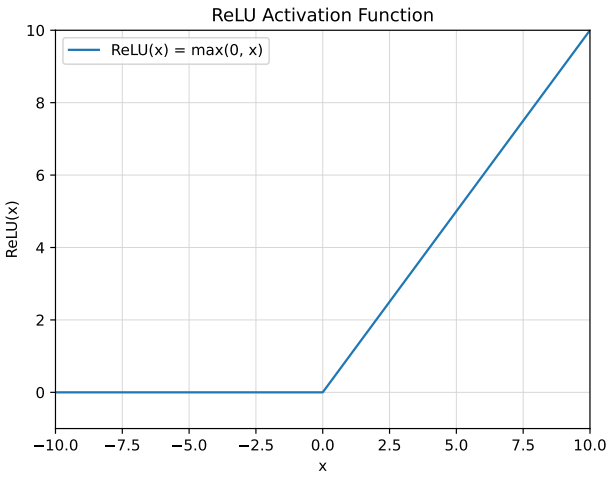


Figure 3.6: Rectified Linear Unit (ReLU) activation function.

The use of activation layers allows the network to learn more complex and discriminative features by introducing non-linearities in the learned representations. This is particularly important for tasks such as image classification, where the input data can be highly complex and varied.

3.1.3 Pooling layer

Pooling layers are commonly used in CNNs to reduce the spatial dimensions of feature maps generated by convolutional layers by using some function (max, average) to summarize subregions (receptive fields). The main purpose of pooling is to decrease the computational cost of the model, while also making the learned features more invariant to small translations in the input image [17]. The process of pooling involves moving a window across the input data and sending the contents of the window to a pooling function.

There are several types of pooling layers, but the most common ones are *max* and *average* pooling (Figure 3.7) [51]. *Max pooling* takes the maximum value within a window (e.g., 2x2) of the feature map and discards the other values. This allows the model to capture the most salient features of the image, while also reducing the spatial resolution of the feature map. *Average pooling* takes the average value within a window and discards the other values, resulting in a similar reduction in spatial resolution.

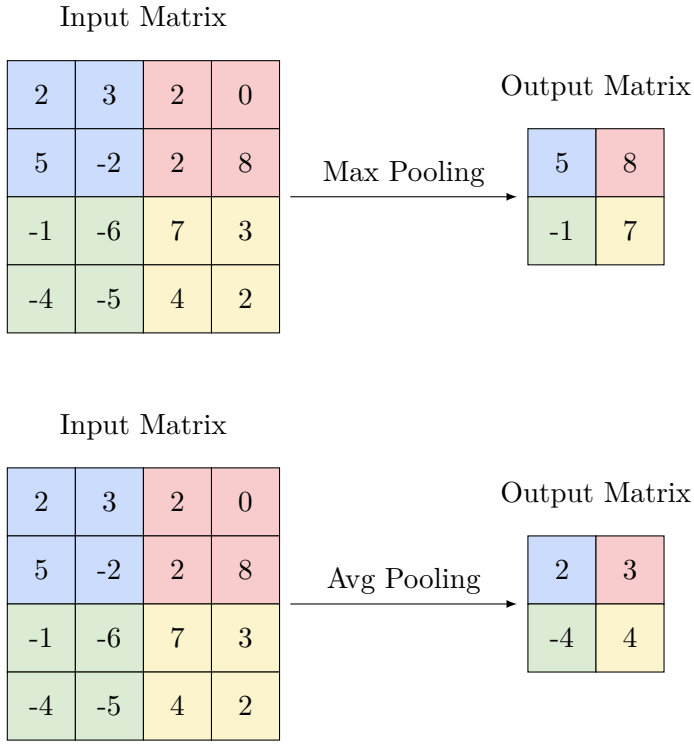


Figure 3.7: Illustration of max pooling (top) and average pooling (bottom) operations on input matrices using a stride of 2 and 2x2 pooling. The max pooling operation selects the *maximum* value from each sub-matrix, while the average pooling operation calculates the *mean* value of each sub-matrix. Both methods employ a stride of 2, which means they move two units at a time horizontally and vertically when scanning the input matrix.

Max pooling is especially effective at preserving the most important features of the image, such as edges or corners. It also helps to reduce the size of the feature maps and to introduce a degree of translation invariance. On the other hand, average pooling is useful in cases where the magnitude of the features is important, such as in medical imaging or satellite imagery.

Pooling layers, however, can also lead to loss of information, which may be critical for certain tasks, such as object localization or segmentation. In addition, pooling may also lead to loss of fine-grained details in the image, which can negatively affect the performance of the model on some tasks. For these reasons, some recent CNN architectures have replaced pooling layers with other techniques, such as strided convolutions [54] or spatial pyramid pooling [20], which aim to preserve more spatial information.

3.1.4 Fully-connected layer

Fully-connected layers in CNNs are the classic type of neural network layer where every neuron in a layer is connected to every neuron in the next layer. In CNNs, fully-connected layers are typically used at the end of the network to process the features learned by the convolutional and pooling layers and to generate the final output [17].

After the convolutional and pooling layers extract high-level features from the input, the fully-connected layers can take these features and use them to make a classification decision. The fully-connected layer takes the flattened output from the previous layer (i.e., the feature vector) and multiplies it by a weight matrix. The output of this multiplication is then passed through a non-linear activation function to introduce non-linearity into the network.

Fully-connected layers are used to learn more complex relationships between the features extracted from the input. They have the ability to represent any function, given enough neurons, and therefore are often used in the final layers of the network for classification and regression tasks [9]. However, fully-connected layers have a high number of parameters, which can lead to overfitting on small datasets. Additionally, they do not take into account the spatial structure of the input, which can lead to the loss of important information. To address these issues, other types of layers, such as convolutional and pooling layers, are typically used in combination with fully-connected layers in CNNs [17].

3.2 Benchmarks

To measure the performance of various CNN architectures and foster advancements in computer vision, a variety of benchmark datasets and competitions have been established. Two of the most prominent and widely used benchmarks are CIFAR [30] and ImageNet [10]. They are crucial for evaluating the accuracy, generalization capabilities, and computational efficiency of different CNN models, providing a standardized reference point for researchers and practitioners alike. They serve as an essential tool for driving the development of novel architectures, optimization techniques, and data augmentation methods, ultimately pushing the boundaries of what is possible in image recognition and classification tasks. By engaging the community through these benchmark challenges, the field of computer vision continues to evolve, resulting in models with improved performance and a broader range of applications.

3.2.1 CIFAR-10/CIFAR-100

CIFAR-10 [31] consists of 60,000 color images, each of size `32x32` pixels, divided into 10 different classes, such as airplanes, automobiles, birds, cats, and more (Figure 3.8). There are 6,000 images per class, with 5,000 images for training and 1,000 images for testing. The relatively small size of the dataset, combined with its diverse set of object classes, makes CIFAR-10 a popular choice for benchmarking and evaluating machine learning algorithms, particularly CNNs.

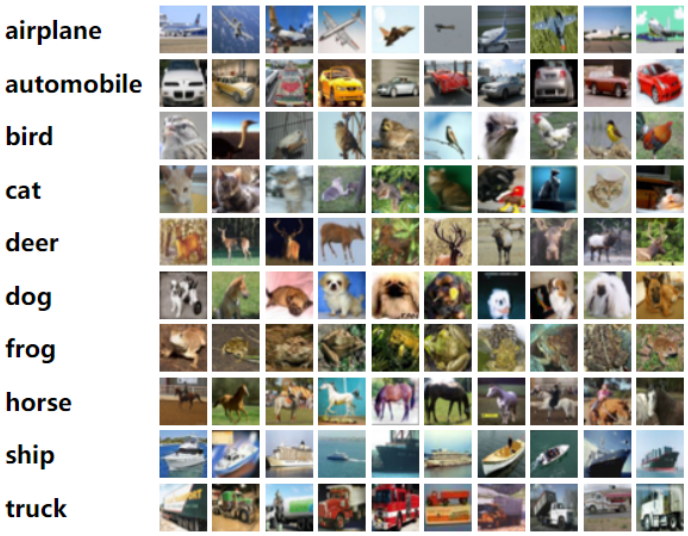


Figure 3.8: CIFAR-10 dataset sample, showing 10 classes and 10 random images per class. From [30].

A more challenging variant of the CIFAR-10 dataset is called CIFAR-100 [32]. The number of images and image resolution are both the same, but there are 600 images per class and 100 different classes overall. These classes are divided into 20 additional, more general categories, such as flowers, vehicles, and aquatic mammals. CIFAR-100 is a more challenging benchmark for assessing the performance and generalization abilities of machine learning models, particularly CNNs, due to the increased number of classes.

3.2.2 ImageNet

ImageNet [10] is a large-scale dataset, originally created for the *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) [50], which has significantly influenced the development of deep learning and computer vision. The dataset contains millions of labeled images spanning thousands of object classes, making it one of the most comprehensive and diverse image databases available. The high-resolution images and the vast number of classes have made ImageNet an essential resource for training and benchmarking deep learning models, particularly CNNs. The annual ILSVRC competition has been instrumental in fostering the development of more advanced CNN architectures and accelerating progress in the field of computer vision.

Chapter 4

Neural Architecture Search

Creating successful and effective machine learning models requires making numerous design choices. Related challenges are particularly pronounced in *deep learning*, a subset of machine learning that involves training ANNs with multiple layers to perform complex tasks. Engineers must select network architectures relevant to a given task, with adequate performance, while also carefully choosing hyperparameters, training procedures, and regularization methods. This process must be repeated for each new application, and even experts in the field may experience laborious trial and error to identify the best choices for a particular dataset [24].

This is where the field of Automated Machine Learning (AutoML) [24] comes to play. It aims to simplify the problematic decisions mentioned in the previous paragraph by leveraging data-driven, objective, and automated approaches. Practitioners need only to provide data, and the AutoML system automatically attempts to determine the optimal approach for a given application. Even non-experts, people lacking the necessary resources to understand the underlying technologies in depth, can use AutoML to quickly and effectively design and implement high-performing machine learning models. This makes it possible for virtually everyone to access customized, state-of-the-art machine learning solutions with ease [24].

Neural Architecture Search is one of the most challenging elements of AutoML as it works with an extremely large design space and the fact that a single evaluation of a neural network is a long, computationally demanding process [13]. NAS attempts to solve the process of automating the design of optimal neural network architecture for specific tasks. The conventional design of neural networks has relied on human expertise and manual trial-and-error, which can be time-consuming, error-prone, and often subjected to biases. Considering these difficulties, automating the design process was a logical decision and by utilizing intelligent search algorithms, including genetic algorithms, reinforcement learning, and Bayesian optimization, NAS can overcome the aforementioned limitations and can automatically identify high-performing architectures without the need for manual intervention. NAS methods achieved numerous successes, outperforming manually designed network architectures on tasks including image classification [69, 45], object detection [69] or semantic segmentation [6].

The use of NAS, however, also comes with a number of difficulties. The high computational cost involved in comparing various architectures is one of the main problems. Numerous neural network architectures must be trained and evaluated as part of NAS, which can be computationally expensive for each architecture. The scalability of NAS to bigger and more complicated datasets may be constrained by this high computational cost.

The sheer size of the search space presents another challenge, making it impossible to investigate every potential architecture. As a result, to lower the computational cost and boost NAS efficiency, researchers have turned to a variety of techniques like weight-sharing [43], surrogate models [25], and neural architecture compression [5]. However, each of these approaches has its own set of drawbacks, and new approaches are constantly looked for to increase the NAS's scalability and effectiveness.

According to research by Elsken et al. [13], the methods of NAS can be categorized as follows: search space, search strategy, and performance estimation strategy. See Figure 4.1 for an abstract illustration of the NAS principles.

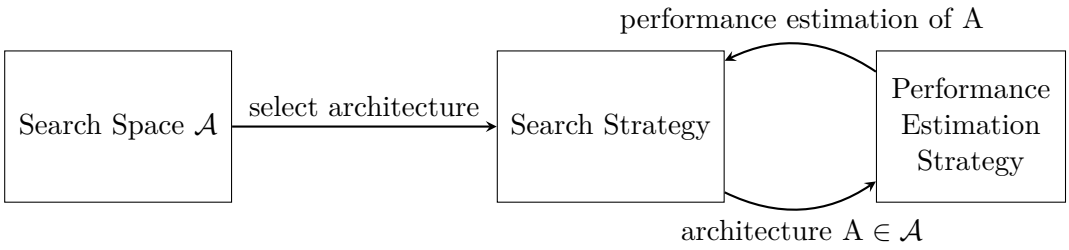


Figure 4.1: NAS involves selecting an architecture A from a predefined search space $\mathcal{A}$ using a search strategy. The chosen architecture is then evaluated for performance using a performance estimation strategy. Adapted from [13].

4.1 Search Space

The search space of NAS represents the set of all possible neural network architectures that can be considered during the search process and that the NAS algorithm can explore. It can be defined in various ways, such as by specifying the types of layers (convolutional layers, pooling layers, recurrent layers), the connectivity between layers (skip connections, dense connections), and the hyperparameters of each layer (number of filters in a convolutional layer, size of a pooling window).

The size and complexity of the search space are critical factors that can significantly affect the efficiency and effectiveness of the NAS process. A small search space may limit the diversity of the architectures that can be explored, while a large search space may make the search process computationally expensive or even infeasible. Thus, designing an appropriate search space is a crucial step in NAS, as it directly impacts the quality and efficiency of the discovered neural network architectures.

Global search space One category of neural architecture search space is the *global search space*. It is defined for the graphs representing a *complete* neural architecture and where the arrangement of operations, like convolution or pooling, is not restricted to a specific pattern or structure. The simplest example of a global search space is the *chain-structured search space*. It consists of architectures that can be represented by an arbitrary sequence of ordered nodes, where each node in the chain has only one parent and represents an operation or layer applied to the input data [26].
The global search space offers more flexibility in architectural design, which can potentially lead to the discovery of novel and highly effective architectures that may not exist in a *cell-*

based search space, another widely used type. However, this flexibility comes at the cost of increased computational complexity, as the search space is typically much larger, and therefore the search process is more computationally demanding. The global search space may also require more manual engineering to adapt a discovered architecture to a different dataset or problem since there are no modular building blocks like cells that can be easily adjusted or transferred.

Cell-based search space In this search space, a neural network architecture is built by replicating a *cell* structure, which is a relatively small neural network module that can be stacked repeatedly to form a larger network (as shown in the example cell on the right side of Figure 4.2). The cell's architecture is defined by a directed acyclic graph (DAG), a graph that has directed edges and no cycles, ensuring the flow of information is unidirectional. In the context of neural networks, a DAG allows for complex layer connections and skip connections while maintaining a valid computational graph [26].

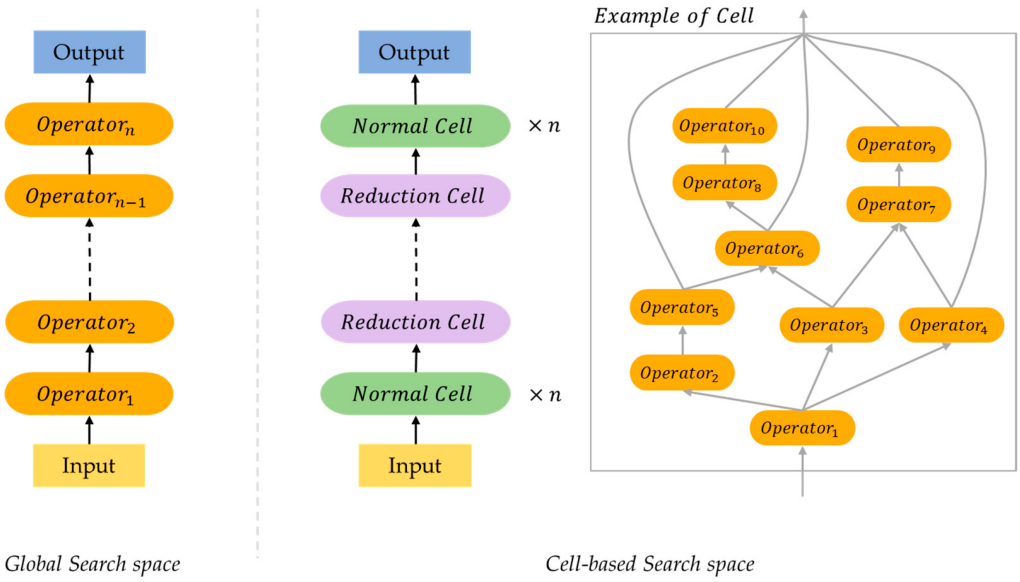


Figure 4.2: Comparison of simplified *global search space* (left) and *cell-based search space* (right) in NAS. From [26].

There are different ways to represent operations and connections in the DAG, depending on the search space formulation. In some NAS benchmarks, such as NAS-Bench-101 [67], nodes represent operations (e.g., convolutional layers, pooling layers), and directed edges represent the flow of information between these operations. In other NAS formulations like NAS-Bench-201 [11], the directed edges themselves represent the operations or layers, and the nodes act as intermediate states or tensors in the network.

The cell-based search space offers several advantages over the other types. Namely, the search space is relatively small, since cells are usually made of significantly fewer layers when compared to the whole network search architectures. Another benefit is that using architectures constructed from cells allows for simpler transfer or modification of the model to other datasets, as the number of cells and filters can be adjusted accordingly. It has also been shown that the process of creating network architecture by repeating individual building blocks is a useful design principle in general [59].

4.2 Search Strategy

A search strategy, in this context, refers to a method of search space exploration and identification of promising neural architectures. The goal is to find an architecture that maximizes some performance measure (e.g. accuracy on a validation set on unseen data, computational cost, or other). Search strategies include random search [35], Bayesian optimization [64], evolutionary methods [57], reinforcement learning [68], and gradient-based methods [52].

When employing evolutionary algorithms (EA) in NAS, each network structure is represented as a string, and random mutations and recombinations of these strings are performed during the search process. Subsequently, each resulting string is trained and evaluated on a validation set, and the best-performing models produce offspring.

The reinforcement learning (RL) approach uses an agent to execute a sequence of actions that determine the structure of the model. The model is then trained, and its validation performance is returned as a reward, which is used to update the RNN controller. While both EA and RL methods have been successful in discovering network structures that surpass manually designed architectures, these approaches demand significant computational resources [36].

4.3 Performance Estimation Strategy

Given the importance of selecting promising neural architectures during the search process, a crucial component of NAS is *performance estimation*. This aspect allows for the evaluation and comparison of candidate architectures based on their expected performance on a given task, guiding the search strategies (described in Section 4.2) toward optimal solutions. By accurately estimating the performance of various architectures, NAS can effectively search through a vast design space and identify the most promising candidates [24].

Performance estimation plays a pivotal role in *ranking* and *selecting* architectures, ultimately guiding the search for more effective and efficient solutions. It can be done in several ways.

The simplest, straightforward approach, though with high computational demands, is to train an architecture on training data and then evaluate its performance on validation data. This strategy, directly measuring the performance of a fully trained model on the validation set, provides the most accurate performance estimation. However, it is often infeasible in practice, especially for deep learning models and large search spaces, as it requires exhaustively training a large number of architectures.

To reduce the computational burden of training each architecture from scratch, there have been developed new methods, approximating the performance of architectures, trading off some accuracy for efficiency, collectively called *performance predictors* [13].

4.3.1 Performance Predictors

Numerous methods have recently been proposed for predicting the final validation accuracy of a neural architecture by training a model on an encoding of the architecture. Gaussian processes, neural networks, and tree-based methods are some popular choices for such models [65]. However, these methods typically require hundreds of fully-trained architectures to be used as training data, leading to high initialization time. On the other hand, learning curve extrapolation methods require little or no initialization time, but each prediction involves partially training the architecture, resulting in high query time. Recently, a few techniques have been introduced that are both fast in query time and initialization time, predicting based on a single minibatch of data. Furthermore, using shared weights is a popular paradigm for NAS, although its effectiveness in ranking architectures is debated [65].

A *performance predictor* can be generally defined as any function f' which predicts the final accuracy or ranking of an architecture without fully training it. This means that evaluating f' should take less time than evaluating the *validation error* f of architecture a after training on a fixed dataset for a predetermined number of epochs E, and the set $\{f'(a) \mid a \in \mathcal{A}\}$, where $\mathcal{A}$ is a NAS search space, should ideally exhibit a high correlation or rank correlation with the set $\{f(a) \mid a \in A\}$.

According to the comprehensive study on performance predictors in NAS by White et al. [65], predictors can be categorized into the following families: *Model-based*, *Learning curve-based*, *Zero-cost Proxies*, *Weight sharing*, as depicted in Figure 4.3.

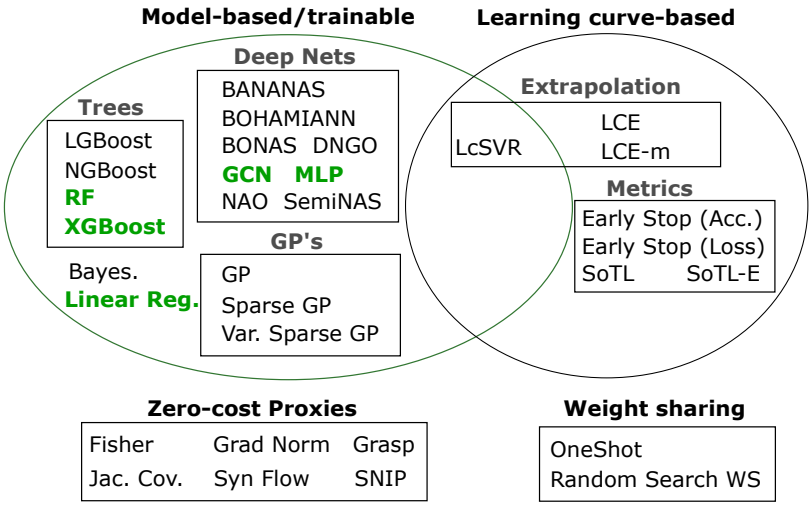


Figure 4.3: Performance Estimation Strategy families, as classified by White et al. [65]. The figure illustrates four major families: model-based, learning curve-based, weight sharing, and zero cost. Predictors employed in this thesis are highlighted in green.

Each performance predictor consists of two primary routines:

- **Initialization** routine – general pre-computation to set up the predictor
- **Query** routine – conducting architecture-specific calculations

The initialization and query routines' runtimes can vary significantly depending on the predictor type. Within NAS algorithms, the *initialization* routine is usually executed once at the beginning, while the *query* routine is performed multiple times throughout the algorithm. Some performance predictors also utilize an *update* routine to modify parts of the computation from initialization without re-executing the entire procedure (e.g., updating a model in a NAS algorithm based on newly trained architectures). The content presented in this section draws mainly from the work conducted by Elsken et al. [13].

Model-based (trainable) methods These techniques use *machine learning models* for prediction of how a particular neural architecture will perform without actually training it. In this approach, a surrogate model is trained using a dataset of neural architectures and the performance metrics that correspond to them. The model must frequently be trained using an initial set of architectures and their performance metrics, which implies a high initialization time. The query time, however, is typically brief, as surrogate models are designed for efficient inference. Various machine learning models have been used in this context, ranging from regression models, including *linear regression*, *support vector regression*, to tree-based models, including *decision trees*, *random forest*, and neural networks.

Learning curve-based methods These methods extrapolate the performance of neural architectures based on their *learning curves*. Learning curves show the progress of the training process, typically by plotting the validation loss or accuracy against training time or epochs. By observing the learning curve of a partially trained model, these methods try to predict its final performance. While the initialization time is none, as they do not require an upfront training process like surrogate models in model-based methods, the query time can be high since architectures need to be partially trained. The accuracy of the prediction depends on how well the *extrapolation* captures the actual learning behavior.

Zero-cost methods These methods estimate the performance of neural architectures without training them, by analyzing the architecture itself. They rely on simple, computationally inexpensive *heuristics* or *metrics*, such as network depth, width, floating point operations per second (FLOPs), or parameter count, to predict performance. Although these methods have low computational costs, their predictive accuracy may not be as high as other methods, especially for complex tasks.

Weight sharing methods *Weight sharing methods* leverage the idea of sharing learned weights among different architectures in the search space. These methods train a single *super-network* or *one-shot model* that encompasses all possible architectures in the search space. The performance of individual architectures is then estimated by extracting the corresponding sub-networks and their shared weights from the super-network. This approach reduces the overall training time and computational cost, as multiple architectures can be evaluated simultaneously. However, the accuracy of these methods depends on how well the shared weights generalize to the individual architectures.

4.4 Benchmarks

The fact that different NAS methods frequently do not use the same search spaces, hyperparameters, and evaluation metrics makes it difficult to reproduce the experiments and results of different NAS algorithms. Without a standardized benchmark, it can be challenging, or even impossible, to compare the performance of different NAS algorithms fairly [67]. This is where NAS benchmarks come in. NAS benchmarks are datasets with pre-trained neural network designs, and they are used to assess how well NAS techniques function by evaluating their performance. Therefore, these benchmarks provide a consistent method for assessing and contrasting the performance of NAS algorithms and determining which designs perform better on various datasets. This makes it simpler for researchers to design effective neural networks for specific applications and facilitate the reproducibility of the results. Standardized benchmarks can also provide a common ground for researchers to share their results and insights, which can accelerate the progress in the field of NAS [67].

Over time, various such NAS benchmarks have been proposed. These include, among others, NAS-Bench-101 [67] and NAS-Bench-201 [11], varying in their complexity, dataset size, search space, and performance metrics, providing a range of options for researchers to choose from depending on their specific research goals.

4.4.1 NAS-Bench-101

NAS-Bench-101 is the first publicly available dataset of CNN architectures for NAS research trying to address the issues of NAS research – namely, high computational demands for experiment reproduction, and the fact that it is challenging to credit the success of each method to the search algorithm itself because different NAS methods are not comparable to one another due to various training methods and search spaces [67].

NAS-Bench-101 search space, designed to be compact, yet expressive, consists of approximately 423,000 unique convolutional architectures, mapped to their training and evaluation metrics. All of the architectures were trained and then evaluated on CIFAR-10 dataset [30].

Implementation details

The dataset contains small feedforward structures called *cells*, which are represented as directed acyclic graphs. Each DAG consists of V nodes: an input node, an output node, and a fixed number of intermediate nodes. In each node, there is one of L labels assigned, signifying the associated operation.

Each CNN architecture has a fixed macro structure (Figure 4.4 left), consisting of a cell (specific for the given architecture) stacked 3 times, followed by a downsampling layer. This stacking and the downsampling process is repeated 3 times, followed by a global average pooling layer and a final dense softmax layer. The initial layer of the architecture is called a *stem*. This is a common way of representing a CNN architecture [21, 22].

The encoding of a cell can affect the effectiveness of NAS algorithms, and a common encoding is a 7-vertex DAG represented by a binary matrix and a list of 5 labels. This encoding has approximately 510 million total unique models, but many of them are invalid or not

computationally unique. The search space thus consists of approximately 423,000 unique graphs.

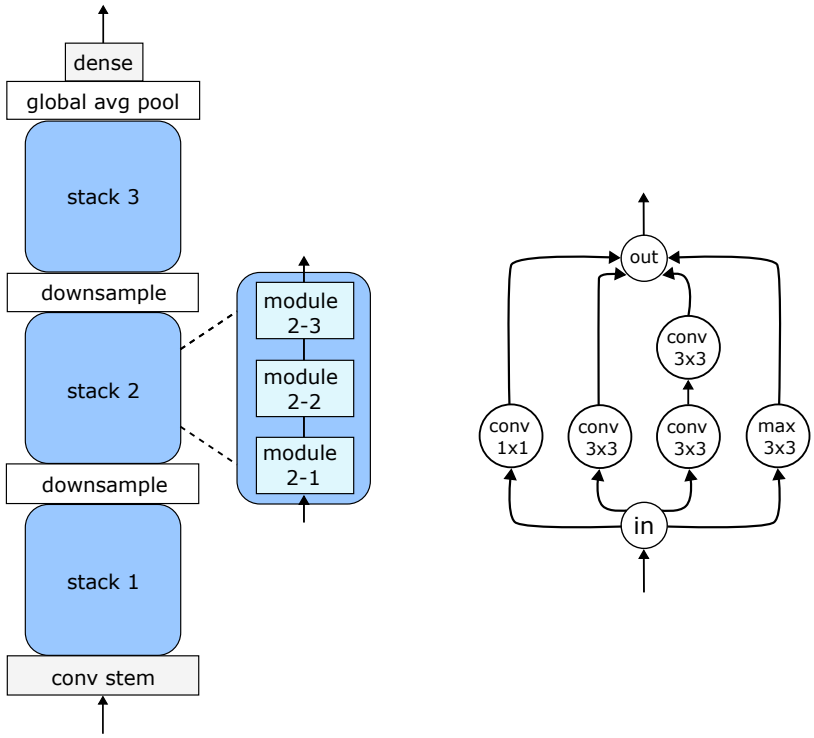


Figure 4.4: Depiction of the shared *skeleton* of CNNs in NAS-Bench-101 search space (left) and an example *module* in the form of an Inception-like cell (right). Adapted from [67].

Because of the fact that the search space made of mentioned cell-like structures would grow exponentially in both V and L, there were several *constraints* imposed:

- $L = 3$, and allows just the following operations:
  - 3×3 convolution
  - 1×1 convolution
  - 3×3 max-pool
- $V \leq 7$
- Maximum number of edges is 9.

All NAS-Bench-101 CNN models use a fixed set of hyperparameters chosen to be robust across different architectures. Each architecture is evaluated after training three times with random initializations, for each $\{4, 12, 36, 108\}$ number of epochs. The metrics of each network architecture include training accuracy, validation accuracy, testing accuracy, training time in seconds, and the number of trainable model parameters.

Chapter 5

Design of Accuracy Predictors

The aim of this chapter is to describe the programming aspects involved in analyzing and comparing various performance estimation methods that utilize regression models as predictors.

The search space encompasses neural architectures from NAS-Bench-101 (Subsection 4.4.1), utilizing the cell-based approach to represent the architecture of a CNN. For the regression model (accuracy predictor), several machine learning techniques have been selected, including linear regression (Section 2.3), random forest (Section 2.4), XGBoost (Section 2.5), MLP (Section 2.2), and GCN (Section 2.6).

The project was implemented using the Python programming language. For the implementation of linear regression and random forest models, the scikit-learn library, described in Section 5.1, was employed. It is a widely-recognized and popular ML library in Python that offers a comprehensive suite of tools for data analysis and modeling. The library is chosen for its simplicity, ease of use, extensive documentation, and the strong community support it receives. Moreover, scikit-learn provides efficient implementations of a wide range of ML algorithms, including linear regression and random forest, making it a suitable choice for the rapid development and evaluation of the models in this study.

For implementing the MLP and GCN models, the PyTorch library, described in Section 5.2, was selected. As a versatile and powerful deep learning framework in Python, PyTorch is particularly well-suited for working with neural networks. The choice of PyTorch is driven by its dynamic computation graph feature, which enables a more natural model development and debugging process. Additionally, PyTorch offers efficient GPU support for accelerated training and benefits from a rich ecosystem of complementary tools, libraries, and a vibrant community. These attributes make PyTorch an appropriate choice for developing and assessing the MLP model in the context of this study.

5.1 Scikit-Learn

In the context of this project scikit-learn, a popular ML library in Python, was employed for implementing *linear regression* and *random forest* models. The library's straightforward API and efficient implementations of these algorithms allowed for rapid development and evaluation of the models, which contributed to the overall efficiency of the project.

Figure 5.1: Scikit-learn modeling pipeline, adapted from [2].

The scikit-learn modeling pipeline, shown in Figure 5.1, involves the following steps:

- **Preprocessing** – preparing the data for modeling by cleaning, transforming, and scaling the input features. This can include techniques like handling missing values, encoding categorical variables, and normalizing or standardizing numerical features.
- **Feature Selection** – identifying the most relevant features for the specific problem at hand, which can improve model performance and reduce computational complexity.
- **Model selection** – choosing the appropriate ML algorithm for the specific problem at hand. In the context of this project, this would be the *linear regression*, *random forest*, and *XGBoost* models.
- **Model training** – fitting the selected model to the training data by adjusting its parameters to minimize the error between the predicted outputs and the actual output labels.
- **Evaluation** – assessing the performance of the trained model using various evaluation metrics, such as *mean squared error*, *coefficient of determination*, or *accuracy*, depending on the type of problem being addressed (regression or classification).
- **Hyperparameter tuning** – optimizing the model's *hyperparameters*, such as the number of trees in a *random forest* or the regularization strength in *linear regression*, to improve its performance.

5.2 PyTorch

PyTorch is a versatile and powerful deep-learning framework in Python, designed specifically for working with neural networks. Developed by Facebook's AI Research lab (FAIR), PyTorch has gained significant popularity due to its flexibility, ease of use, and dynamic computation graph capabilities. It provides powerful features for deep learning, such as tensors, autograd, dynamic computation graphs, and GPU acceleration.

Tensors Tensors, n-dimensional arrays, are the basic building blocks in PyTorch and are used to represent data. They are similar to NumPy arrays but with additional features to support deep learning applications, such as GPU acceleration.

Autograd Automatic gradient computation (Autograd) is a key feature in PyTorch that automates the computation of tensor gradients (partial derivatives) with respect to their input variables. This process is essential for optimizing model parameters during training. As a result, it simplifies the creation and training of neural networks by eliminating the need for manual gradient calculations.

Dynamic computation graph In general, computational graphs serve as a method for representing mathematical expressions or formulas. The *dynamic computation graph* in PyTorch (Figure 5.2) is a flexible way to build and compute neural network models during *runtime*. It allows developers to build and modify computational graphs on-the-fly as they execute operations, making it easier to experiment with different model architectures.

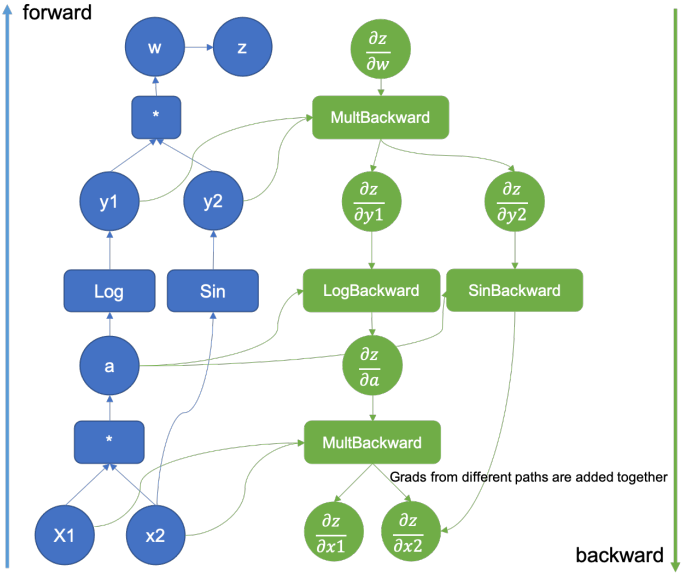


Figure 5.2: Example of an augmented computational graph in PyTorch[1], featuring input variables, operations, and output values. The graph demonstrates both the *forward* pass (data flow from inputs to outputs) and the *backward* pass (gradient computation for model parameters), including gradient nodes and sums gradients from different paths to obtain final gradient values.

[1]Image from: https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/

PyTorch provides efficient GPU support for the accelerated training of neural networks. It leverages NVIDIA's `cuDNN` library and is capable of seamlessly switching between CPU and GPU computations, allowing for faster model training and evaluation.

Choosing PyTorch over scikit-learn for implementing the MLP in this study is advantageous for several reasons. PyTorch is specifically designed for deep learning and neural networks, offering greater flexibility and customization options for the model architecture and training process. Its dynamic computation graph allows for a more intuitive development and debugging experience. Moreover, PyTorch provides efficient GPU support, which accelerates the training and evaluation of neural networks. While scikit-learn has a basic MLP implementation, it lacks the depth of features, customization options, and GPU acceleration that make PyTorch a more suitable choice for this study. Additionally, PyTorch was chosen for implementing the `NASBench101Dataset` class, representing the NAS-Bench-101 dataset, which can be found in the `dataset.py` file. The decision to use PyTorch for this purpose is based on the fact that PyTorch's `Dataset` class provides an efficient and convenient interface for handling large datasets, allowing for easy integration with other PyTorch functionalities such as data loaders, batching, and preprocessing. By inheriting from the PyTorch `Dataset` class, the `NASBench101Dataset` class gains the benefits of this interface.

5.3 Performance Predictor Specification

In this section, the specifications of the performance estimation predictor for CNNs are discussed. The design process of building (Subsection 5.4.1) and applying (Subsection 5.4.2) the predictor involves selecting an appropriate task for the CNNs, constructing a dataset, and defining features as input to the predictor. The predictor is trained on specified features and is ready to estimate the performance on unseen neural architectures.

5.3.1 Task selection

The chosen task for the CNNs in this study is *image classification*, a widely-studied problem in computer vision. Image classification involves assigning an input image to one of several predefined categories. This research utilizes the NAS-Bench-101 to search for optimal architectures in the predictor search space, focusing on architectures trained and evaluated on CIFAR-10 (Section 3.2). A detailed description of the benchmark can be found in Subsection 4.4.1.

5.3.2 Dataset construction (search space)

To construct the dataset, information from the NAS-Bench-101 dataset is utilized. The benchmark contains precomputed performance metrics and architecture descriptions for a wide range of CNNs (423,624 samples), enabling efficient comparison and evaluation. The dataset will include architecture and performance data from the benchmark, and the performance data will serve as the ground truth for training and evaluating the accuracy predictors.

5.3.3 Feature extraction and selection

Feature extraction and selection are crucial for building an effective accuracy predictor. In this study, the features will be extracted from the architecture descriptions provided by the NAS-Bench-101 benchmark.

NAS-Bench-101 information As described in the corresponding Subsection 4.4.1, the NAS-Bench-101 dataset contains *cell-based* representations of CNN architectures, with additional metadata and metrics. This information was analyzed to select the most useful features, used for training the predictor model. Multiple implementations, including Neural Predictor by Google Brain [62], suggest using of the architecture encoding and the related operations as input features, because they provide a compact representation of the architecture, capturing essential information about the connectivity and operation types.

Other potentially useful information, analyzed in Section 7, is the *number of trainable parameters* of a neural network, which can impact the model's capacity and its ability to generalize. This number is known in advance, as it can be deduced from the architecture encoding itself and can improve the performance of the predictor. Another possibly useful information could be the *architecture depth* as a feature, which is known to influence the network's capacity and the ability to learn complex patterns. In general, deeper networks can learn more complex representations, which may lead to better performance on certain tasks [53].

It is not guaranteed, that this additional information (number of trainable parameters and network depth) adds some more discriminative power to the predictor. This is why *correlation analysis* (Subsection 5.3.4) is performed on the aforementioned information. The conducted experiments in Chapter 7 compare the predictive abilities of the predictors utilizing this additional information, essentially determining whether it is useful or not in the context of NAS.

5.3.4 Correlation analysis

This technique measures the linear relationship between pairs of features and the target variable (classification accuracy). Features with a high correlation to the target variable and low intercorrelation with other features are considered more informative and relevant. By using correlation analysis to identify and select the most relevant features, the accuracy predictor's performance can be improved, while reducing the complexity and computational cost of the predictor models.

The analysis itself is conducted in the `correlation_analysis.ipynb` notebook, carefully comparing potentially useful features. The file contains a detailed examination of the NAS-Bench-101 dataset, focusing on the relationships between various architecture characteristics and performance metrics. The goal of this analysis is to determine the potential usefulness of these characteristics as features in a performance prediction model. The NAS-Bench-101 dataset is utilized to extract and analyze several architecture properties, including adjacency matrix (connections), and operations (operations). Additional properties were calculated based on the neural network information, including network depth, number of connections, the average number of connections per vertex, conv3x3 count, conv1x1

count, and max pooling count. The performance metrics include the number of trainable parameters (weights), training time, training accuracy, and validation accuracy.

The mentioned features are analyzed for their correlation with validation accuracy (Figure 5.3). The number of trainable parameters has a positive correlation ($r = 0.24$) with the target variable. Surprisingly, network depth is found to have a negative correlation with validation accuracy ($r = -0.14$). The conv3x3 count has a moderate positive linear correlation with the validation accuracy ($r = 0.46$). This suggests that as the number of 3x3 convolutional layers in the network increases, the validation accuracy tends to improve. This may be because a higher conv3x3 count might provide the network with more capacity to learn complex patterns in the data. When the problem requires learning intricate features, a network with more capacity (more convolutional layers) might perform better.

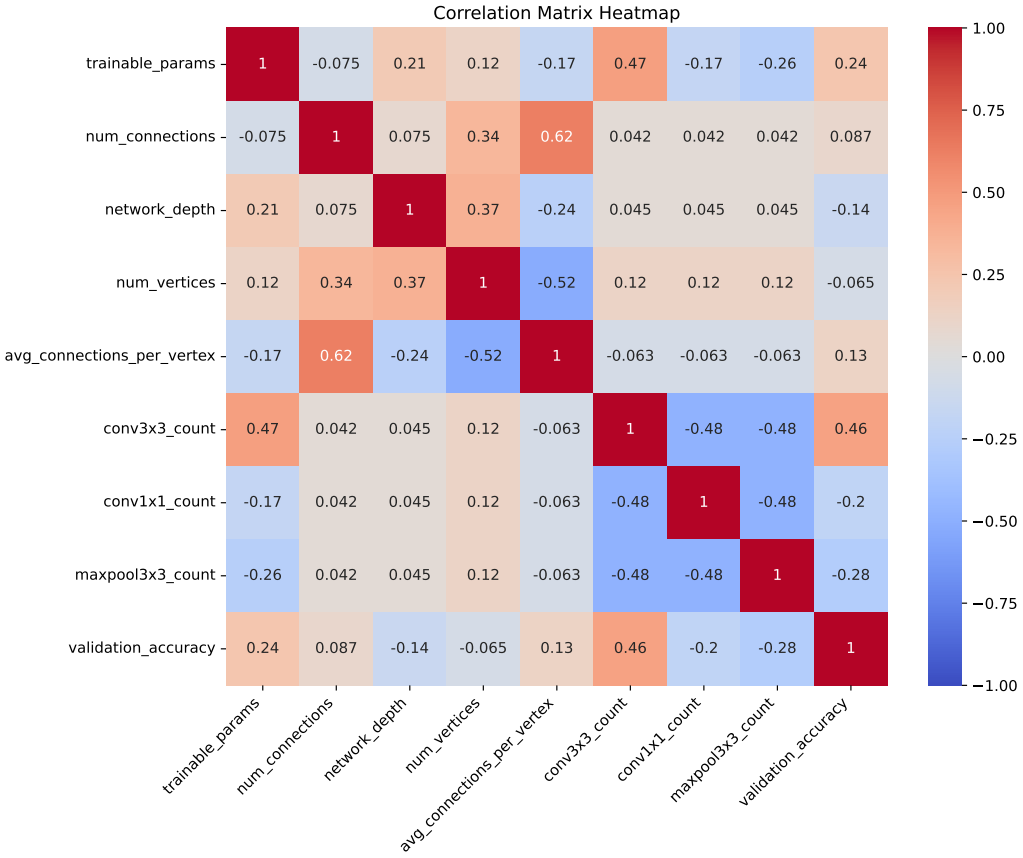


Figure 5.3: Correlation matrix of potentially useful features, extracted from the NAS-Bench-101 dataset.

One of the aims of this thesis is to investigate whether the additional features, not typically used in practice, can improve the performance of the model. The notebook provides an investigation of the relationships between various architecture characteristics and performance metrics in the NAS-Bench-101 dataset. While some features show potential for use in performance prediction models, none exhibit strong correlations with validation accuracy. This analysis serves as a foundation for further research into the usefulness of these features in predicting the performance of neural network architectures.

5.3.5 Hyperparameter tuning

A *hyperparameter tuning* strategy, conducted in `hyperparameter_tuning.ipynb` notebook, is employed to optimize the performance of the predictor models, systematically searching for the best combination of hyperparameters for each ML method, such as the number of decision trees and maximum depth for random forests, or the number of hidden layers, neurons per layer, and activation functions for MLPs.

The *random search* technique, combined with *cross-validation* (Subsection 5.3.6), was also utilized to ensure that the chosen hyperparameters generalize well to unseen data, leading to higher accuracy and better generalization to new CNN architectures, ultimately contributing to a more efficient NAS process.

5.3.6 Cross-validation strategy

Cross-validation is an essential aspect of training ML models, as it helps to mitigate overfitting and provides an unbiased evaluation of their performance. In this work, a *k-fold cross-validation* strategy (Figure 5.4), implemented using the `KFold` function from the scikit-learn library (Section 5.1), is used within the hyperparameter tuning strategy (Subsection 5.3.5). *K-fold cross-validation* involves partitioning the dataset into k equally-sized subsets, or "folds". The training and validation process is then repeated k times, with each fold serving as the validation set exactly once, while the remaining folds are used for training. This ensures that every data point is used for both training and validation, providing a comprehensive evaluation of the model's performance.

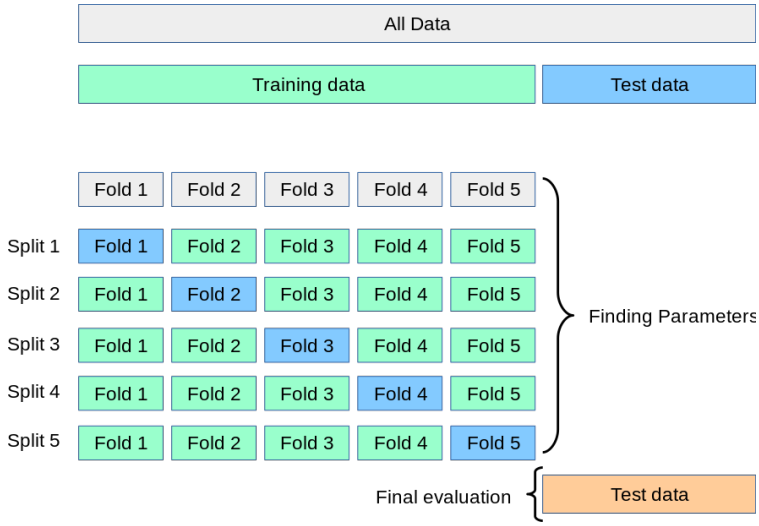


Figure 5.4: K-fold cross-validation from scikit-learn[2].

The primary advantage of k-fold cross-validation, as implemented by the `KFold` function in scikit-learn, is its ability to generate a more reliable estimate of the model's performance on unseen data. By averaging the performance metrics, such as MAE, or R^2, across the k iterations, a more accurate measure of the model's predictive capabilities can be obtained. This cross-validation strategy helps to ensure that the chosen predictor models are robust and generalize well to new CNN architectures.

[2]Image source: https://scikit-learn.org/stable/modules/cross_validation.html

5.3.7 Objective functions

The selection of objective functions for performance predictors in NAS is crucial. This work employs the following objective functions commonly used in regression analysis:

- *Mean Squared Error* (MSE): MSE averages the squared differences between predicted and actual performance values. It is mathematically defined for n pairs of actual (y) and predicted ($\hat{y}$) values as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.1}$$

  MSE is prevalent in regression problems and emphasizes larger errors due to the squaring operation [18].

- *Mean Absolute Error* (MAE): MAE computes the average absolute difference between the estimated and actual performance values. It is given by:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{5.2}$$

  Unlike MSE, MAE is less sensitive to large errors and outliers, thereby providing a more robust estimation when the error distribution has significant outliers [66].

- *Pearson's Correlation Coefficient* (PCC): PCC measures the linear correlation between the predicted and actual performance values. It ranges from -1 to 1, with 1 indicating a perfect positive correlation, -1 a perfect negative correlation, and 0 no correlation. It is defined as:

$$PCC = \frac{\sum_{i=1}^{n}(y_i - \bar{y})(\hat{y}_i - \bar{\hat{y}})}{\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2(\hat{y}_i - \bar{\hat{y}})^2}} \tag{5.3}$$

  where $\bar{y}$ and $\bar{\hat{y}}$ are the means of the observed and predicted values, respectively [41].

- *Coefficient of Determination* (R^2): R^2 computes the proportion of variance in the actual performance values that is explained by the predictor model:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{5.4}$$

  where $\bar{y}$ is the mean of the observed values y_i. A higher R^2 denotes a better model fit [41].

- *Kendall's Tau*: Kendall's Tau is a rank correlation coefficient measuring the similarity between two sets of rankings [27]. Given two sets of rankings R_1 and R_2, Kendall's Tau is calculated as:

$$\tau = \frac{n_c - n_d}{\frac{1}{2}n(n-1)} \tag{5.5}$$

  where n_c is the number of concordant pairs, n_d is the number of discordant pairs, and n is the number of items [27]. In the context of NAS, Kendall's Tau assesses how well the predictor can identify the most promising architectures in terms of their *ranking* rather than exact accuracy values.

5.4 Performance Predictor Structure

In this section, the focus is on providing an overview of the overall structure of the predictor and illustrating how the chosen ML methods are applied to predict classification accuracy.

The selected ML methods are applied to the dataset by first training the models on the extracted and selected features from the CNN architectures in the NAS-Bench-101 benchmark. The trained models are then used to *predict the classification accuracy* of unseen CNN architectures in the search space. The results obtained from applying each predictor are compared with one another.

5.4.1 Building an accuracy predictor

The procedure of building an accuracy predictor, depicted in Figure 5.5, is characterized by a series of critical steps, each specifically crafted to maximize both the efficiency and accuracy of the predictor.

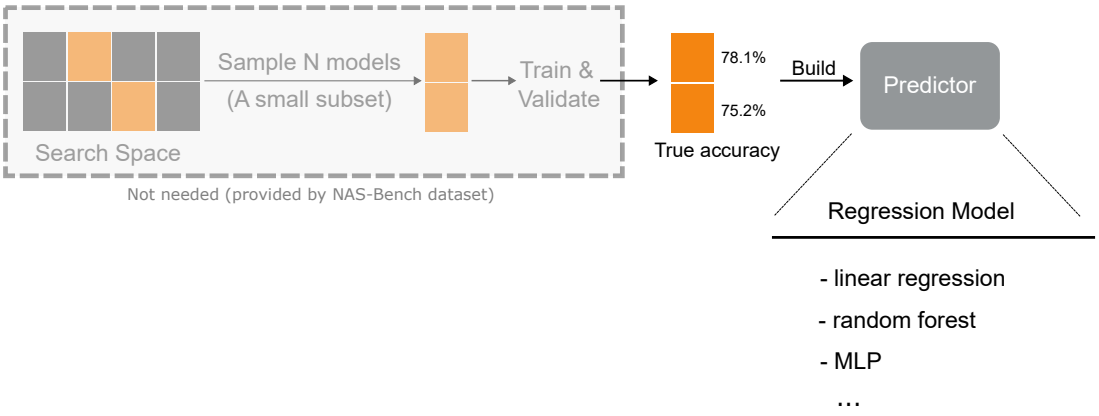


Figure 5.5: Typical *building* process of a performance predictor for NAS, inspired by [62].

The first step entails sampling N neural architectures from a specified search space. As discussed in Section 5.4.2, striking a balance between computational cost and time is crucial, so the number N should not be excessively large to maintain the predictor's usefulness. This is because all N sampled models must be trained and validated to determine their true accuracies, and training neural networks is a resource-intensive process.

The resulting dataset, comprising neural network architectures and their corresponding true accuracies, enables the predictor to estimate the performance of unseen architectures. Finding an optimal balance for the number of sampled architectures N is essential—large enough to effectively train the predictor, yet small enough to reduce the burden of neural network training. This trade-off is critical to achieve the best results.

Figure 5.5 shows that the first stages of building a predictor (reduced opacity) are not required for this study. This is because the dataset has already been collected in advance using the NAS-Bench-101 benchmark, which provides information on numerous CNN architectures and their true performance metrics. Consequently, the initialization process is significantly simplified.

To train a performance predictor, it is essential to extract relevant features that represent the neural network architectures within the search space. As previously mentioned, in NAS the choice of N (number of architectures used for predictor training) depends on the compute budget one has. In this work, however, we can afford to increase the value of N substantially, as we do not need to exhaustively train and evaluate each architecture.

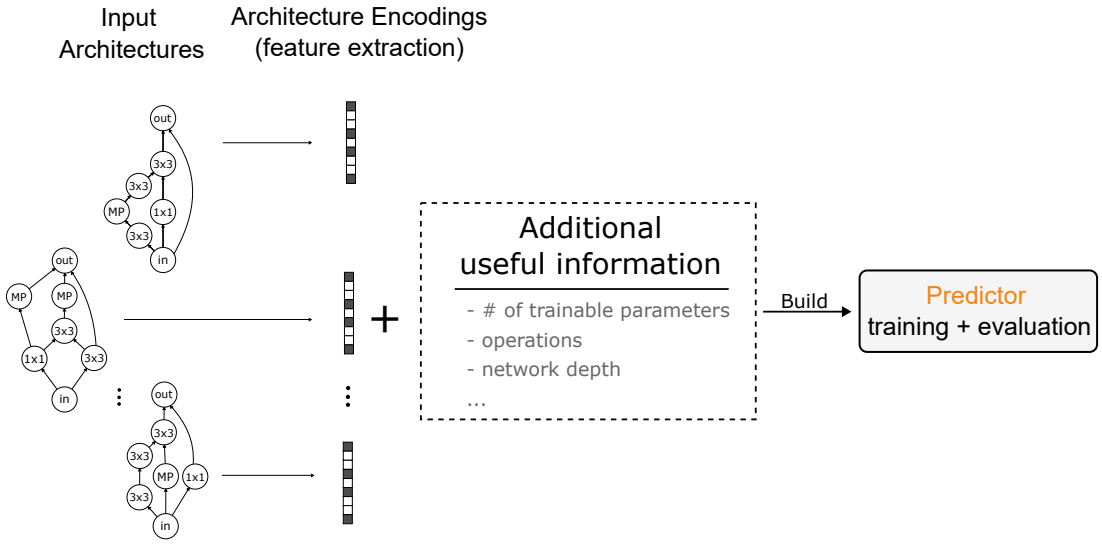


Figure 5.6: Detail view of the typical predictor building process, showing feature extraction and selection, leading to the building of the predictor.

As depicted in Figure 5.6, the process of building an accuracy predictor involves gathering encodings for each network architecture in the training set, including operations such as max pooling (Subsection 3.1.3) and convolution (Subsection 3.1.1). Additionally, the number of trainable parameters for a specific network architecture and the network depths are considered (Subsection 5.3.4), as they both can be calculated in advance from the network structure. In the first approach, this information is flattened and fed into the predictor for training and evaluation (Experiment Sets in Sections 7.2 and 7.3). Conversely, the second approach, utilizing GCN capabilities (Section 2.6) aims to preserve the architecture's structure by retaining the DAG graph structure, which could potentially lead to a better predictive power (Experiment Set in Section 7.4).

To ensure the ML models (predictors) generalize well to unseen data, a suitable training data split strategy is employed. One common approach is to use *cross-validation* (Subsection 5.3.6), where the training data is divided into k equal-sized subsets (folds), with each subset being used as a validation set exactly once, while the remaining $k-1$ subsets form the training set. The model is then trained and evaluated k times, and the average performance across all iterations is calculated. This process reduces the likelihood of overfitting and helps to obtain a more reliable estimate of the model's performance.

5.4.2 Applying an accuracy predictor

Figure 5.7 illustrates the process of employing trained predictor models to make predictions for new CNN architectures. The gray rectangle on the right side of the figure (reduced opacity) represents the portion of the performance predictor application process that is not directly carried out in this work. This is because the search space used in this study is based on the NAS-Bench datasets (specifically NAS-Bench-101), which already provide the necessary information.

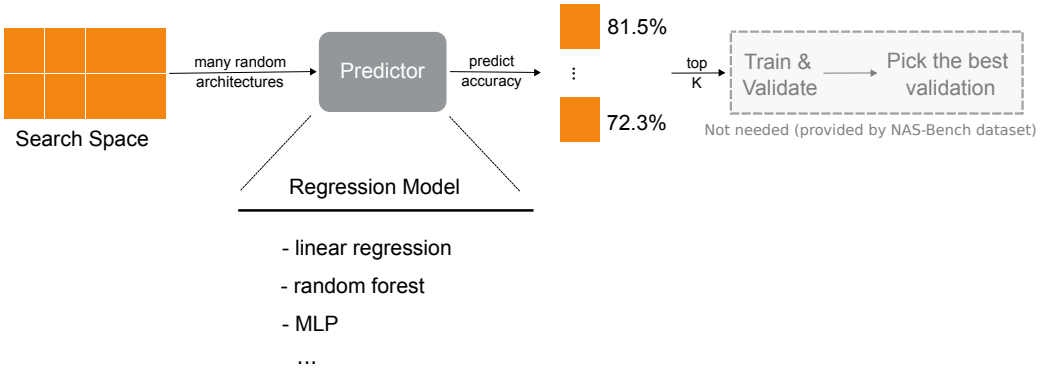


Figure 5.7: Typical way of *applying* the performance predictor for NAS, inspired by [62].

Once the predictor models have been trained and fine-tuned using the existing dataset, they can be leveraged to estimate the classification accuracy of unexplored CNN architectures. This involves inputting numerous random (unseen) architectures into the predictor, which in turn estimates their accuracy. Subsequently, the top K architectures with the highest predicted accuracy are selected as the most promising candidates. These K architectures must then undergo exhaustive training and evaluation to obtain their actual metrics. As noted in Subsection 5.4.1 regarding the value of N architectures during training, it is also crucial to strike a balance for the number K. Ideally, K should be large enough to account for the predictor's imperfect accuracy estimations while remaining small enough to minimize computational resources and time spent on training the architectures.

By leveraging the predictor's ability to estimate performance rapidly, the NAS process can be guided more efficiently in the search for optimal architectures. The predictor models can be used to prioritize the evaluation of architectures that are expected to yield high performance, thereby reducing the time and computational resources spent on evaluating suboptimal architectures. As the NAS process iterates, the predictor models can be updated with new data and continually refined, further improving the efficiency of the search. By incorporating the predictor models into the NAS process, it becomes possible to explore the vast search space of CNN architectures more effectively and identify high-performing architectures with significantly reduced time and computational cost.

Chapter 6

Implementation

In this chapter, the program implementation, structure, and its usage are described. The design of the program is described in Chapter 5. Python programming language was chosen due to its versatility and extensive support for scientific and machine learning libraries. Among the many libraries utilized, scikit-learn and PyTorch stood out as particularly relevant for implementing the models, calculating relevant metrics, and for dataset manipulation. The implementation of performance predictors was inspired by the NASLib[1] project.

6.1 Program Structure

The codebase for this project has been thoughtfully structured to emphasize modularity and ease of use. By organizing the components as described below, the implementation allows for straightforward extension and customization. The `Predictor` base class (in `predictor.py`), adapted from NASLib project [47], serves as the foundation for all predictor models: linear regression (in `linear_reg.py`), random forest (in `random_forest.py`), XGBoost (in `xgb.py`), MLP (in `mlp.py`), and GCN (in `gcn.py`). All predictors are trained, evaluated, compared, and analyzed in the Jupyter Notebook `analysis.ipynb`. NAS-Bench-101 dataset handling is represented by the `NASBench101Dataset` class (in `dataset.py`). A collection of useful functions can be found in `utils.py` file, containing functions for plotting, cross-validation, and other useful operations. By maintaining a well-structured organization within the codebase, each component can focus on specific functionality, ultimately making the entire project easier to understand, maintain, and extend. The project's source code, including the mentioned files, along with a comprehensive user manual, is publicly available on GitHub[2].

6.2 Dataset representation

In this study, the publicly accessible and open-source dataset NAS-Bench-101 (Subsection 4.4.1) is employed. This dataset encompasses a variety of CNN architectures represented as fixed-structure cells and includes key metrics like validation accuracy, test accuracy, and training time. The dataset comes with an official public API hosted on GitHub[3], which simplifies its utilization.

[1] NASLib: https://github.com/automl/NASLib
[2] BP-Accuracy-Predictors: https://github.com/xsmida03/BP-Accuracy-Predictors
[3] Google-Research NAS-Bench-101: https://github.com/google-research/nasbench

The NAS-Bench-101 dataset, assembled by the Google-research team, is officially stored in `nasbench.tfrecord` files. The full dataset, containing 5 million data points for all four epoch lengths $\{4, 12, 36, 108\}$, and a smaller subset featuring models trained only for 108 epochs can be accessed via the official Google-research GitHub repository[4].

```
# Load the data from file (this will take some time)
nasbench = api.NASBench('/path/to/nasbench.tfrecord')

# Create an Inception-like module (5x5 convolution replaced with two 3x3
# convolutions).
model_spec = api.ModelSpec(
    # Adjacency matrix of the module
    matrix=[[0, 1, 1, 1, 0, 1, 0],    # input layer
            [0, 0, 0, 0, 0, 0, 1],    # 1x1 conv
            [0, 0, 0, 0, 0, 0, 1],    # 3x3 conv
            [0, 0, 0, 0, 1, 0, 0],    # 5x5 conv (replaced by two 3x3's)
            [0, 0, 0, 0, 0, 0, 1],    # 5x5 conv (replaced by two 3x3's)
            [0, 0, 0, 0, 0, 0, 1],    # 3x3 max-pool
            [0, 0, 0, 0, 0, 0, 0]],   # output layer
    # Operations at the vertices of the module, matches order of matrix
    ops=[INPUT, CONV1X1, CONV3X3, CONV3X3, CONV3X3, MAXPOOL3X3, OUTPUT])

# Query this model from dataset, returns a dictionary containing the metrics
# associated with this model.
data = nasbench.query(model_spec)
```

Listing 6.1: Example usage of NAS-Bench-101 dataset, from official Google-Research GitHub (see footnote 4).

As acknowledged in the official NAS-Bench-101 publication [67], loading the entire `tfrecord` is a time-consuming process. Consequently, alternative formats have been developed to enable faster loading. In this study, the `hdf5` binary file representing the NAS-Bench-101 dataset is used. It can be downloaded from Google Drive[5]. This file format supports efficient read and write operations for sizable and intricate datasets, making it faster to load than the official `tfrecord` files. The decision to convert from `tfrecord` to `hdf5` is driven by the need to save both time and computational resources during the loading process. An open-source script allowing conversion from NAS-Bench-101 `tfrecord` file to `hdf5` format can be found in the Neural Predictor implementation on GitHub[6]. This script retains all the essential information about neural architectures from the NAS-Bench-101 dataset.

NAS-Bench-101 dataset handling The `NASBench101Dataset` class representing the NAS-Bench-101 dataset can be found in the `dataset.py` file. This class, leveraging PyTorch `Dataset` class capabilities, was created to manage and interact with the NAS-Bench-101 dataset in an efficient and convenient manner. It serves several purposes: it preprocesses the dataset to be used in ML models, facilitates easy access to individual data points, and provides utility functions for normalizing, denormalizing, and computing various dataset properties. The class takes the path to the preprocessed `hdf5` file and an optional dataset split as input arguments. It initializes the dataset by loading various attributes such as hash, number of vertices, trainable parameters, adjacency matrices, operations, and metrics from the `hdf5` file, which are used in the context of the performance estimation process.

[4] Google-Research NAS-Bench-101: https://github.com/google-research/nasbench
[5] NAS-Bench-101 `hdf5` file: https://drive.google.com/open?id=1x1EQCyClzHBVDHloUCtES_-M_E9o4MeF
[6] Neural Predictor GitHub: https://github.com/ultmaster/neuralpredictor.pytorch

The `NASBench101Dataset` class provides methods to obtain the length of the dataset, retrieve individual items by index, normalize and denormalize values, and compute the mean and standard deviation of accuracies. Furthermore, it includes private helper methods to resample accuracies, check if an accuracy value is below a certain threshold (addressing NAS-Bench-101 noise), compute network depth, and convert operation indices to a one-hot encoded format. The class also offers methods for obtaining the necessary feature and target encodings for each architecture in the dataset, as required by specific machine learning models.

6.3 Predictor Implementations

In this section, a detailed overview of the predictor implementations is provided. Figure 6.1 illustrates the inheritance structure of the individual predictors, with each subclass representing a specific ML model.

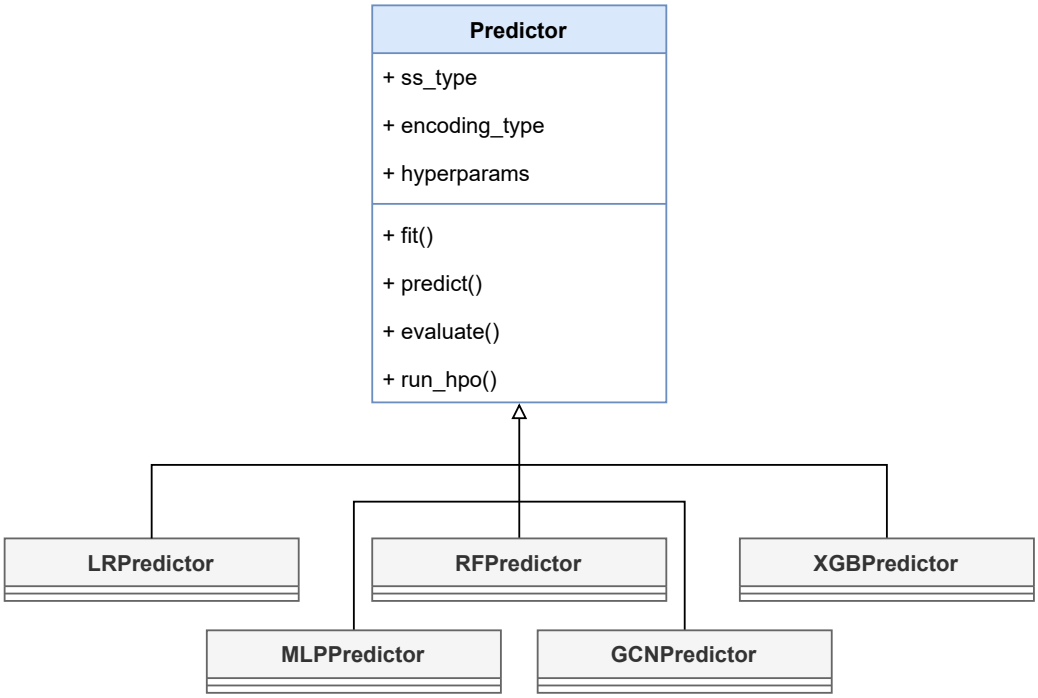


Figure 6.1: The structure of the individual predictors in the implementation. The base class `Predictor` ensures a consistent interface and provides the essential methods (only the most important ones are illustrated) and properties for each subclass, which represents a specific machine learning model (e.g., Linear Regression, Random Forest, XGBoost, MLP, and GCN). Each subclass inherits from the `Predictor` class and can override or extend its methods and properties as needed.

6.3.1 The `Predictor` base class

The base class, `Predictor`, defines the essential methods and properties that are shared by all individual predictors. These methods include `fit()`, `predict()`, `refit()`, `evaluate()`, and `run_hpo()` (for hyperparameter optimization). The base class also provides meth-

ods for setting and getting hyperparameters and methods for saving and loading predictor models. The `Predictor` class serves as a blueprint for creating specific predictor implementations. Each subclass inherits from this base class and can override or extend its methods and properties to tailor the functionality specific to the machine learning model it represents. All inheriting predictors share the same interface for fitting, querying, and evaluating the models, ensuring a consistent and unified approach to implementing various machine learning algorithms.

The `fit()` and `predict()` methods are the core methods for training and querying the predictor. The `refit()` method allows updating the predictor with new training data. The `evaluate()` method calculates various evaluation metrics, such as mean absolute error (MAE), and correlation coefficients. The `run_hpo()` method enables hyperparameter optimization, facilitating the search for optimal predictor settings.

All implemented accuracy predictors, including `LRPredictor`, `RFPredictor`, `XGBPredictor`, `MLPPredictor`, and `GCNPredictor`, inherit from the `Predictor` base class and utilize the provided methods and properties, customizing them according to the specific machine learning algorithm used. This modular design allows for easy integration of additional predictors in the future while maintaining a consistent and unified structure.

Linear-Regression-based predictor The `LRPredictor` class is a linear regression predictor (Section 2.3) inheriting from the `Predictor` base class. It serves as a simple *baseline* model for regression tasks, suitable for comparing the performance of more complex models.

Random-Forest-based predictor The `RFPredictor` class is a random forest regression predictor (Section 2.4) that inherits from the `Predictor` base class. It serves as a more sophisticated model compared to the linear regression-based predictor, offering a robust and accurate estimation of the performance of neural network architectures.

XGBoost-based predictor The `XGBPredictor` class is an XGBoost regression predictor that inherits from the `Predictor` base class. It serves as a powerful and accurate model for estimating the performance of neural network architectures, leveraging the strengths of the XGBoost algorithm (Section 2.5).

MLP-based predictor The `MLPPredictor` class is an MLP regression predictor (Section 2.2) that inherits from the `Predictor` base class. It offers a flexible and powerful option for regression tasks, leveraging the expressive power of MLPs to estimate the performance of neural network architectures.

GCN-based predictor The `GCNPredictor` class is a GCN regression predictor (Section 2.6) adapted from the Neural Predictor [62] implementation on GitHub[7], inheriting from the `Predictor` base class. Unlike the previous predictors, it uses a more sophisticated encoding of the features to handle the input data and improve the predictions. The predictor leverages the power of GCNs to better capture the underlying structure of neural architectures, providing a more accurate performance estimation compared to other predictors.

[7]Neural Predictor: https://github.com/ultmaster/neuralpredictor.pytorch

6.4 Accuracy predictors – analysis

The root of the work happens in the file `analysis.ipynb`, which is a Jupyter Notebook document. Jupyter Notebook was the preferred tool for conducting the analysis and comparison of performance predictors, correlation analysis (found in `correlation_analysis.ipynb`), and hyperparameter tuning (contained in `hyperparameter_tuning.ipynb`). This preference stems from its interactive and intuitive environment, which facilitates seamless integration of code execution, data visualization, and documentation within a singular platform. This unique functionality of Jupyter Notebook enhances the transparency and reproducibility of the research by enabling researchers to document their thought processes and share insights directly alongside the code. Moreover, the capability to execute code cells independently fosters an efficient environment for experimentation, while the built-in support for data visualization promotes a more comprehensive understanding of the results.

The `analysis.ipynb` notebook initiates the process by loading the NAS-Bench-101 dataset and extracting relevant features for the machine learning models. These features encompass both standard and extended feature sets (as outlined in Sections 7.2 and 7.3) along with DAG-based features, which are specific to the GCN model. Subsequently, the individual predictors are trained using their corresponding feature sets, with the training time being meticulously measured and displayed across three training runs.

Upon the completion of the training phase, the predictors are evaluated and applied to a 100 000-sample subset of the comprehensive NAS-Bench-101 dataset. This application aims to assess their metrics and properties. Concurrently, the results are visualized and plotted to facilitate an intuitive understanding of the performance and efficiency of the predictors. This comprehensive pipeline implemented within the Jupyter Notebook ensures an effective, clear, and reproducible research process.

Chapter 7

Experimental results

The primary objective of the experiments conducted in this study is to evaluate and compare the performance of various accuracy predictors, namely, linear regression, random forest, XGBoost, MLP, and GCN, when applied to NAS. These experiments are divided into three sets:

- The first set of experiments (Section 7.2) utilizes a conventional approach for performance estimation, which involves using flattened architecture encoding and one-hot encoded operations as input features. The goal is to facilitate a comparison among various model-based accuracy predictors used in this study.
- The second set of experiments (Section 7.3), extends the input features to include not only the architecture encoding and operations but also the number of trainable parameters and the number of 3x3 convolution operations. These additional features were selected based on their strong correlations, as discovered in our correlation analysis (Subsection 5.3.4). The purpose of this set of experiments is to test whether the incorporation of this supplementary information enhances the predictive capacity of the accuracy predictors, and if so, to evaluate the associated computational cost.
- The final set of experiments consists of a singular experiment (Experiment 7.4.1) that adopts a unique approach. Rather than flatten the architecture encoding, this method employs Graph Neural Networks, specifically GCNs, maintaining the architecture encoding information in the form of directed acyclic graphs. This method, simulating Google Brain's Neural Predictor [62], is anticipated to improve performance, as it preserves the additional information inherent in the graph-like structure of the encoding [63].

Throughout the experiments, a subset of the NAS-Bench-101 benchmark is employed to provide a comprehensive evaluation of the predictors in its search space. The performance of each predictor is assessed using different evaluation metrics (Subsection 5.3.7). The experimental results will be used to draw conclusions on the effectiveness of the proposed predictors, based on the type of the predictor, and the encoding of input features, and to identify potential areas for future research and improvements.

7.1 Experiment setup

The experiment setup is carefully designed to ensure a robust and comprehensive evaluation of the performance predictors. The idea behind the chosen experimental setup is to provide a fair comparison between the predictors and facilitate the identification of the most effective approaches for the NAS task. The process of training and evaluating the performance predictors involves the following steps:

- **Preprocessing the dataset**: The NAS-Bench-101 dataset is preprocessed (file `dataset.py`) to make its usage more comfortable and extract the necessary features for both the flattened architecture encoding (both basic and extended) and the DAG-based encoding. For more related information, refer to Section 6.2.
- **Training the predictors**: ML models are trained using the preprocessed dataset. Each model was trained on a dataset consisting of 172 unique architecture-accuracy pairs. The selection of this specific number of training data was inspired by Google Brain's Neural Predictor research paper [62], as it provides a good compromise between the values of N (number of architectures used to train a predictor) and K (number of the most promising architectures selected by applying a predictor), for more details refer to the Subsection 5.4.1. Hyperparameter tuning was performed in advance (in `hyperparameter_tuning.ipynb` notebook), optimizing the performance of each predictor model. Default hyperparameters are described in Sections 7.2 and 7.4, for each corresponding ML model used in a given experiment. The performance measures were normalized using their mean and standard deviation.
- **Applying the predictors**: The trained models are evaluated on unseen data, comprising a subset of the NAS-Bench-101 dataset of 100 000 CNNs, simulating the process of predicting the performance of novel CNN architectures. This enables a realistic assessment of the predictors' ability to guide the NAS process efficiently.

The chosen metrics (Subsection 5.3.7) are used to assess the predictors' performance. It is important to note that *rank-based* metrics, such as Kendall's Tau, are particularly relevant for NAS tasks because the *ranking* of predictions is often more important than the actual values of predictions. This is because the primary goal of NAS is to identify the best-performing architectures rather than to precisely estimate their performance. Rank-based metrics provide a more robust measure of the predictors' ability to correctly rank architectures in terms of their performance, which is crucial for guiding the search process efficiently [65].

7.1.1 Hardware and Software Configurations

The hardware components utilized in the experiments include:

- CPU: `Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz`
- GPU: `NVIDIA GeForce GTX 1650`
- RAM: `32 GB`

The required software used in this work is described along with a user manual on GitHub[1].

[1] https://github.com/xsmida03/BP-Accuracy-Predictors

7.2 Experiment Set 1: Conventional Approach

This set of experiments employs a common approach in feature selection for performance estimation strategies. It utilizes a flattened architecture encoding and one-hot encoded operations, concatenated into a single vector. This vector subsequently serves as an input for the accuracy predictors.

Objective The aim of these experiments is to evaluate and compare the performance of selected accuracy predictors, using conventional input features: an adjacency matrix and operations.

Features The features used in the ML models for these experiments typically describe the cell-based search space of NAS-Bench-101. These include the matrix of operations used in a CNN and the adjacency matrix that represents the connections between the individual operations.

Consider an arbitrary CNN architecture (single cell) employing the cell-based encoding (Section 4.1) from the NAS-Bench-101 dataset illustrated in Figure 7.1.

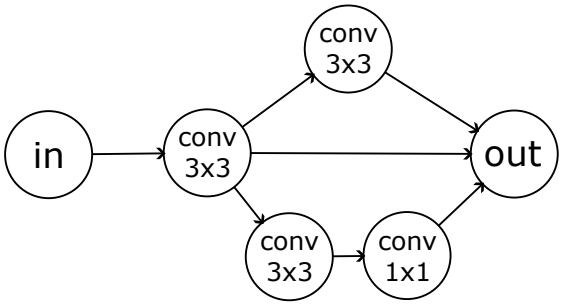


Figure 7.1: An arbitrary CNN cell architecture, from the NAS-Bench-101 dataset.

Consider the following operation types in the NAS-Bench-101 architecture:

```
[
 'input',
 'conv3x3-bn-relu',
 'conv3x3-bn-relu',
 'conv1x1-bn-relu',
 'conv3x3-bn-relu',
 'output'
]
```

These operations are first encoded into integers, then padded, and finally one-hot encoded:

```
Encoded operations (int): [-1 0 0 1 0 -2]

Encoded and padded: [-1 0 0 1 0 -2 0]
```

One-hot encoded operations:

```
[[0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 1, 0, 0],
 [1, 0, 0, 0, 0],
 [0, 0, 1, 0, 0]]
```

The adjacency matrix represents the connections between vertices in a DAG, corresponding to a cell in a CNN architecture. For instance, an adjacency matrix might appear as follows:

```
[[0, 1, 0, 0, 0, 0, 0],
 [0, 0, 1, 0, 1, 1, 0],
 [0, 0, 0, 1, 0, 0, 0],
 [0, 0, 0, 0, 0, 1, 1],
 [0, 0, 0, 0, 0, 1, 0],
 [0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 0]]
```

These two features, the one-hot encoded operations, and the adjacency matrix, are flattened and concatenated into a single 84-element vector, which serves as the input to each accuracy predictor in this set of experiments.

7.2.1 Linear Regression Predictor

Setup The `LRPredictor` class, employing the scikit-learn's `LinearRegression` model, uses the library's default parameters: `fit_intercept=True` (calculate the y-intercept), `normalize=False` (manual normalization), `copy_X=True` (prevent overwriting input data), `n_jobs=None` (single processor), and `positive=False` (coefficients in any direction).

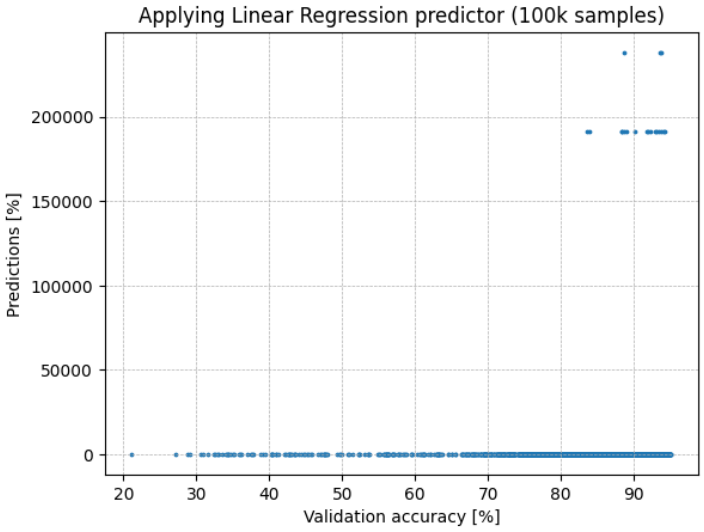


Figure 7.2: Predicting validation accuracies on a set of 100k CNN architectures with the Linear Regression-based predictor.

Results The Linear Regression predictor's performance, as depicted in Figure 7.2, indicates the presence of several outliers beyond the realistic accuracy range of 0 to 100 %. These extreme estimations are likely due to the model's linear nature and its limitations in capturing complex, nonlinear patterns within the NAS-Bench-101 dataset. Given the inherent simplicity of the Linear Regression model, this result was anticipated and aligns with the initial expectations.

7.2.2 Random Forest Predictor

Setup The RFPredictor class, utilizing `sklearn`'s `RandomForestRegressor`, is set up with a specific set of default hyperparameters: 116 trees (`n_estimators`), a maximum number of features considered at each split as approximately 17 % (`max_features`), and minimum samples required to be at a leaf node and to split an internal node both set to 2 (`min_samples_leaf` and `min_samples_split`). Furthermore, `bootstrap` is set to `False`, indicating that the whole dataset is used to build each tree.

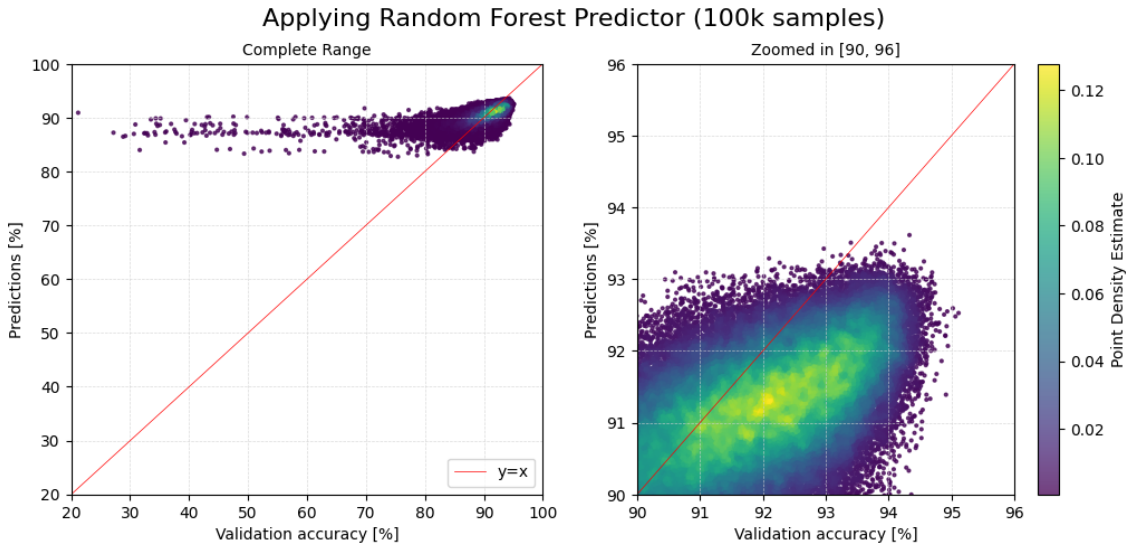


Figure 7.3: Predicting validation accuracies on a set of 100k CNN architectures with the Random Forest-based predictor.

Results The Random Forest predictor, being a more advanced model, yielded significantly better results. Figure 7.3 illustrates its predictions, with the majority of the neural networks clustered in relatively close proximity. This approach however tends to underestimate the majority of architectures as can be seen on the right subplot of Figure 7.3. The left subplot of Figure 7.3 reveals certain data points with vastly inaccurate validation accuracy predictions. This discrepancy is likely a consequence of noise present in the NAS-Bench-101 dataset, also addressed by the Google Brain researchers [62], where the majority of the CNN architectures demonstrate a validation accuracy exceeding 90 % and there are not enough examples of architectures with lower validation accuracies. This phenomenon is also examined in the `correlation_analysis.ipynb` notebook, which provides an in-depth analysis of the NAS-Bench-101 dataset. The corresponding initialization and query time can be found in Table 7.1, and the measured metrics are listed in Table 7.2.

7.2.3 XGBoost Predictor

Setup The `XGBPredictor` class uses the XGBoost Regressor model from the `xgboost` library, configured with the following default hyperparameters: a gbtree `booster`, an objective of reg:squarederror, and an evaluation metric of root mean square error (rmse). The maximum tree depth (`max_depth`) is set to 6, the minimum sum of instance weight (`min_child_weight`) is 1, and the subsample ratio of columns for each tree and each split (`colsample_bytree` and `colsample_bylevel`) are both 1. The learning rate is initially set to 0.3.

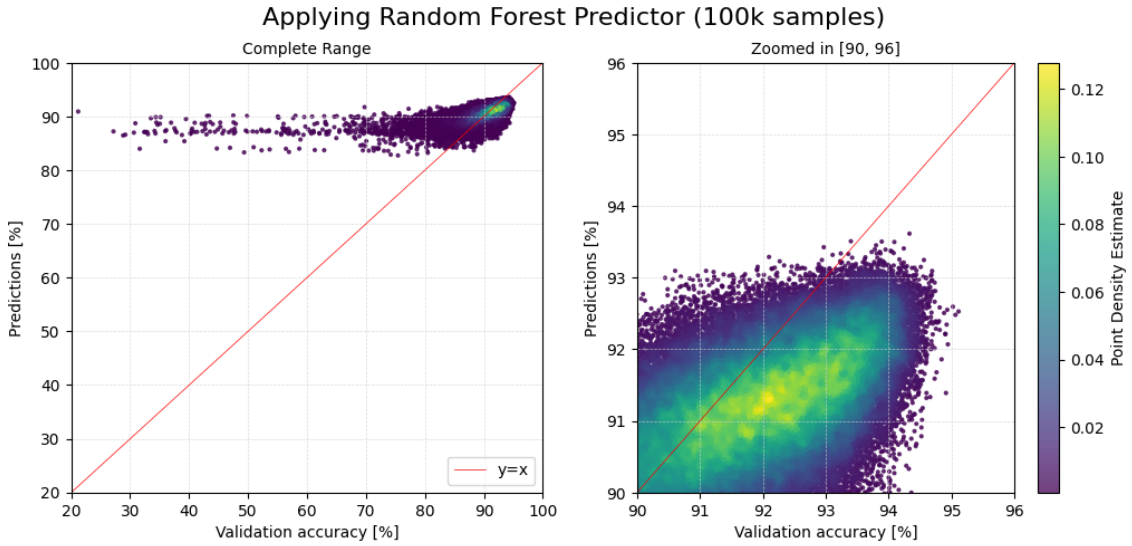


Figure 7.4: Predicting validation accuracies on a set of 100k CNN architectures with the XGBoost-based predictor.

Results The XGBoost approach showed even more improved performance of the performance prediction on the same set of data. Most of the predicted validation accuracies is located near the *perfect prediction line* $y = x$ (red). When compared to the Random Forest model (Subsection 7.2.2), XGBoost does not tend to underestimate the validation accuracies for the densest areas of the plot, resulting in better accuracy estimation. The corresponding initialization and query time can be found in Table 7.1, and the measured metrics are listed in Table 7.2.

7.2.4 MLP Predictor

Setup The `MLPPredictor` class uses a Feedforward Neural Network model from the PyTorch library, embodied in the `FeedforwardNet` class. The model, stored in the `self.model` instance variable, is configured with `num_layers`=20 and `layer_width`=20 units for each layer as the default hyperparameters. The model's activation function can be customized and is set to ReLU by default. The model is trained using the Adam optimizer with a learning rate (`lr`) of 0.001, and MSE as the loss function. To prevent overfitting, L1 regularization is also applied with a regularization strength (`regularization`) of 0.2. The model is trained for `epochs`=500 with a `batch_size` of 32. The target values (accuracies), stored in `ytrain`, are normalized by subtracting the mean (`self.mean`) and dividing by

the standard deviation (`self.std`) before training, and these transformations are reversed when making predictions.

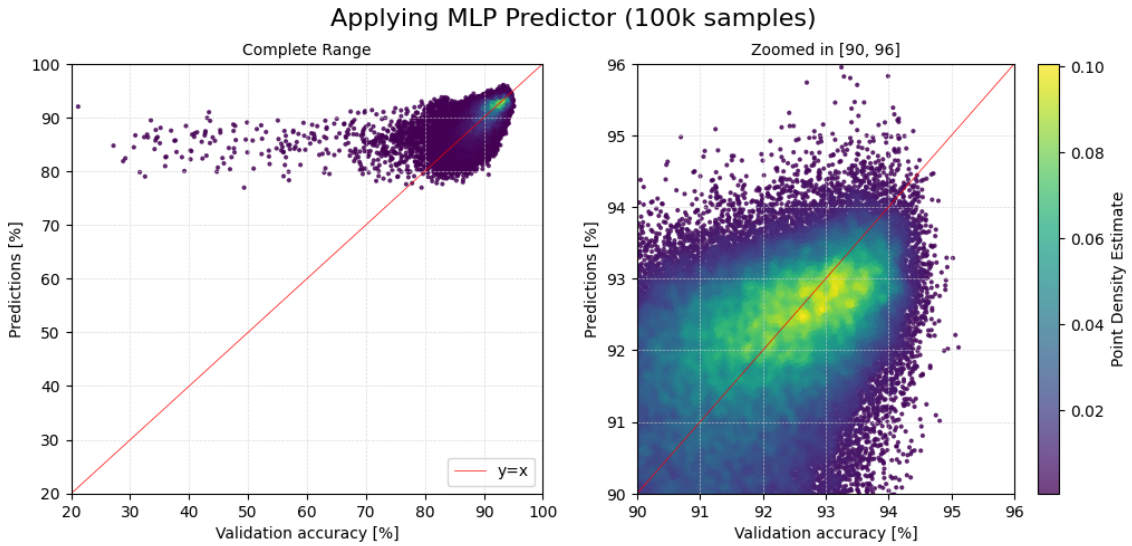


Figure 7.5: Predicting validation accuracies on a set of 100k CNN architectures with the MLP-based predictor.

Results The MLP-based predictor (Figure 7.5) was anticipated to yield the best results in the initial set of experiments. However, as shown in Table 7.2, its performance is comparable to that of the Random Forest and XGBoost predictors. Yet, the initialization and query times are higher, as outlined in Table 7.1.

Summary The first set of experiments successfully demonstrated the potential of different types of machine learning models for predicting the performance of various CNN architectures in the NAS-Bench-101 dataset. Each model was tested on the same dataset and with the same preprocessing steps, allowing for a fair comparison.

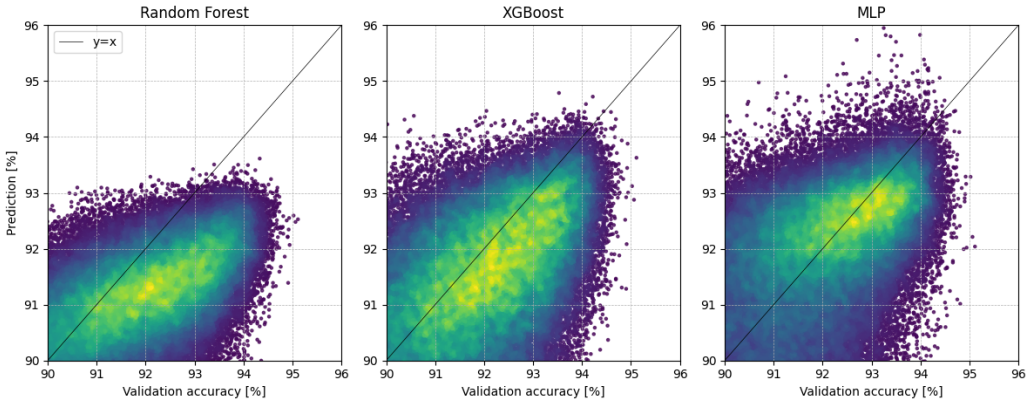


Figure 7.6: Comparison of 3 model-based predictors, trained on 172 samples from NAS-Bench-101 dataset (Point Density Estimate color bars are not shown for better clarity).

Figure 7.6 visually presents the performance comparison of the models estimating the validation accuracy of 100 000 randomly selected architectures from the NAS-Bench-101 dataset. The Figure excludes the Linear Regression model due to its inability to provide accurate and reliable predictions within the realistic accuracy range.
Overall, in terms of accuracy prediction, the XGBoost model seems to be the most reliable. However, the significantly longer training time for the MLP predictor should not overshadow its promising performance metrics. Depending on the specific use case and the available resources, one might opt for the MLP model, keeping in mind the trade-off between time efficiency and prediction accuracy.

It is important to note that these results were obtained using a basic feature set (adjacency matrix and operations). It is possible that a more advanced or custom feature set could lead to different outcomes, which is addressed in the second set of experiments (Section 7.3).

7.3 Experiment Set 2: Extended Feature Set

This set of experiments extends the first (detailed in Section 7.2) by incorporating additional features into the same four models.

Objective The purpose of this experiment set is to evaluate the performance of the selected accuracy predictors when utilizing an *extended* feature set, which includes information beyond the adjacency matrix and operations used in the previous set of experiments, in Section 7.2.

Features In this experiment set, the input vector for the ML models, akin to the previous set of experiments (Section 7.2), is expanded with two additional features: the number of trainable parameters and the count of 3x3 convolution operations in the CNN. Selected for their strong correlation, these features are standardized using the `StandardScaler` from scikit-learn before being concatenated with the one-hot encoded operations and adjacency matrix, resulting in an extended 86-element input vector for each accuracy predictor.

Results As this set of experiments employed the same ML models as in the previous section, the results are visually similar and can be found in Appendix A.

Training and Evaluation Times As can be seen from Table 7.1, the inclusion of extended features did not substantially affect the training times of the models. The training and evaluation times for the Linear Regression (LR) and the Random Forest (RF) models remained constant, while the XGBoost models experienced a slight increase in evaluation time. The MLP model's training time slightly decreased, despite the larger feature set, which could be due to stochastic variations in the training process.

Performance Metrics The results of the extended feature set experiments show notable improvements in predictive performance for RF, XGBoost, and MLP models, while the performance of the LR model deteriorated significantly. The LR model's performance declined considerably when employing the extended feature set, as evidenced by its MSE and MAE increasing to 2.94×10^{16} and 2.37×10^{6}, respectively. It's also worth noting that the LR model had a negative Pearson correlation coefficient and R^2 value, indicating a very

poor fit to the data. On the other hand, RF, XGBoost, and MLP models exhibited improved performance when utilizing the extended feature set. RF demonstrated the highest R^2 value of 0.4215, a marked improvement from 0.3394 when using the standard feature set. The MSE and MAE of the RF model also reduced, indicating better prediction accuracy with the extended features. Similar trends were observed for XGBoost and MLP, where MSE and MAE reduced while Pearson correlation, Kendall's τ, and R^2 values increased.

Summary These results suggest that the inclusion of additional features, specifically the number of trainable parameters and the count of 3x3 convolution operations, enhanced the performance of RF, XGBoost, and MLP models. However, the LR model's performance significantly deteriorated with these added features. The results underscore the importance of feature selection and its impact on the performance of different ML models.

7.4 Experiment Set 3: Graph Convolutional Networks

This experiment set utilizes a different approach to both Experiment Set 1 (Section 7.2) and Experiment Set 2 (Section 7.3). It employs the power of Graph Neural Networks, specifically GCNs, to take into account the graph structure information itself.

7.4.1 GCN Predictor

Setup The `GCNPredictor` class sets up a GCN-based model for accuracy prediction defined within the `NeuralPredictorModel` class, which uses `DirectedGraphConvolution` for each GCN layer. The hyperparameters for the GCN model are set by default as follows: the number of hidden units in the GCN layers (`gcn_hidden`) is 144, the size of training batches (`batch_size`) is 10, the learning rate (`lr`) is 1e-4, the weight decay for L2 regularization (`wd`) is 1e-3, and the number of training epochs (`epochs`) is 300. The batch size during model evaluation (`eval_batch_size`) is set to 1000. The target values are normalized using the mean (`self.mean`) and standard deviation (`self.std`) of the training set, and these transformations are reversed when making predictions. This setup was inspired by Google Brain's Neural Predictor setting [62].

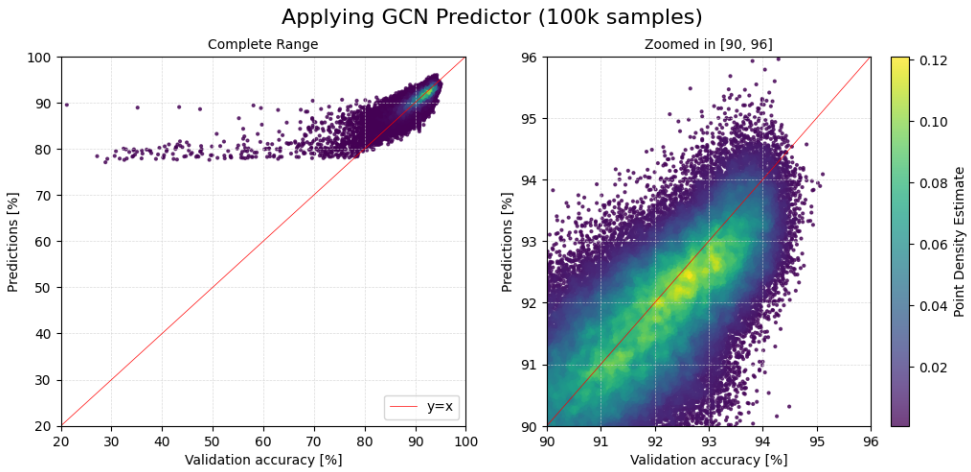


Figure 7.7: Predicting validation accuracies on a set of 100k CNN architectures with the GCN-based predictor.

Result The GCN model was applied to a set of 100 000 CNN architectures, as depicted in Figure 7.7. The results showed that the GCN-based predictor successfully leveraged the topological information present in the graph representations of the CNN architectures. The predictions were quite accurate, demonstrating the potential of GCNs as a tool for predicting the accuracy of different architectures.

Table 7.1: Training and Evaluation Times of Predictors (training set of 172 training samples, evaluating on 100 000 subset of the NAS-Bench-101 dataset)

Predictor	**Feature Set**	**Training Time (s)**	**Evaluation Time (s)**
Linear Regression	Standard	0.01	0.01
Random Forest	Standard	0.10	0.80
XGBoost	Standard	0.34	0.07
MLP	Standard	22.45	1.41
Linear Regression	Extended	0.01	0.02
Random Forest	Extended	0.10	0.77
XGBoost	Extended	0.24	0.07
MLP	Extended	21.16	1.20
GCN	DAG	53.24	3.07

Training and Evaluation Times Table 7.1 shows the training and evaluation times of the GCN-based predictor in comparison with the predictors from Experiment Sets 1 and 2. The GCN model required more time for training, but this was compensated by the improved accuracy of the predictions (Table 7.2). The evaluation time also increased, but it remained acceptable given the complexity of the task and the size of the evaluation set.

Table 7.2: Performance Metrics of All Predictors (KT stands for Kendall's Tau)

Predictor	**Features**	**MSE**	**MAE**	**Pearson**	**KT**	R^2
LR	Standard	753.12	0.39	0.0010	0.4800	-8.8×10^5
RF	Standard	0.0006	0.0140	0.6246	0.4981	0.3394
XGBoost	Standard	0.0006	0.0151	0.5280	0.4560	0.2485
MLP	Standard	0.0006	0.0148	0.6046	0.4843	0.3147
LR	Extended	2.9×10^{16}	2.4×10^6	-0.0003	-0.2325	-3.4×10^{19}
RF	Extended	0.0005	0.0125	0.6683	0.5631	0.4215
XGBoost	Extended	0.0005	0.0126	0.6440	0.5668	0.4107
MLP	Extended	0.0005	0.0130	0.6618	0.5178	0.4327
GCN	DAG	0.0003	0.0097	0.8018	0.6498	0.6276

Performance Metrics The performance metrics for the GCN model, shown in Table 7.2, indicate its superior performance in comparison to the other models, including those with extended features. The GCN model achieved the lowest MSE and MAE and the highest Pearson correlation, Kendall's tau, and R^2 values. This suggests that the GCN model is particularly effective at integrating the architectural information of the CNNs to make accurate predictions.

Summary These results further underline the efficacy of the GCN model as a prediction tool. Despite the increased computational requirements, the GCN model outperformed all the other models in terms of prediction accuracy. This suggests that incorporating architectural information into the prediction process can significantly enhance the accuracy of the results.

7.5 Efficiency Gain Analysis of Accuracy Predictors in NAS

In the context of performance estimation in NAS, the potential speed-up achieved in training time can be quantified by utilizing accuracy predictors. This speed-up arises from the reduced number of architectures that need to be fully trained, a saving made possible by the predictors.

As described in Subsection 5.4.1, the most computationally expensive aspect of using accuracy predictors is the training of architectures: N architectures for predictor building and K architectures representing the most promising candidates selected by the predictor. In this study, N is 172 (inspired by Google Brain's Neural Predictor [62]), which corresponds to the number of CNN architectures used for predictor training. For this analysis, let's assume $K = 100$ (arbitrary choice, influenced by Google Brain's Neural Predictor [62]), representing the top-100 architectures selected by the predictor. Additionally, the computational overhead of initializing and querying the predictor must be considered. The GCN predictor will be used for this analysis, as it is the most computationally intensive among the tested models and also showed the most promise. The training time for the GCN predictor is approximately 53 seconds, and the query time for 100 000 CNN samples is just over 3 seconds.

Utilizing the NAS-Bench-101 dataset, which provides the training time for each individual CNN architecture, the total time saved when using the predictor can be calculated. The mean training time for the CNNs in the dataset is approximately 1932 seconds. The total training time for the selected 172 CNNs used for predictor training is 350 385 seconds. Therefore, the total time required for the predictor to function with a reasonable degree of accuracy, denoted as T_{total}, is calculated as the sum of the time taken to train N architectures (T_N), the time taken to train K architectures (T_K), the time required to initialize the predictor (T_{init}), and the time taken to query the predictor (T_{query}). Mathematically, this can be represented as:

$$T_{\text{total}} = T_N + T_K + T_{\text{init}} + T_{\text{query}} = 350\,385 + 193\,200 + 53 + 3 = 543\,641 \text{ seconds} \quad (7.1)$$

In contrast, training all 100 000 CNNs without a predictor would take approximately 193 200 000 seconds (mean CNN training time multiplied by 100 000). The speed-up achieved by using the predictor is therefore:

$$\text{Speed-up} = \frac{T_{\text{without predictor}}}{T_{\text{total}}} = \frac{193\,200\,000}{543\,641} \approx 355 \quad (7.2)$$

This result underlines the significant efficiency gains that can be achieved using accuracy predictors in NAS, demonstrating approximately a 355-fold reduction in computational time. This highlights the crucial role of accuracy predictors in enabling efficient and practical NAS.

Chapter 8

Conclusion

The primary objective of this bachelor thesis was to construct accuracy predictors for CNNs within the NAS framework, leveraging a diverse array of ML models. The scope of this work spanned the examination of existing performance estimation strategies in NAS, the intricate processes of feature selection and extraction, the design and implementation of each individual predictor, and the rigorous comparison and analysis of their respective performance.

This thesis began with the foundational concepts of AI and ML (Chapter 2), focusing on the specific ML models utilized for performance estimation in the context of NAS. It then delved into the intricacies of CNNs, describing their architecture and prevalent benchmarks. Chapter 4 introduced the concept of NAS, outlining key elements such as search spaces, search strategies, and performance estimation strategies. The design process of the accuracy predictors was detailed in Chapter 5, followed by an in-depth description of each predictor's implementation. All the scripts and source code used in this project are publicly available on GitHub[1].

The experimental results (Chapter 7) demonstrated the power and efficacy of ML models as accuracy predictors. The GCN model, in particular, showed superior performance in predicting the accuracy of CNN architectures. The GCN model achieved the lowest MSE and MAE, and the highest Pearson correlation, Kendall's Tau, and R^2 values. Importantly, the analysis revealed a significant efficiency gain with the use of accuracy predictors, demonstrating approximately a 355-fold reduction in computational time.

Looking forward, this work opens up several avenues for future research. An interesting direction could be to extend this work to different search spaces and make it compatible across them. This could potentially enhance the versatility of these predictors and their applicability in various contexts. Exploring different features for the ML models and incorporating more sophisticated methods to preserve additional information about architecture encoding could also further improve the accuracy of predictions. This thesis has successfully demonstrated the potential of ML models as effective and efficient tools for performance estimation in NAS, setting the stage for further exploration and innovation in this field.

Moreover, this work has been recognized for its contribution to reducing energy consumption through the automation of neural network architecture exploration and was awarded at the Excel@FIT 2023 student conference on innovation, technology, and science[2] by the expert panel.

[1] https://github.com/xsmida03/BP-Accuracy-Predictors
[2] https://excel.fit.vutbr.cz/vysledky/

Bibliography

[1] Barry, A., Ochiai, B., Ringley, B., Acerbi, J., Bollini, M. et al. Spot: A Mobile Manipulator for Industrial Applications. *IEEE Robotics & Automation Magazine*. IEEE. 2021, vol. 28, no. 3, p. 28–36.

[2] Beuzen, T. *Scikit-learn GridSearch and Pipelines* [https://www.tomasbeuzen.com/post/scikit-learn-gridsearch-pipelines/]. Accessed: 2023-04-27.

[3] Biggi, G. and Stilgoe, J. Artificial Intelligence in Self-Driving Cars Research and Innovation: A Scientometric and Bibliometric Analysis. 2021. Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3829897.

[4] Breiman, L. Random Forests. *Machine Learning*. october 2001, vol. 45, p. 5–32. DOI: 10.1023/A:1010950718922.

[5] Cai, H., Gan, C., Han, S., Wang, X., Zhang, Y. et al. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In: *Advances in Neural Information Processing Systems*. 2019, p. 9462–9473.

[6] Chen, L.-C., Collins, M. D., Zhu, Y., Papandreou, G., Zoph, B. et al. Searching for Efficient Multi-Scale Architectures for Dense Image Prediction. *ArXiv*. 2018, abs/1809.04184.

[7] Chen, T. and Guestrin, C. XGBoost: A Scalable Tree Boosting System. *CoRR*. 2016, abs/1603.02754. Available at: http://arxiv.org/abs/1603.02754.

[8] Cox, D. D. and Dean, T. L. Neural Networks and Neuroscience-Inspired Computer Vision. *Current Biology*. 2014, vol. 24, p. R921–R929.

[9] Cybenko, G. V. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*. 1989, vol. 2, p. 303–314.

[10] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. et al. Imagenet: A large-scale hierarchical image database. In: Ieee. *2009 IEEE conference on computer vision and pattern recognition*. 2009, p. 248–255.

[11] Dong, X. and Yang, Y. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. *ArXiv*. 2020, abs/2001.00326.

[12] Dumoulin, V. and Visin, F. A guide to convolution arithmetic for deep learning. *ArXiv*. 2016, abs/1603.07285.

[13] Elsken, T., Metzen, J. H. and Hutter, F. Neural Architecture Search. In: Hutter, F., Kotthoff, L. and Vanschoren, J., ed. Springer, 2019, chap. 3, p. 69–86.

[14] Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*. 1980, vol. 36, p. 193–202.

[15] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] Guo, R., Zhao, Z., Wang, T., Liu, G., Zhao, J. et al. Degradation state recognition of piston pump based on ICEEMDAN and XGBoost. *Applied Sciences*. september 2020, vol. 10, p. 6593. DOI: 10.3390/app10186593.

[17] Géron, A. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. Sebastopol, CA: O'Reilly Media, 2019. ISBN 978-1492032649.

[18] Hastie, T., Tibshirani, R. and Friedman, J. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009. Springer series in statistics. ISBN 9780387848846. Available at: https://books.google.cz/books?id=eBSgoAEACAAJ.

[19] Haykin, S. *Neural Networks and Learning Machines*. 3rd ed. Pearson, 2009. ISBN 9780131471399.

[20] He, K., Zhang, X., Ren, S. and Sun, J. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2014, vol. 37, p. 1904–1916.

[21] He, K., Zhang, X., Ren, S. and Sun, J. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, p. 770–778.

[22] Huang, G., Liu, Z. and Weinberger, K. Q. Densely Connected Convolutional Networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, p. 2261–2269.

[23] Hubel, D. H. and Wiesel, T. N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology*. 1962, vol. 160.

[24] Hutter, F., Kotthoff, L. and Vanschoren, J., ed. *Automatic Machine Learning: Methods, Systems, Challenges*. Springer, 2019.

[25] Kandasamy, K., Schneider, J. and Póczos, B. Neural Architecture Search with Bayesian Optimisation and Optimal Transport. In: *Advances in Neural Information Processing Systems*. 2018, p. 2016–2025.

[26] Kang, J.-S., Kang, J., Kim, J.-J., Jeon, K.-W., Chung, H.-J. et al. Neural Architecture Search Survey: A Computer Vision Perspective. *Sensors*. 2023, vol. 23, no. 3. DOI: 10.3390/s23031713. ISSN 1424-8220. Available at: https://www.mdpi.com/1424-8220/23/3/1713.

[27] Kendall, M. G. A New Measure of Rank Correlation. *Biometrika*. Oxford University Press. 1938, vol. 30, 1/2, p. 81–93.

[28] Khan, M. Y., Qayoom, A., Nizami, M., Siddiqui, M. S., Wasi, S. et al. Automated Prediction of Good Dictionary EXamples (GDEX): A Comprehensive Experiment with Distant Supervision, Machine Learning, and Word Embedding-Based Deep Learning Techniques. *Complexity*. september 2021. DOI: 10.1155/2021/2553199.

[29] Kingma, D. P. and Ba, J. Adam: A Method for Stochastic Optimization. *CoRR*. 2014, abs/1412.6980.

[30] Krizhevsky, A. Learning multiple layers of features from tiny images. *Citeseer*. 2009.

[31] Krizhevsky, A., Nair, V. and Hinton, G. CIFAR-10 (Canadian Institute for Advanced Research). Available at: http://www.cs.toronto.edu/~kriz/cifar.html.

[32] Krizhevsky, A., Nair, V. and Hinton, G. CIFAR-100 (Canadian Institute for Advanced Research). Available at: http://www.cs.toronto.edu/~kriz/cifar.html.

[33] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira, F., Burges, C., Bottou, L. and Weinberger, K., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012, vol. 25. Available at: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[34] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE*. 1998, vol. 86, p. 2278–2324.

[35] Li, L. and Talwalkar, A. Random search and reproducibility for neural architecture search. *ArXiv preprint arXiv:1902.07638*. 2019.

[36] Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L.-J. et al. Progressive Neural Architecture Search. In: *European Conference on Computer Vision*. 2017.

[37] Liu, Y., Wang, X., Li, Z. and Li, L. Difference between cross-correlation and convolution on convolutional neural networks. *ArXiv preprint arXiv:1803.07294*. 2018.

[38] L'Heureux, A., Grolinger, K., Elyamany, H. F. and Capretz, M. A. M. Machine Learning With Big Data: Challenges and Approaches. *IEEE Access*. 2017, vol. 5, p. 7776–7797. DOI: 10.1109/ACCESS.2017.2696365.

[39] Mitchell, T. *Machine Learning*. McGraw-Hill, 1997. McGraw-Hill International Editions. ISBN 9780071154673. Available at: https://books.google.cz/books?id=EoYBngEACAAJ.

[40] Montavon, G., Samek, W. and Müller, K. Methods for Interpreting and Understanding Deep Neural Networks. *CoRR*. 2017, abs/1706.07979. Available at: http://arxiv.org/abs/1706.07979.

[41] Montgomery, D. C., Peck, E. A. and Vining, G. G. *Introduction to Linear Regression Analysis*. John Wiley & Sons, 2012.

[42] OpenAI. *GPT-4 Technical Report*. 2023.

[43] Pham, H., Guan, M. Y., Zoph, B., Le, Q. V. and Dean, J. Efficient Neural Architecture Search via Parameter Sharing. In: *International Conference on Machine Learning*. 2018.

[44] Popescu, M.-C., Balas, V., Perescu Popescu, L. and Mastorakis, N. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*. july 2009, vol. 8.

[45] Real, E., Aggarwal, A., Huang, Y. and Le, Q. V. Aging Evolution for Image Classifier Architecture Search. In: *AAAI Conference on Artificial Intelligence*. 2019.

[46] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*. 1958, 65 6, p. 386–408.

[47] Ruchte, M., Zela, A., Siems, J., Grabocka, J. and Hutter, F. *NASLib: A Modular and Flexible Neural Architecture Search Library* [https://github.com/automl/NASLib]. GitHub, 2020.

[48] Ruder, S. An overview of gradient descent optimization algorithms. *ArXiv*. 2016, abs/1609.04747.

[49] Rumelhart, D. E., Hinton, G. E. and Williams, R. J. Learning representations by back-propagating errors. *Nature*. Nature Publishing Group. 1986, vol. 323, no. 6088, p. 533–536.

[50] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S. et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*. 2014, vol. 115, p. 211–252.

[51] Scherer, D., Müller, A. C. and Behnke, S. Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. In: *International Conference on Artificial Neural Networks*. 2010.

[52] Shi, X., Zhou, P., Chen, W. and Xie, L. Darts-Conformer: Towards Efficient Gradient-Based Neural Architecture Search For End-to-End ASR. *ArXiv preprint arXiv:2104.02868*. 2021.

[53] Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*. 2014, abs/1409.1556.

[54] Springenberg, J. T., Dosovitskiy, A., Brox, T. and Riedmiller, M. A. Striving for Simplicity: The All Convolutional Net. *CoRR*. 2014, abs/1412.6806.

[55] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 2014, vol. 15, p. 1929–1958.

[56] Sun, C., Shrivastava, A., Singh, S. and Gupta, A. K. Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, p. 843–852.

[57] Sun, Y., Yen, G. G., Zhang, M. and Tan, K. C. *Evolutionary deep neural architecture search: Fundamentals, methods, and recent advances*. Springer, 2023.

[58] Sze, V., Chen, Y. hsin, Yang, T.-J. and Emer, J. S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*. 2017, vol. 105, p. 2295–2329.

[59] Szegedy, C., Ioffe, S., Vanhoucke, V. and Alemi, A. A. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *ArXiv*. 2016, abs/1602.07261.

[60] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E. et al. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2014, p. 1–9.

[61] Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio', P. et al. Graph Attention Networks. *ArXiv*. 2017, abs/1710.10903.

[62] Wen, W., Liu, H., Li, H. H., Chen, Y., Bender, G. et al. Neural Predictor for Neural Architecture Search. *ArXiv*. 2019, abs/1912.00848.

[63] White, C., Neiswanger, W., Nolen, S. and Savani, Y. *A Study on Encodings for Neural Architecture Search*. 2021.

[64] White, C., Neiswanger, W., Savani, Y., Schneider, J. and Poczos, B. BANANAS: Bayesian optimization with neural architectures for neural architecture search. *ArXiv preprint arXiv:1910.11858*. 2020.

[65] White, C., Zela, A., Ru, B., Liu, Y. and Hutter, F. How Powerful are Performance Predictors in Neural Architecture Search? *ArXiv*. 2021, abs/2104.01177.

[66] Willmott, C. J. and Matsuura, K. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate research*. Inter-Research. 2005, vol. 30, no. 1, p. 79–82.

[67] Ying, C., Klein, A., Christiansen, E., Real, E., Murphy, K. et al. NAS-Bench-101: Towards Reproducible Neural Architecture Search. In: Chaudhuri, K. and Salakhutdinov, R., ed. *Proceedings of the 36th International Conference on Machine Learning*. Long Beach, California, USA: PMLR, 09–15 Jun 2019, vol. 97, p. 7105–7114. Proceedings of Machine Learning Research. Available at: http://proceedings.mlr.press/v97/ying19a.html.

[68] Zoph, B. and Le, Q. V. Neural Architecture Search with Reinforcement Learning. *ArXiv*. 2016, abs/1611.01578.

[69] Zoph, B., Vasudevan, V., Shlens, J. and Le, Q. Learning Transferable Architectures for Scalable Image Recognition. In:. June 2018, p. 8697–8710. DOI: 10.1109/CVPR.2018.00907.

Appendix A

Experiment Set 2 visualizations

In this set of experiments, every ML model was trained on 172 samples of architectures from the NAS-Bench-101 dataset. With the exception of the linear regression predictor, all other models (including random forest, xgboost, and MLP) demonstrated improved performance in accuracy prediction (compared to the Experiment Set 1, described in Section 7.2).

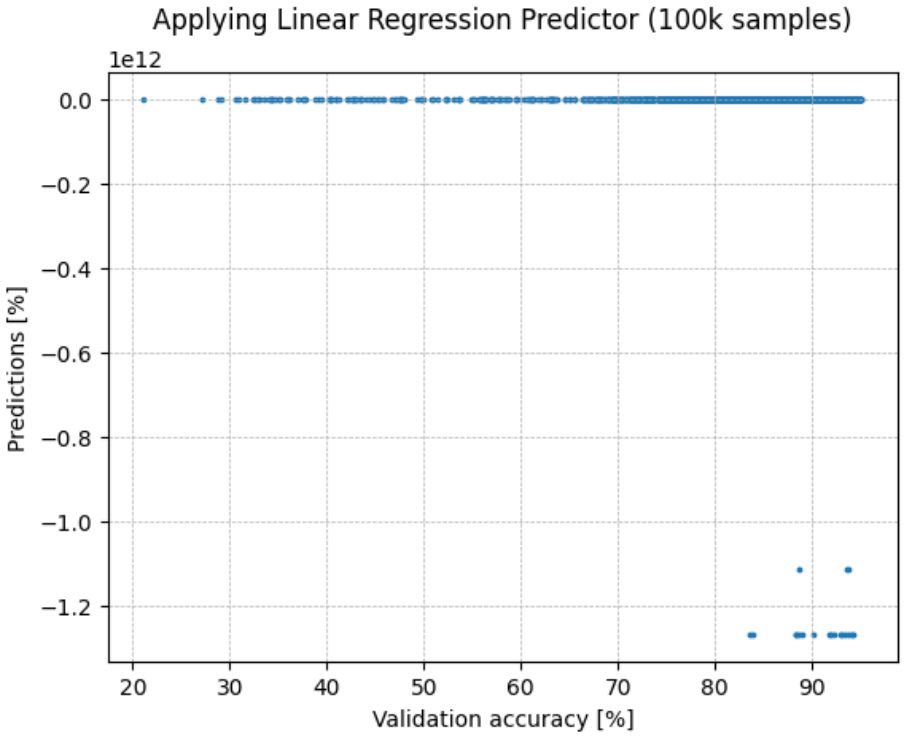


Figure A.1: Predicting validation accuracies on a set of 100k CNN architectures with the Linear-Regression-based predictor. Utilizing the extended feature set.

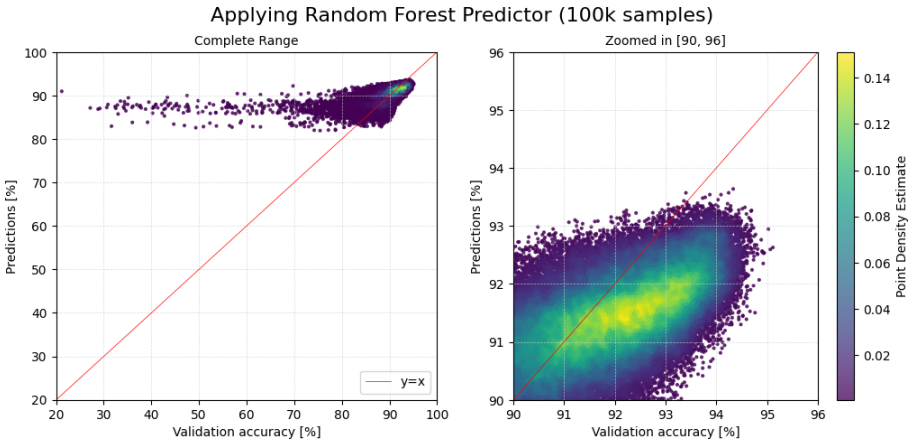


Figure A.2: Predicting validation accuracies on a set of 100k CNN architectures with the Random-Forest-based predictor. Utilizing the extended feature set.

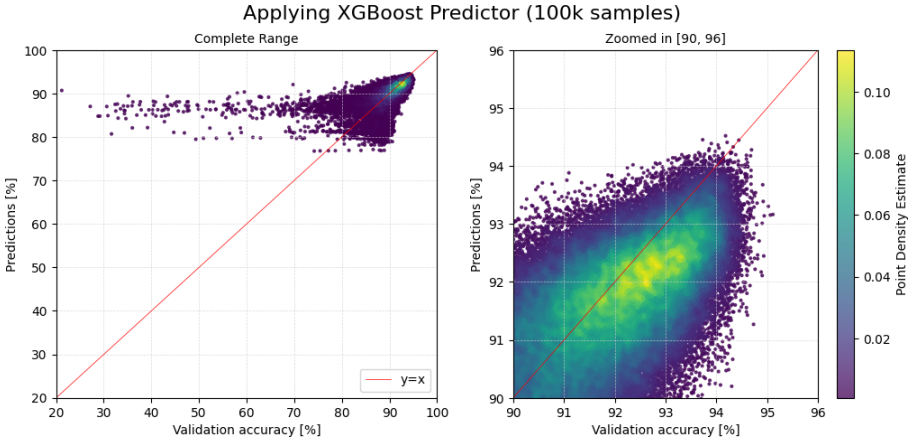


Figure A.3: Predicting validation accuracies on a set of 100k CNN architectures with the XGBoost-based predictor. Utilizing the extended feature set.

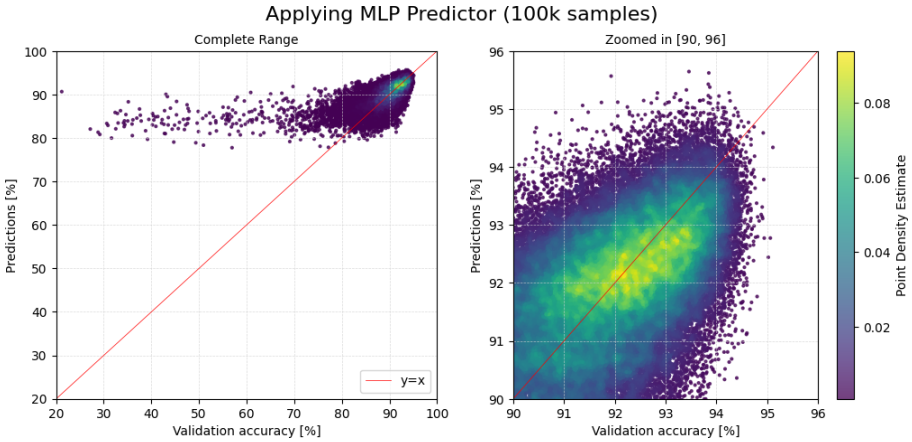


Figure A.4: Predicting validation accuracies on a set of 100k CNN architectures with the MLP-based predictor. Utilizing the extended feature set.