Lehrstuhl für Data Science

Comparing different search strategies in Neural Architecture Search (NAS)

Masterarbeit von

Muhammad Arsal Munir

1. Prüfer	2. Prüfer
Prof. Dr. Michael Granitzer	Prof. Dr. Harald Kosch

February 5, 2022

Contents

Abstract

In recent years, Machine Learning (ML) has seen a huge spike in the use of its models in various application areas such as image recognition, machine translation, and many more. Almost all of these models have multiple layers that are computationally expensive and memory intensive. Not only this, but it also takes tremendous manual efforts to come up with models with optimized parameters that give the best results for a given task. Automated machine learning helps in automating the process of applying ML algorithms to real-world problems. Neural Architecture Search (NAS), also considered a subdomain of AutoML, helps automate designing neural networks. NAS-Bench-101 is one such public dataset of architectures for NAS research purposes. We enhanced the strategy of NAS-Bench-101; in addition to random search and regularised evolution strategy, we also explored genetic algorithm and provided a comparative study to understand the different behavior of each search strategy. Based on our results, the genetic algorithm finds more architectures yielding higher accuracy as compared to regularised evolution strategy, which in turn finds more architectures with accuracy relatively higher than architectures found by random search. In the end, to overcome this rigorous training, we used regression to predict the results of architectures based on their graphs attributes.

Acknowledgments

There are many personalities who helped me during the whole course of my master studies here at the University of Passau; I would like to thank all of them. But definitely, I want to express my gratitude to some people who had a very impactful role in my studies and where I am right now.

First, I would like to thank my master thesis advisor Mr. Julian Stier, who gave me the opportunity to work under his guidance, and whenever I need any help or if I am stuck anywhere, he is always there to take me back on track. But, mainly, he helped me a lot to expand my knowledge related to research areas of machine learning and data science.

Second, I would like to thank Prof. Dr. Michael Granitzer and Prof. Dr. Harald Kosch for supervising my master thesis. In my initial presentation, Prof. Dr. Granitzer's comments on my research goals provided me with new ideas towards my research improvement and how to make it more crucial.

I like to thank the University of Passau for providing me with resources to accomplish my studies. During my stay in Passau, I met many people across nations and became friends who helped me in every aspect academically and personally.

Last but not least, I would like to thank my parents, and brothers, who have been very supportive and helpful. They made sure that I gave 100% in every situation. Thank you so much for believing in me; this work would not have been possible without their endless support and encouragement.

List of Figures

List of Tables

1 Introduction

In the past few years, Neural Architecture Search (NAS) has already made a significant impact in many applications regarding neural networks. So now researchers are working on its improvement in different aspects. Especially, NAS is extremely slow in computing. It takes around a few months, which is not feasible for many researchers. Another problem is that everyone is working in this field in their own way, so their methods are unique and cannot be comparable, making it is challenging to improve each other's methodologies [Yin+19]. There are many algorithms that are performing the search operation under the different search spaces. After getting these architectures, training has to be done using different approaches, e.g., hyper-parameters, data augmentation, regularization. Each algorithm has its own approach; therefore, it leads towards the comparability problem among each other. For tackling this problem, NAS-Bench-101 has already shown success in overcoming this problem [DY20].

Moreover, Evolution strategies (ES) are the stochastic search algorithms, which use mutation, recombination, and selection applied to any population that consists of candidate solutions and iteratively evolve itself to get the better solutions and eventually evolve itself towards the optimal solution. There are many evolution strategies, but we will discuss more about Regularised Evolution Strategy and Genetic Algorithm and its pros and cons in our work.

Additionally, Figure 1.1 shows a NAS framework. The general framework of Neural Architecture Search (NAS). In which NAS starts with the predefined set of operations, and from that search space, it obtains a large number of network architectures. After getting these architectures set, candidate architecture is trained and ranked. They then obtained new candidate network architectures after adjusting the search strategy by getting that ranking information. When the search is finished, the best network architecture is further checked for the final performance evaluation [Ren+20].

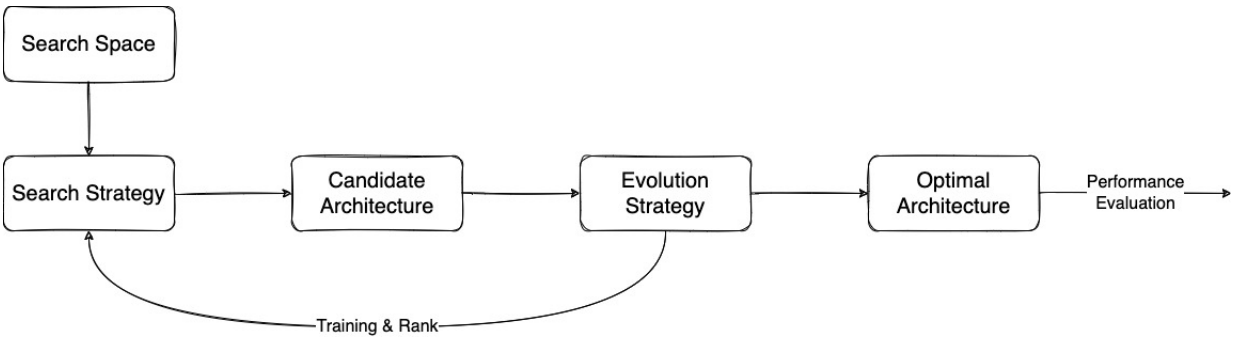

Figure 1.1: Neural Architecture Search (NAS) framework [Ren+20]

.

1.1 Motivation

In Neural Architecture Search (NAS), the main idea is to minimize human interference as much as possible to find better architectures from the given search space and to motivate the use of architecture search algorithms to discover them automatically. Because the design of network architecture is very significant regarding the final performance, therefore, nowadays researchers are working on different strategies to find out more suitable architecture to get optimal results. Similarly, in NAS-Bench-101, they have used random search and regularized search strategy. Therefore, reproducing these results and adding another search strategy such as a genetic algorithm to enhance this approach can be more constructive.

1.2 Research Goals

The primary goal of this master thesis is to examine the effects of different search strategies on NAS, i.e., random search, regularised evolution strategy, and genetic algorithm. Later on, some modifications in the regularized evolution strategy and compare these results with random search and genetic algorithm. Later on, the results of the random search will be stored in the newly created dataset, including the attributes of the graph. Based on these features (attributes), a few regressors is being trained and predict the performance without training it on the dataset (CIFAR-10). We plan to achieve this goal by answering the following research questions.

1. **Compare the performance of random search and regularized evolution strategy on NAS. Which strategy has the better results?**

   The first part of the research would be to reproduce the results of NAS-Bench 101 [Yin+19] and to extend this approach by adding the genetic algorithm into it. We will tackle the task in the following steps:

   - Apply random search and regularised evolution strategy on the search space of NAS. Train and test our approach on the dataset, e.g. CIFAR-10, and reproduce the results as NAS-Bench-101.
   - Additionally, we will apply one more evolution strategy (ES) genetic algorithm to enhance this approach to better compare.
   - After performing these experiments, we will have the results of random search, genetic algorithm, and regularised evolution strategy for the comparative study.

2. **Can the performance of NAS be estimated based on graph attributes without training the architectures?**

   In the second phase of our research, we enlarge the idea of a comparative study by adding some performance metrics, and we will be focused on getting results without having rigorous training by using several steps, which are followings:

   - First, We will create a dataset in which attributes of the graphs will be stored, such as the number of nodes, edges, labels, density, etc.

- After creating the database, it will be divided into the train and test sets. Then few regressors are trained on that training set and predict the performance of the remaining unseen data, test data.
- Then, we report an R-squared value showing how these data fit each regressor model.

1.3 Structure of the Thesis

The thesis contains a total of seven chapters. Chapter 1 introduces the topic and shows some motivation regarding why we are working on this topic? For example, what are the research goals of reproducing the results of NAS-Bench-101 and extending the approach by introducing our methods?

Chapter 2 contains the basic knowledge about neural networks and their related terminologies. Then enhance it towards Neural Architecture Search, many search strategies, their pros and cons, databases, and different performance metrics. We explain the related work in detail in Chapter 3. Chapter 4 briefly demonstrate the different methodologies and experiments to fulfill our research goals. Moreover, implementation of the search space, search strategies, and evaluation procedures.

Furthermore, after performing these detailed experiments, we have many meaningful results that we explain in Chapter 5. After getting these handy results, we discuss its possible outcomes in Chapter 6. Lastly, we conclude our thesis work and briefly discuss the future work in Chapter 7.

2 Technical Background

Deep learning and neural networks, more appropriately known as an 'artificial' neural network (ANN), are significant areas in artificial intelligence and the technology industry, inspired by the biological neural networks that naturally perform most of the functions in the human brain. They are currently providing the optimal solutions to real-life problems in speech recognition, image recognition, and natural language processing (NLP). For example, artificial intelligence (AI) that can learn to paint has been included in several articles [Ji+21], create a user interface (pix2code), build 3D Models, some create images by using a sentence description. Day by day, there are many more unbelievable things being done in these domains [HOT06].

The definition of a neural network by the inventor of one of the neurocomputers of its first kind, Dr. Robert Hecht-Nielsen, is:

"*...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.*"

Furthermore, there is another way we can think about the ANN as a computational system that works more like natural neural networks, which is by default fitted in our human brain.

2.1 Inspiration from human brain

A neuron is the fundamental computing unit of the human brain. Human nervous system consists of 86 billion neurons approximately [Bar18] and they are connected by each other with almost $10^{14} - 10^{15}$ synapses [MS95]. The brain is responsible for all body functioning and human intelligence, which perceives things, reacts accordingly and generates the output. On the other hand, the basic unit of the artificial neural

network is also a neuron, but more often known as a node or unit. These nodes are interconnected with each other to receive inputs and act accordingly.

An activation function – that we will discuss further in this section – is also inspired by the biological changes in our brain, where different stimuli activate different neurons fire. For example, if we feel happy, certain neurons will be fired, and if we smell bad or feel bad, some other neurons will be activated. So we can say that if we feel good, then the value of the neurons will be 1. In contrast, if we feel bad, then the value would be 0. Therefore, we can map our happiness or sadness on a scale of 0 to 1 using a specific activation function. It is not always the case that the activation function will transform the value from 0 to 1. It depends on which activation function we are using for which problem. So it can be lie between -1 to 1 while using any other activation function.

2.2 Perceptron

The perceptron is a mathematical representation of the biological neuron and an algorithm for supervised learning of binary classifiers. The algorithm allows neurons to learn and interpret data in the training set one at a time. Moreover, it accepts numerical inputs to communicate within the network. Now the question arises, what is the difference between a perceptron and a neural network? Primarily, the perceptron is a single-layer neural network without any hidden layers developed by F. Rosenblatt [Ros58], inspired by [MP56], a linear classifier. It helps in the classification of the input data in supervised learning. Suppose we include a hidden layer in it. In that case, it will become Multilayer Perceptron (MLP), and it is a supplement of a feed-forward neural network that generates a set of outputs through its hidden layers from a set of inputs.

As shown in the figure 2.1, Single Layer Perceptron (SLP) consists of five major steps: inputs, weights, weighted sum, activation function, and output. And it goes through with the following steps.

1. Each input, x, will be multiplied by its corresponding weights w.
2. After assigning weights, every input goes through the weighted sum, which is just adding all the weighted inputs.

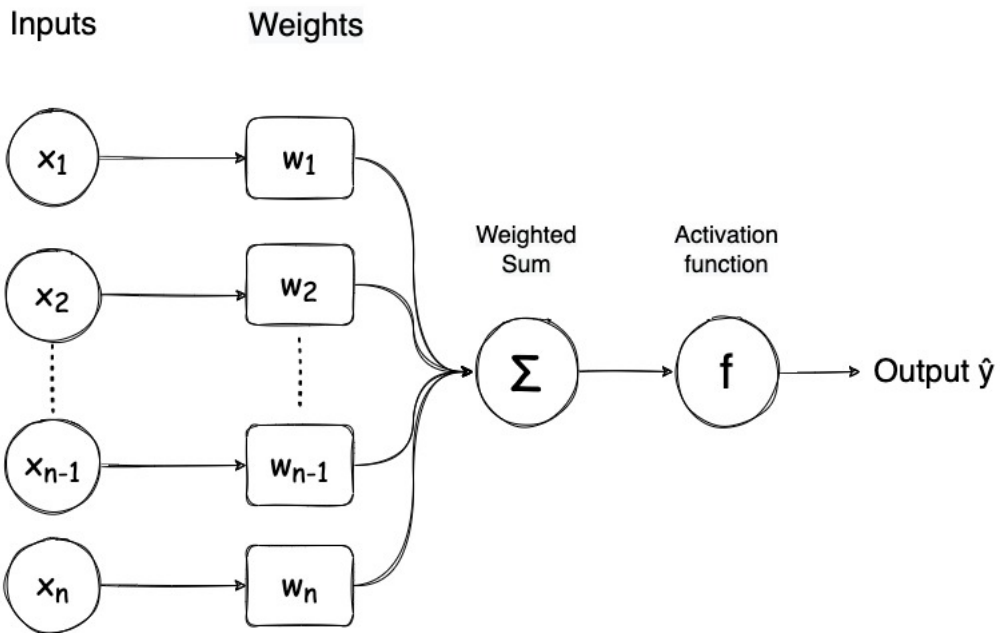

Figure 2.1: Single Layer Perceptron (SLP) with its binary inputs, $x_1, x_2, ..., x_n$, weights $w_1, w_2, ..., w_n$, and output $\hat{y}$

.

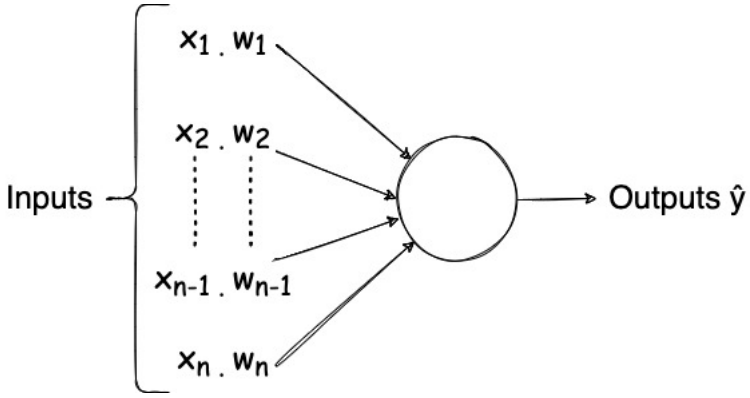

Figure 2.2: Multiplying inputs with there corresponding weights

.

$$h(X) = \sum_{i=1}^{n} w_i.x_i \tag{2.1}$$

3. Then, implement the activation function (e.g. ReLU, Sigmoid) on this weighted sum.

4. Lastly, transfer it to the output layer.

2.2.1 Activation Functions

An activation function is a very important part of a perceptron. It transforms the node's input values and ensures that the resulting output lies between (0, 1) or (-1, 1). The most popular activation functions are Sigmoid, Rectified Linear Unit (ReLU), and Hyperbolic Tangent (Tanh). These are all nonlinear activation functions and are commonly used where it is impossible to get the desired output using a linear function.

Rectified Linear Unit (ReLU)

Rectified Linear Unit (ReLU) is very popular non-linear activation function.

$$y = max(x, 0) \tag{2.2}$$

According to the equation, the value of x will return if and only if it is greater than 0 unless it will return 0. In simple words, ReLU is used as a filter where only positive values ($x > 0$) can pass through to the following layers of the neural network. Following figure 2.2 shows the line plot of the ReLU activation functions.

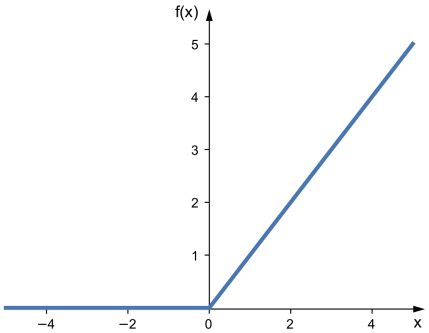

Figure 2.3: Rectified Linear Unit (ReLU)

.

The important thing is ReLU function can be used anywhere in the neural network but not in the final layer. Unlike sigmoid, it eliminates the need to use complex exponential functions as its positive input's derivative always remains 1.

Exponential Linear Unit (ELU)

The Exponential Linear Unit (ELU) is the improved version of ReLU, as it accommodates the smooth negative values as well. When $x < 0$, it will return the value different from 0 (which is not the case in ReLU), and it will always be closer to 0. Indeed, as x drops, ELU saturates and reaches a negative value. This saturation indicates that ELU has a low derivative, which reduces the fluctuation of the result and, as a result, the amount of information conveyed to the next layer.

$$\begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases} \tag{2.3}$$

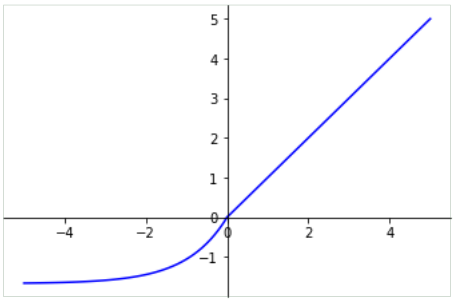

Figure 2.4: Exponential Linear Unit (ELU)

.

Sigmoid

The sigmoid function gives the output value in the range of 0 and 1. So if we talk about any classification problem such as songs reviews, if the out is closer to 1 after applying the sigmoid activation function, we can consider it positive feedback. In contrast, if the output is near 0, it will be negative feedback. In the figure 2.5, the line plot shows its trend.

$$Sigmoid,\ S(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

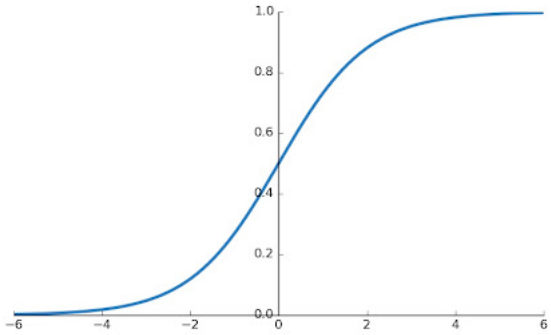

Figure 2.5: Sigmoid Activation Function

.

Hyperbolic Tangent (Tanh)

Hyperbolic Tangent (Tanh) is, in fact, a mathematically shifted version of the Sigmoid activation function. Additionally, Sigmoid gives the result between 0 and 1, but Tanh squashes the result between -1 and 1. If we take our previous example about rating the songs after applying Tanh, if the output would be near to 1, then it will consider as positive feedback. On the other hand, if the result is near -1, it is definitely a negative review of that specific song.

$$Tanh,\ tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

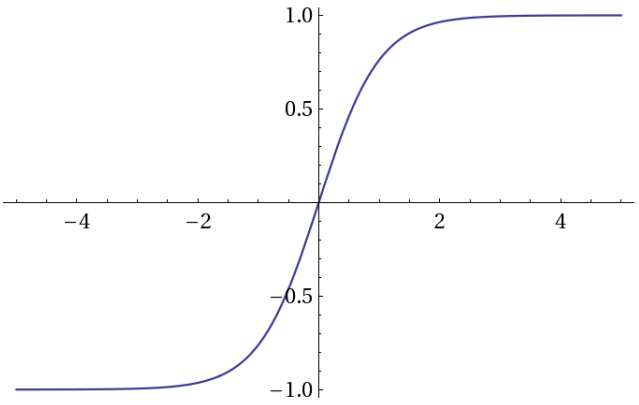

Figure 2.6: Hyperbolic Tangent (Tanh) [Men18]

.

Customized activation functions

Sometimes, predefined activation functions are not enough to tackle every problem. In particular, we may need to create our own activation functions if we cannot get the desired results or if we want some specific transformation to our data. To create an activation function, we have to consider the following points:

- An activation function must be non-linear. Different from the $f(x) = ax+b$, which cannot be represented by the straight line.
- An activation function takes tensor as an input. So if we want to use exponential value, then we cannot use the library math.exp(x) because here x is a real number. But we must use tensorflow.match.exp(x), where x is a tensor.

2.3 Neural Networks

A neural network is a class of computing systems. Layers of neurons make up neural networks, and these neurons comprise the neural network's main processing units that we have already discussed earlier. A neural network basically consists of three different parts. It starts from the input layer, then some hidden layers, and at the end output layer. The input layer accepts the data, whereas the output layer predicts the final outcome. Between them are the hidden layers, which execute the majority of the network's computations.

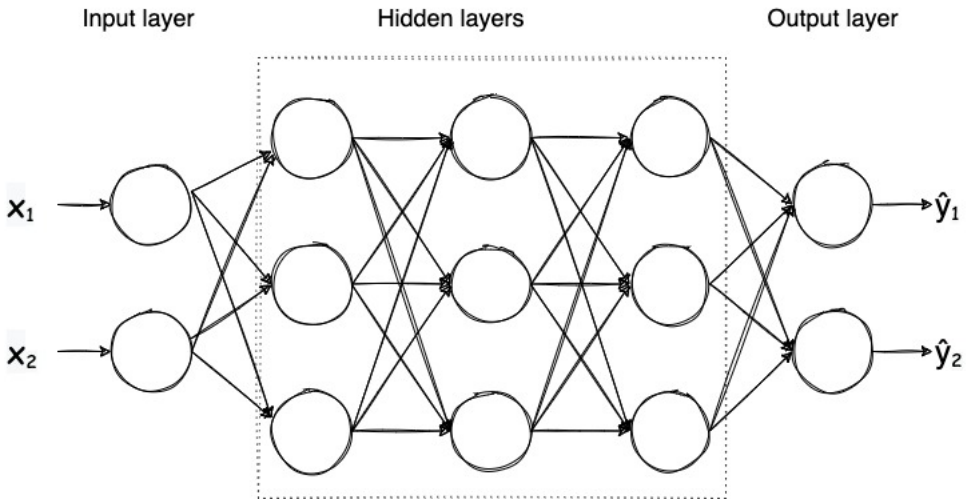

Figure 2.7: An Artificial Neural Network (ANN) with an input layer, three hidden layers, and an output layer, fully connected.

2.3.1 Types of Neural Network

Before discussing more the terminologies related to neural networks, we are going to step back and take a broader look at the neural networks, some of the types of neural networks that exist, and look at a few of them, and the kind of problems that they are used to solve. In figure 2.8, we have almost a complete list of neural networks compiled by Fjodor van Veen [1]. The reason why this image is so useful is that it does not just show the general architectures, it has a nice chart for what the different neurons in the neural network charts are doing. Now we have kind of dealt with just the basics of neural networks, and the purpose is to not go deep into all of these complicated types of neural networks, but to discuss a few of the most popular neural networks and put the figure just to show the wide range of neural networks that exist that scholars who specialize in machine learning have figured out how to create to solve very specific problems.

Feed Forward Neural Network

In figure 2.7, a feed-forward Artificial Neural Network (ANN) is shown, which takes two inputs x_1 and x_2 in the input layer. And the same number of outputs in the output layer.

[1] `https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464`

Each neuron is directly connected to all the neurons of the very next layer. In between input and output, hidden layers can be seen, consisting of a bunch of perceptrons and processing the inputs with the weighted-sum and sending the processed data towards the output layer.

The input layer accepts the data from the outside world, and it transfers the data to the hidden layers. And output layer receives the process data from hidden layers. Moreover, the side of the output layer depends on the number of outcomes we need. Let's take an example in the shape image recognition problem; if we want to know whether the input image is a square, triangle or a circle, then the output layer should consist of three neurons of each shape.

Recurrent Neural Network (RNN)

RNNs are very useful for audio-based problems, so they are really good at sequencing audio information. The reason why is because their work is different from the other neural network. Now, if we notice in the figure 2.8 that this one got a recurrent cell and what the recurrent neural network is able to do is instead of actually just going from the beginning to the output, it is able to iterate information into a neuron. So it can store this information and store it in the memory and go back to it to edit and later in that iteration, or it takes a hidden layer and send information backward and re-allows it to go back to that specific data again after some calculations are done in this layer. So this means, in RNN order of the input is very important; for example, if we have to feed in "milk" and then "cookies" may have different results instead of providing in "cookies" and then "milk".

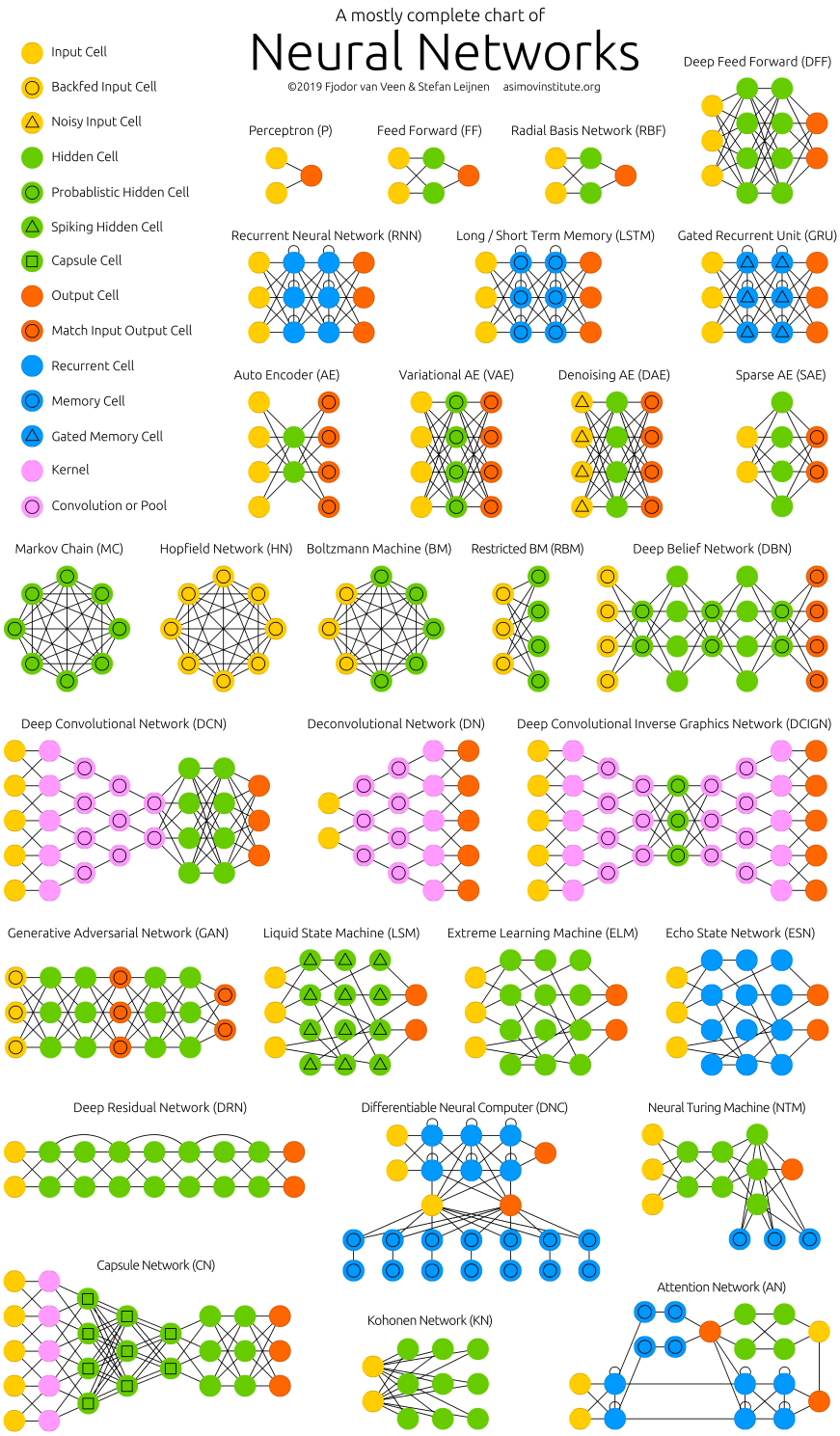

Figure 2.8: This figure visualizes various types of neural networks, from a simple perceptron to more complex attention networks.

Convolutional Neural Network (CNN)

CNNs are useful for solving image-based problems, and convolutional layers do differently from a typical dense layer in Keras or TensorFlow is going to be the way in which they iterate across the data. So in a normal neural network, the neuron will act upon the input data of a certain size, but on the other hand, a convolutional layer will iterate across an image in a predetermined matrix of information or matrix of pixels. And that allows it to be really good at identifying and extracting different features of an image. It allows for the neural network to kind of look at an image piece by piece, break it down, and each neuron or node is able to kind of become familiar with and be able to activate at a higher level depending on if that feature is found. Our research work is mostly revolving around the convolutional neural network.

2.3.2 Batch Gradient Decent

A neural network has fixed inputs and outputs. Therefore, we can only tweak the weights applied to the hidden layers to improve the output. An error function E After getting the results from the output layer, we can calculate the error in our output by using the equation 2.6:

$$E(w) = \frac{1}{2}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2 \tag{2.6}$$

Weights are then updated using this error function as:

$$w \coloneqq w - \eta\nabla E(\mathrm{w}) \tag{2.7}$$

where learning rate is η and $\nabla E(\mathrm{w})$ is partial derivative of the cost function, computed for each weight in the weight vector as:

$$\nabla E(\mathrm{w}) = \frac{\partial E(\mathrm{w})}{\partial w_j} \tag{2.8}$$

By substituting the value of $E(\mathrm{w})$ from equation 2.6 in above equation, we can derive $\nabla E(\mathrm{w})$, as shown by [Ras15] and [HOT06], as:

$$\begin{aligned}
\nabla E(\mathrm{w}) &= \frac{\partial}{\partial w_j}\frac{1}{2}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2 \\
&= \frac{1}{2}\sum_{i=1}^{n}\frac{\partial}{\partial w_j}(\hat{y}^{(i)} - y^{(i)})^2 \\
&= \frac{1}{2}\sum_{i=1}^{n}2(\hat{y}^{(i)} - y^{(i)})\frac{\partial}{\partial w_j}(\hat{y}^{(i)} - y^{(i)}) \\
&= \sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})\frac{\partial}{\partial w_j}(\hat{y}^{(i)} - \sum_{j} w_j.x_j^{(i)}) \\
&= \sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})(-x_j^{(i)})
\end{aligned} \tag{2.9}$$

By substituting the value from equations 2.9 in 2.7, we can re-write the weight update formula as:

$$w \coloneqq w + \eta\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})(x_j^{(i)}) \tag{2.10}$$

This whole process of adjusting errors and improving results is called Batch Gradient Decent. To achieve convergence, these training and weights update phases must be repeated.

2.3.3 Backpropagation

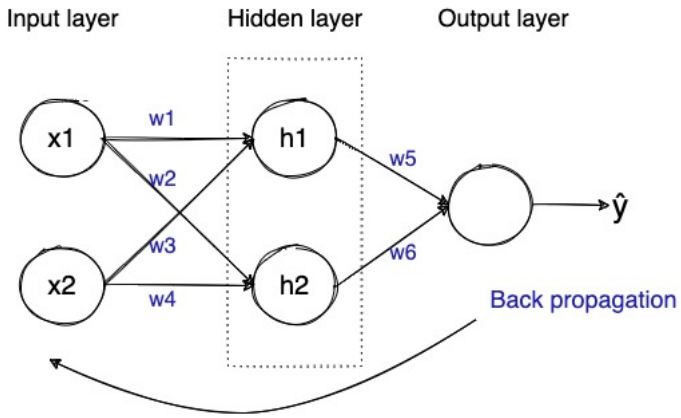

Figure 2.9: Backpropagation in a simple neural network with two inputs x_1 and x_2, and an prediction output $\hat{y}$

.

Backpropagation is an essential operation when we have to minimize the error using the chain rule. Given the error function, the backpropagation process starts from the output layer and goes backward within the hidden layers to the input layer by adjusting the weights.

The figure 2.9 shows a simple neural network with two neurons in the input and output layer and a hidden layer. Error estimation at the output layer is basically the difference between the actual result and the predicted result. So, this difference can be minimized when we backpropagate and adjust the weights in the hidden layers. After adjusting weights, the feedforward operation takes place, and the error function is applied after getting the results. This process runs continuously till the predicted result comes closer to the actual result.

Let's break it down the whole process. When we have the results at the output layer, we then start our backpropagation, thereby traversing the whole neural network backward and repeating this procedure multiple times to improve the results. Consider the following figure of the output layer of the neural network 2.9.

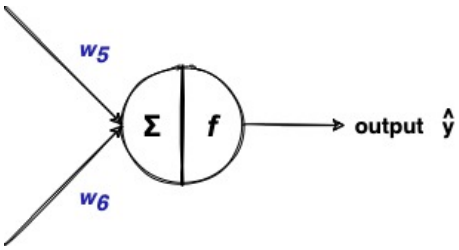

Figure 2.10: Backpropagating error from output layer to modify weights w_5 and w_6.

We must first compute $\frac{\partial E}{\partial w_5}$ and $\frac{\partial E}{\partial w_6}$ before updating w_5 and w_6. The discrepancy between the goal y and the projected output $\hat{y}$ is called error. Furthermore, applying the weighted sum to an activation function yields the predicted output $\hat{y}$. The weighted sum is determined as follows:

$$weighted\ sum\ (z) = \hat{y}_{h1}w_5 + \hat{y}_{h2}w_6 \tag{2.11}$$

where $\hat{y}_{h1}$ is output of the hidden neuron h_1 and $\hat{y}_{h2}$ is output of the hidden neuron h_2.

As a result, the partial derivative of E with respect to w_5 and w_6 computed using the following formula:

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_5} \tag{2.12}$$

$$\frac{\partial E}{\partial w_6} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_6} \tag{2.13}$$

To compute the partial derivative of error E with regard to w_1, w_2, w_3, and w_1, we use the similar procedure.

Now move further backward towards the hidden neuron h_1 which is showed in the following figure 2.11:

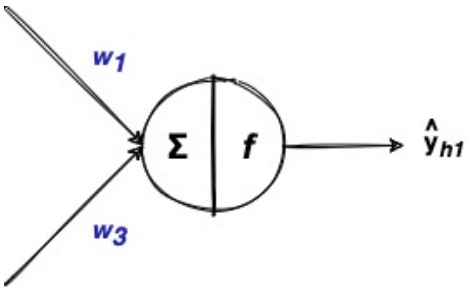

Figure 2.11: Backpropagating error from h_1 to modify weights w_1 and w_3.

Again, to update w_1 and w_3, we first compute $\frac{\partial E}{\partial w_1}$ and $\frac{\partial E}{\partial w_3}$ as:

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial \hat{y}_{h1}} \frac{\partial \hat{y}_{h1}}{\partial z_{h1}} \frac{\partial z_{h1}}{\partial w_1} \tag{2.14}$$

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial \hat{y}_{h1}} \frac{\partial \hat{y}_{h1}}{\partial z_{h1}} \frac{\partial z_{h1}}{\partial w_3} \tag{2.15}$$

Here, $\hat{y}_{h1}$ is output of the hidden neuron h_1 and z_{h1} is the weighted sum used to compute $\hat{y}_{h1}$, calculated as:

$$z_{h1} = i_1 w_1 + i_2 w_3 \tag{2.16}$$

Lastly, take a look at the image below, which depicts the neural network's hidden neuron h_2:

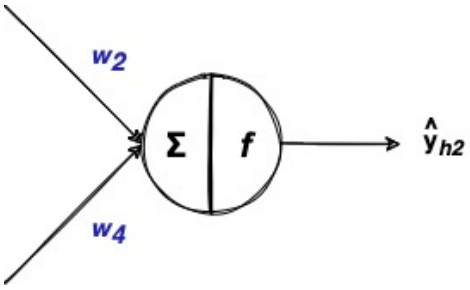

Figure 2.12: Backpropagating error from h_2 to modify weights w_2 and w_4.

To update w_2 and w_4, we first compute $\frac{\partial E}{\partial w_2}$ and $\frac{\partial E}{\partial w_2}$ as:

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial \hat{y}_{h2}} \frac{\partial \hat{y}_{h2}}{\partial z_{h2}} \frac{\partial z_{h2}}{\partial w_2} \tag{2.17}$$

$$\frac{\partial E}{\partial w_4} = \frac{\partial E}{\partial \hat{y}_{h2}} \frac{\partial \hat{y}_{h2}}{\partial z_{h2}} \frac{\partial z_{h2}}{\partial w_4} \tag{2.18}$$

Here, $\hat{y}_{h2}$ is output of the hidden neuron h_2 and z_{h2} is the weighted sum used to compute $\hat{y}_{h2}$, calculated as:

$$z_{h2} = i_1 w_2 + i_2 w_4 \tag{2.19}$$

By using the following equation 2.20, update each weight and then train the neural network with updated weights once we get the partial derivatives of error E with respect to weights. This process is repetitive till the neural network converges.

$$w_j \coloneqq w_j - \eta \frac{\partial E}{\partial w_j} \tag{2.20}$$

A typical neural network has a complicated topology since each layer is tightly coupled. Furthermore, information in such a neural network goes only in one way, i.e., there are no feedback loops where the input to a function is also dependent on the output. There are, however, alternative types of neural networks that are either less computationally complicated (for example, Convolutional Neural Networks) or support feedback loops (i.e., Recurrent Neural Networks).

2.4 Automated Machine Learning (AutoML)

Before discussing automation in artificial intelligence and neural architecture search (NAS), we need to understand deep learning and how it is different from machine learning and artificial intelligence.

2.4.1 Deep Learning

Deep learning is a technique used to empower computers with the ability of intelligence like humans. Deep Learning is part of machine learning which rigorously trains the machines to learn and apply the capability of intelligence. Artificial Intelligence is used to teach computers to learn from examples and enables them to think and make decisions like a human brain. Deep Learning requires a lot of training data and excessive computational resources. The basic architecture of Deep Neural Networks used in the Deep Learning mechanism is inspired by the human brain. A Neural Network generally has two or three hidden layers, whereas, Deep Neural Networks contain hundreds of hidden layers. The word deep in the name refers to the depth of hidden layers of the network. The major difference between Machine Learning and Deep Learning is the way features are used. In Machine Learning, we have to identify the features manually, and use them to train models which can differentiate the characteristics of different inputs. For example, a computer can differentiate between animals and birds based on the visual features of feathers, legs, and wings. On the other hand, in the Deep Learning method, we do not explicitly need to tell the machine which features to consider. Deep Learning enables the machine to recognize and differentiate by learning the patterns and extracting features without human involvement. That is the reason why Deep Learning requires a huge amount of data to be able to precisely and deeply learn to create meaningful features out of it. Additionally, if we ask three students to write down the digit '9', notably, they all will not write it down identically. It is easier for a human brain to detect the differences in handwriting, but for a computer to be able to do the same, we need Deep Learning.

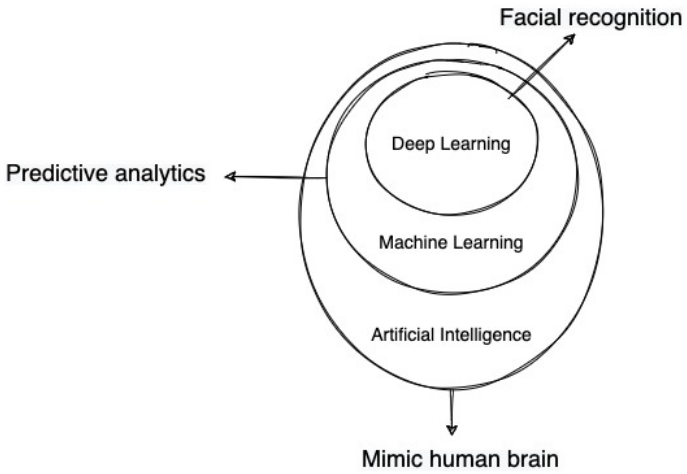

Figure 2.13: Relation between Deep Learning, Machine Learning, and Artificial Intelligence

Scope

Deep Learning is applied in various fields in real life. Such as:

- Chat Bots are prepared using Deep Learning techniques which are extensively used in customer support for automated help and guide.
- In the automotive industry, deep learning is used for object detection and provide facility such as autonomous driving. Such systems have already specialized in differentiating the relevant objects from the surroundings to avoid accidents and provide safety. Apple, Tesla, Nissan, and many more companies are now working on self-driving cars.
- Deep Learning is used in Aerospace industry to detect areas of interest, safe/unsafe zones.

Limitations

Deep learning has a wide scope in its applications, but it also has some limitations. A few of them are as follows:

- The first limitation as discussed earlier is the need for a huge amount of **data**. Although Deep Learning is a very coherent way of dealing with disorganized data, a neural network however requires an enormous quantity of training data.
- To process this massive data is not within every machine's capability, which leads towards the second constraint, which is **computation power**. Training a neural network is quite expensive as it requires Graphical Processing Units (GPUs), which have thousands of cores compared to CPUs.
- Another major limitation of Deep Learning is **training time**. Although there has been a lot of advancements in developing efficient hardware, high-performance GPUs with parallel architecture which is very well suited for Deep Neural Networks and makes Deep Learning possible, it still requires a lot of time for training Deep Neural Networks. The time complexity depends on the size of data and the number of layers in the network, and it can take from a few hours to months to train a Deep Neural Network.

2.4.2 Neural Architecture Search (NAS)

We have to design a neural network manually in deep learning, but we did not discuss how to do it automatically. Neural Architecture Search (NAS) is what the media advertises as "AI that creates AI." Although it sounds very cool and sometimes scary as a concept, and in the last couple of decades, it has become very popular among significant researchers [ZL16], [Kit90], [SM02]. But An automatic NAS has its limitations, which is why it has not become mainstream yet. Nevertheless, we will eventually get to a point where we do not have to design neural networks anymore. By now, NAS has already outperformed manually designed neural networks on some problems [EMH19], especially in image classification [Rea+19], [Zop+18], semantic segmentation [Nek+19], object detection [Zop+18]. According to Hutter, NAS can be seen as a sub-domain of Automated Machine Learning (AutoML) [HKV19].

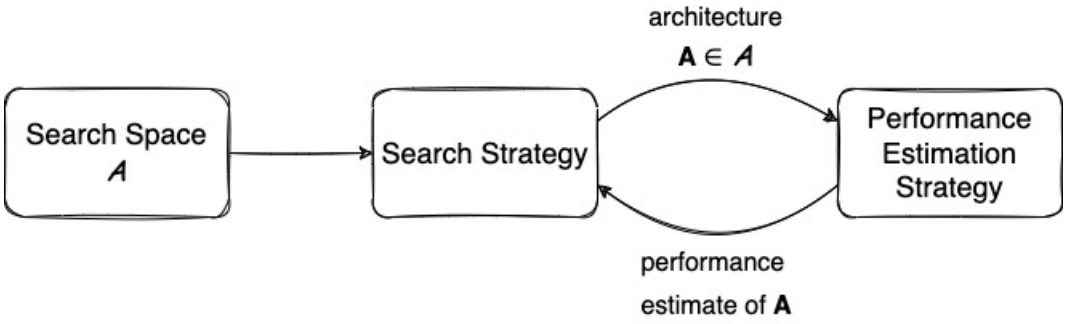

Figure 2.14: Abstract explanation of Neural Architecture Search (NAS) methods [EMH19].

Search Space

All the possible architectures are represented in a specific search space. However, one big challenge about the automated NAS is that there are infinitely many ways one can design a model, so we still have to define a search space to narrow down our search manually, and the choice of the search space also needs careful tuning in many cases. Therefore, it introduces the human bias towards the problem, which may cause the hurdle to find an optimal architecture and possibly to miss that solution because of the limited human knowledge.

Search Strategy

When the search space has been created, we have to define a strategy to traverse the whole search space to find the best architecture for the specific problem. This exploration through a search algorithm can sometimes be more expensive in time and computational power. For example, a search space may consist of possible cell configurations in a ten-layer architecture, and a naïve search algorithm would be a random search. More sophisticated methods use controller models to decide what direction to explore next. Some common themes in architecture search include using reinforcement learning or evolutionary algorithms to propose networks, for example, proposing a large parent model and then searching within its sub-networks or defining a macro architecture and optimizing pieces of it.

In a broad sense, an architecture search algorithm tries to eliminate the knobs in a system by automatically tuning them. However, the current approaches in the architecture search are still now hyperparameter-free by introducing new hyperparameters while eliminating some. For example, the models that generate architectures still need to be hand-designed somehow. Furthermore, hyperparameters like learning rate, regularization strength, and the optimizer type are mostly left outside the scope of the search.

Performance Estimation Strategy

Another problem with the automated model design is defining and measuring the success of a model. Many network architecture research papers optimize the validation error on the different datasets. The most common dataset among them is CIFAR-10. Now the question is, does lower error on the validation set always mean a better model? Especially when we try so many different model configurations, a lower error could be due to pure chance. Some of the research evaluated how their automatically discovered architectures transfer to other datasets, such as ImageNet. Still, it seems that many methods might be leaking information from the validation set by optimizing the network architecture to minimize validation error. This problem is not specifically an automated architecture search problem. However, we can have the same problem with hand-designed model architectures as well when we do too many experiments.

2.5 Types of Search Strategies

After defining a search space, the next task is how to explore this vast search space. Suppose we traverse it one by one, then it will be computationally costly and almost impossible to do that because search space can be an indefinite size. Therefore, we must choose a search strategy to get the optimal output with fewer computations.

There are many search strategies, but we will discuss a few of them that we will be using in our experiments.

2.5.1 Random Search (RS)

Random search (RS) is a class of numerical optimization methods that do not require improving the problem's gradient, allowing it to be applied to functions that are neither continuous nor differentiable. These types of optimization methods are referred to as direct-search, derivative-free, and black-box optimization methods.

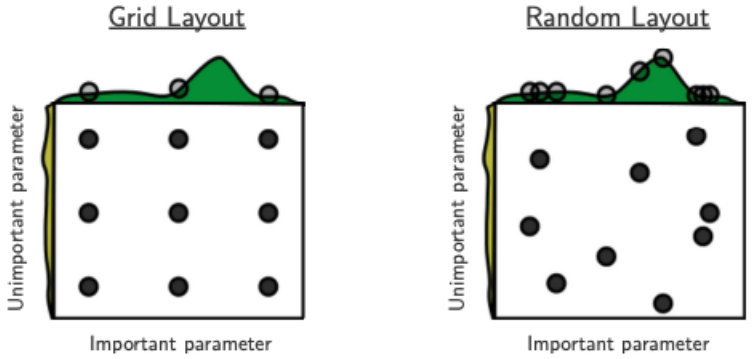

Figure 2.15: Grid search and Random search comparison [BB12].

Grid search is a widely used search strategy for hyper-parameter optimization. Now the question arises, then why is random search beneficial? How does it find the maxima? Leon Bottou [BB12] made our lives easier and explained briefly how random search showed better results than grid search. A comparison with much earlier research that employed grid search and manual search to create neural networks and deep belief networks provides empirical data. He discovered that random search over the same domain could find models as good as or better than neural networks created by a pure grid search in a quarter of the time. Furthermore, when given the same processing budget, random search finds better models by successfully searching a larger, less appealing configuration space.

As shown in the figure 2.15, nine trials of grid search only test in three different locations. With random search, each of the nine trials looks at different values. In high-dimensional hyper-parameter optimization, grid search failure is the rule rather than the exception.

2.5.2 Evolution Search (ES)

This optimization approach is inspired by nature, in which the aim is to take the given architecture and improve it further to make it a better fit for the solution. We search for the optimal solution to the problem. This is inspired by Mendel's modern genetics and Darwin's Theory of Evolution. The major ideas from the theory of evolution are applicable to the Evolution Search Strategy in the context of artificial intelligence. Like the evolution process in the biological species, we choose the artificial evolution process that produces better and optimal solutions. To eliminate any exaggeration, the evolutionary computations provide the estimation of optimal solutions to the complex problems in reality. Such an evolutionary process isn't the supreme tool catering for every problem in existence as it is generally thought of. [Pie20].

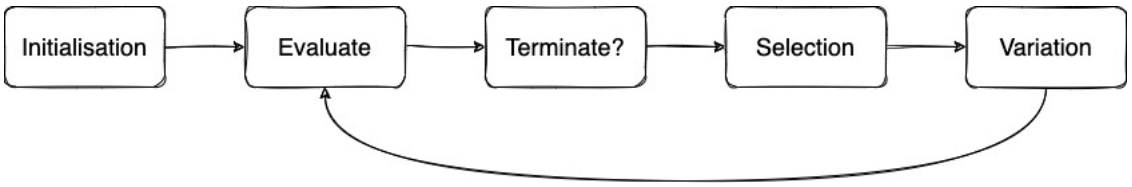

Figure 2.16: Work flow of Evolution Strategy

- In the **initialisation** phase, search space has been defined, where we create our first population out of nowhere.
- In the **evaluation** phase, We calculate the fitness of each of our architectures. Fitness means how well our architecture performs on a certain problem. Higher the fitness value, the better the architecture. And it depends completely on the problem that we are tackling.
- In the **termination** phase, we need to figure out when to stop looping so that the process does not continue forever. We have to define some termination criteria, such as a certain number of generations or until we have our desired result.
- In the **selection** phase, above specific fitness value architectures, survive. According to previous evaluations, we must pick the best samples and eliminate the bad ones.
- In the **variation** phase, also called a mutation, after selecting the better architectures, we consider them as a parent population and use them to produce the child

architectures that we can refer to as the new population for the next generation. [Pie20]

In the bunch of evolutionary algorithms, we will discuss a couple of them that we will use further in our experiments.

Regularised Evolution Strategy

An evolutionary algorithm mutates existing architectures and kills those that do not perform well. Over time, better and better model architectures evolve. In regularised evolution, the aging component is added. So the only way the architecture will be survived is when it is sampled and reproduced; otherwise, it will be eliminated. In other words, after randomly sampled architectures, they trained and evaluated. Higher fitness architectures (based on their performance metrics) will be sampled and passed on to mutate and create child architectures. Then again, we trained and evaluated these child architectures. The oldest architecture in the population that has not been sampled for a long time will be removed from the population. So the main idea here is Aging. AmoebaNet [Rea+19] uses the same search space as NASNet. But, again, It is very computationally expensive and uses a lot of Tensor Processing Units (TPUs).

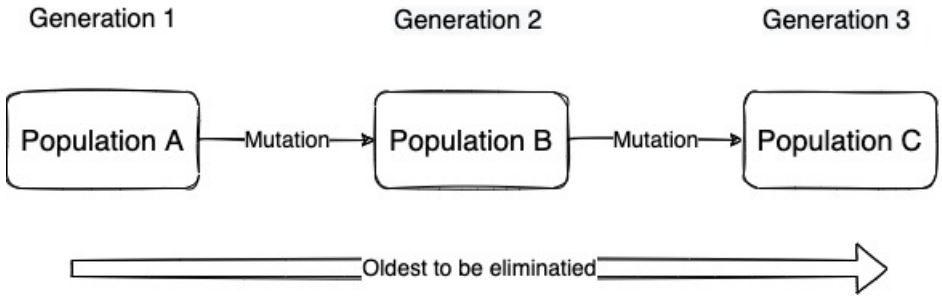

Figure 2.17: Regularised Evolution Strategy

Genetic Algorithm

Genetic algorithms are optimization techniques used to solve non-differentiable or non-linear optimization problems. They imitate the mechanism of biological evolution. Genetic Algorithms initialize by selecting a generation of possible solution candidates tested

against an objective function. Afterwards, we produce subsequent generations of points from the first generation through three phases **Selection**, **Crossover**, and **Mutation**.

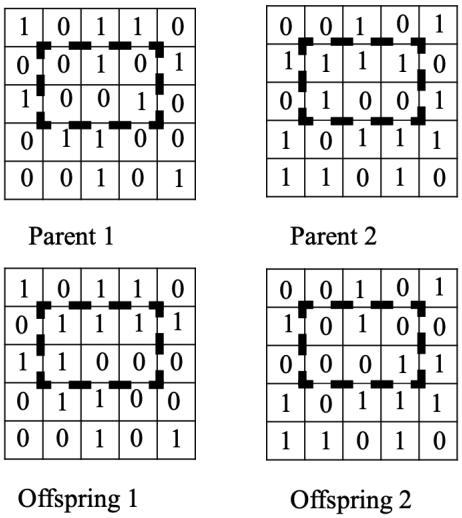

Figure 2.18: An example of matrix crossover
[WMS96]

Selection: Selection means retaining the best-performing parent from one generation to the next. So we look at the figure 2.18, where we can see two matrices, parent one and parent two, from the preceding generation, and they serve as the values of the variables used in the given optimization problem. Since they performed well in the previous generation, they are selected to pass on to the next generation.

Crossover: After the selection phase, these selected matrices will also be used for crossover. In the crossover, we define a rule to merge these parent matrices to create a child matrix for the next generation. This merging can be done through various rules, such as common similarities, or just on a random basis [WMS96].

Mutation: The last phase is a mutation, where a parent matrix is mutated by taking random values and creating a child based on the mutation process. Mutation helps the genetic algorithm better explore the search space by avoiding the local minima.

3 Related Work

There are a lot of researchers who have been working on Neural Architecture Search and optimization techniques. Few of them will be discussed in this chapter to summarise that what has been done so far in the field of AutoML.

In February 2018, Esteban Real et al. published a paper [Rea+19] regarding an evolution strategy to search image classification architectures from the given search space. They used a modified tournament selection evolution algorithm with an age property which works in favor of younger genotypes. They tested their approach on AmoebaNet-A and achieved comparable accuracy to current ImageNet models with complex architecture search methods. Additionally, NAS-Bench-201 [DY20] used this similar approach on CIFAR-10, CIFAR-100, and ImageNet datasets and achieved test accuracy of 93.92, 71.84, and 45.54 respectively. In contrast, we have also used regularized evolution search strategy but with some modifications. After each generation, we eliminate a certain number of architectures that performed really bad in that particular generation. We used two selection techniques during the selection phase; In the first technique, we picked half of the best population and sent it to the mutation phase. Alternatively, in the second approach, we picked the best architecture of a particular generation and mutated it multiple times to create many child architectures, calculate their fitness, and compare them with the best half from the previous generation. So we can say that these two techniques are the mixtures of regularised evolutionary search and the generic evolutionary search. The considerable thing here is that it also favors the younger generation who are good at fitness.

In May 2019, a group of researchers published a paper claiming the first architecture dataset and benchmark for Neural Architecture Search, NAS-Bench-101 [Yin+19]. In which, they evaluated the whole NAS search space by comprehensive analysis as a whole. They thoroughly explained different methodologies regarding the architecture search. They trained and evaluated a large number of Convolutional Neural Networks (CNNs)

on the CIFAR-10 dataset. For architecture search, they used different search strategies such as Random Search and Bayesian optimization. One of the goals of our research was to reproduce results on CIFAR-10 by using random search and regularised evolution strategy. However, to enhance this approach, we also used some flavors of the evolutionary algorithm on CIFAR-10, such as genetic algorithm and some alteration in regularised evolution strategy. In the end, we showed a comparative study of all these search strategies.

In July 2020, Colin White et al. published a paper regarding the encodings for Neural Architecture Search. They defined different approaches of architecture encodings, such as adjacency matrix encodings and path-based encodings including a characterization of the scalability of each encoding. They ran NAS algorithms on architectures encoded using these encoding schemes. After these experiments, these encodings can be differentiated from each other and we can have the best encoding depending on the architectures and algorithms. Both encodings have their pros and cons. In the adjacency matrix encoding scheme, a single architecture can yield multiple adjacency matrices, in contrast, in path-based encoding, multiple architectures can result in the same encoded path. Furthermore, they explained that NAS encodings are quite important because they can directly impact on the overall performance. [Whi+20]. Our research created a search space with thousands of neural networks of the same kind. To represent these neural network architectures, we needed some techniques to apply search strategy. Additionally, many of the significant researches in NAS used adjacency matrix encoding [Yin+19], [ZL16], [LSY18]. We also chose the adjacency matrix encoding in our experiments.

In June 1996, Bradley Wallet et al. published a paper about the matrix representation of the data for genetic algorithm [WMS96]. This is a very old research paper, but it showed some significant results in two-dimensional representation for genetic algorithms. Instead of crossover on the linear structure, they used a two-dimensional structure to apply crossover. Their research used a rectangular patch of the matrices not just for swapping, but they also tackled if they both hold some symmetry. Because if there is some symmetry in the patch chosen for the crossover, it will not impact the results. Additionally, in their experimental design, they randomly created two binary matrices. Moreover, to calculate the fitness of the candidate matrices, they related to R. Axelrod's iterated prisoner dilemma [Axe87]. In the end, they showed a comparison between the performance of the matrix representation method and classical string method. Inspired by their work, we have also used matrix representation for crossover in our experiment

on application of a genetic algorithm for searching the optimal architecture. However, in our crossover approach, we defined some range in rows and columns to disrupt the patch of the matrix. Then, if there is some symmetry, the algorithm will again randomly pick the patch within that specific range and again check for symmetry. In the end, the resultant matrix will be sent to the mutation phase.

Most of the research papers we examined used either evolutionary algorithms or reinforcement learning to search for architectures. Nevertheless, In June 2018, Hanxiao Liu and his fellow researchers published a paper [LSY18] in which they used a different approach and formulated architecture search as a differentiable continuous optimization problem. So to make the search space continuous, they relax the categorical choice of a given operation as a softmax over all possible operations. Once the search is over, the final architecture is discretized by choosing the operations with the highest softmax values. Their approach is reminiscent of sub-network search and fabric-based search algorithms since all possible operations are present during training. Moreover, their search space is similar to NASNet, where they search for cell architectures. DARTS is among the most efficient neural architecture search algorithms, requiring only four GPU days for CIFAR-10 classification.

In August 2018, Thomas Elsken et al. published a research survey [EMH19] regarding the neural architecture search. In which, they provided the complete guide regarding the recent work done in this field. They divided that work into three different dimensions: search space, search strategy, and performance estimation. This literature is very helpful for researcher interested in the Neural Architecture Search domain, and want to get an introduction and understanding of basics of NAS. It helped with starting this research, and provided helpful knowledge and contributed in sparking the interest and inspiration for working in this research area.

Neural Architecture Search (NAS) is currently an in-trend topic among researchers. However, a few of the most recent works include a paper by Andreas Klos et al. on Neural Architecture Search based on genetic algorithm [KRS22], in which they worked on a NAS with high performance and availability. They identified better-performing architectures by exploring nine different configurations in the genetic algorithm. Another remarkable work provides an optimized NAS, namely (GA-NINASWOT), by combining genetic algorithm and noise immunity for neural architecture search without training, published in paper [WLT21] GA-NINASWOT showed better performance than many of the state-of-the-art methods being used.

4 Experimentation

This chapter will explain the methodologies that we used to perform experimentation to achieve research goals, discussed in Section-1.2.

4.1 Overview of the methodology

Our research is divided into two parts. The first part compares the different search strategies to find the best architecture in our search space. In the second part, we have to predict the results of the particular architecture without training it. Figure 4.1 has shown the complete pipeline of experiments and their respective methods.

The first step in the pipeline is choosing the search space. A search strategy is selected to traverse the search space in the second step. After rigorous training of the architectures, we have the final results. Based on the achieved results, we can compare these search strategies. The results of the random search are stored to be used in the second part of the research. The right side of the figure shows the details of the second part of the research. After feature extraction on the results achieved from part-I, and applying regressor, we predict the results of a particular architecture without having to train it again

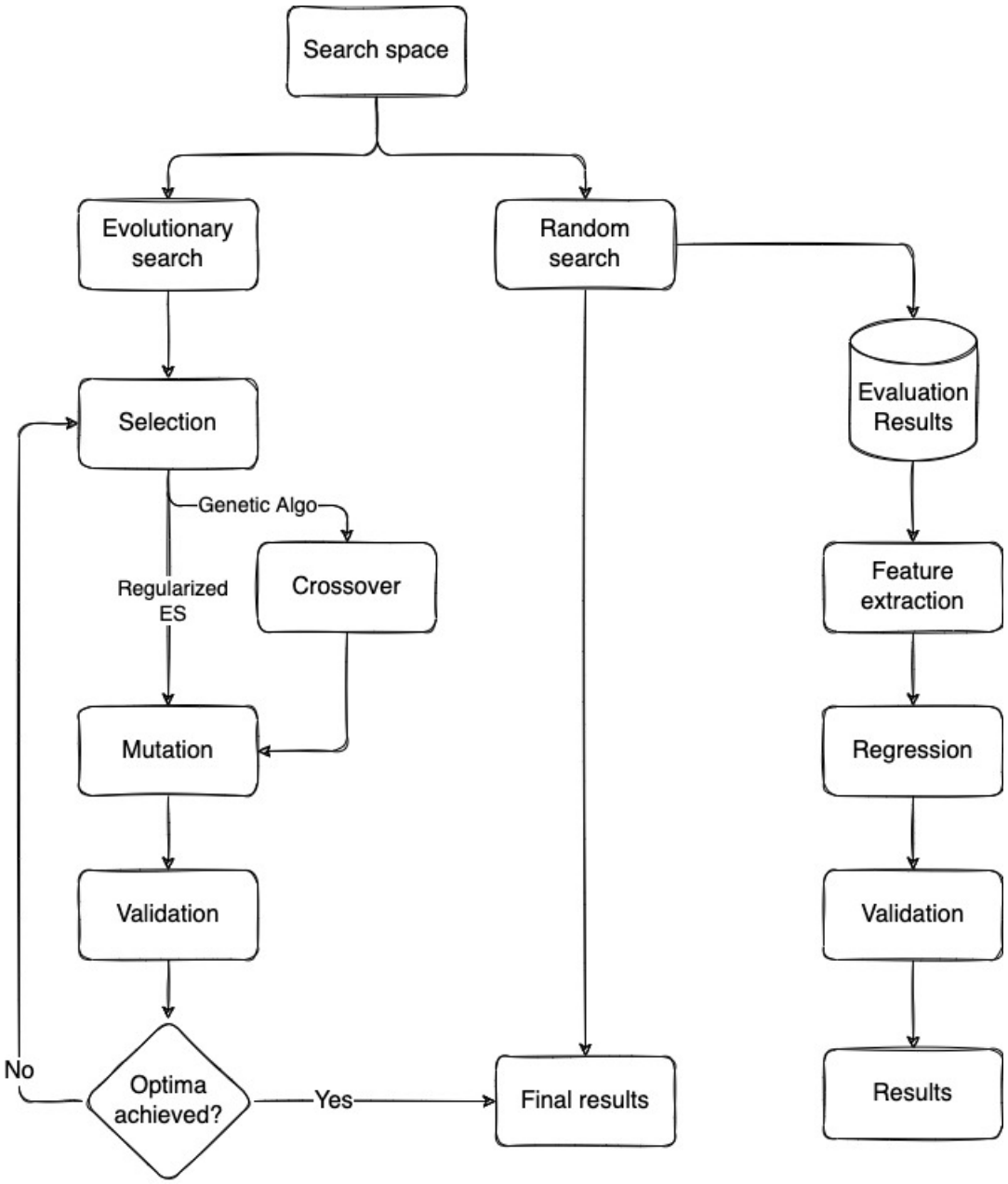

Figure 4.1: This flow chart depicts the methodology of the experiments. We perform random search and two evolutionary search strategies to find optimal architecture from a given search space. We also do performance estimation based on the results from the random search experiment.

4.2 Preprocessing of Experiments

In our research, we have used the CIFAR-10 dataset[1], consisting of 60,000 images of 10 different classes. These images have been divided into train and test images of 50,000 and 10,000, respectively. Most of the NAS researchers have used this dataset because it is not a huge dataset and requires less computation time to train and test [Yin+19]. Additionally, trends have shown that the approaches that perform well on the CIFAR-10 data performed well on the more complex dataset such as ImageNet when scaled up [Zop+18], [KSH12]. Like NAS-Bench-101, we also use the CIFAR-10 dataset to train CNN.

As we previously explained in the Section-2.4.2, the given search space contains all possible architectures. However, because there are endless number of ways to create a model, we must still establish a search space to narrow down our search manually, and the search space needs considerable adjustments. So in the following sub-topics, we will explain the properties of the search space we created for our experiments.

4.2.1 Architectures and Encoding

In our research, we applied the same limitations as [Yin+19] in the structure of the architectures or graphs that we will use further to create a search space. Therefore, we will have a finite number of architectures. Firstly, our space consists of all the directed acyclic graphs, and the maximum number of nodes V in a graph should be 7, and each node has its own Label L representing each operation. And the maximum number of edges should not exceed more than 9. Additionally, the total number of operations L to our nodes that are precisely 3. Which are as follows:

- 1×1 Convolution
- 3×3 Convolution
- 3×3 Max-pool

Apart from that, other labels are IN and OUT, which explicitly represent input and output nodes. After creating these architectures, the next step is its representation.

[1] `https://www.cs.toronto.edu/~kriz/cifar.html`

There are several methods of encoding, and different encoding techniques may impact the results and favor a specific algorithm. We chose a generic encoding approach, i.e., adjacency matrix encoding. We have direct acyclic graphs (DAGs) maximum of 7 nodes and 5 labels (input and output nodes are fixed). Because of DAG, our adjacency matrix will be upper triangular. So, we have 21 maximum possible edges in our encoded matrix and 3 possible operations, so a total number of unique models are: $2^{21} \times 3^5 \approx 510$M. However, most of the matrices are invalid according to our assigned constraints (i.e., graphs cannot have more than 9 edges). Furthermore, there can be unique graphs and their encodings, which we will discuss in the next section. After eliminating all the duplication, a total of 423K possible architectures are left in our search space [Yin+19].

Some examples of simple feed-forwarded structures that we have used for our search space has shown in the figure 4.2.

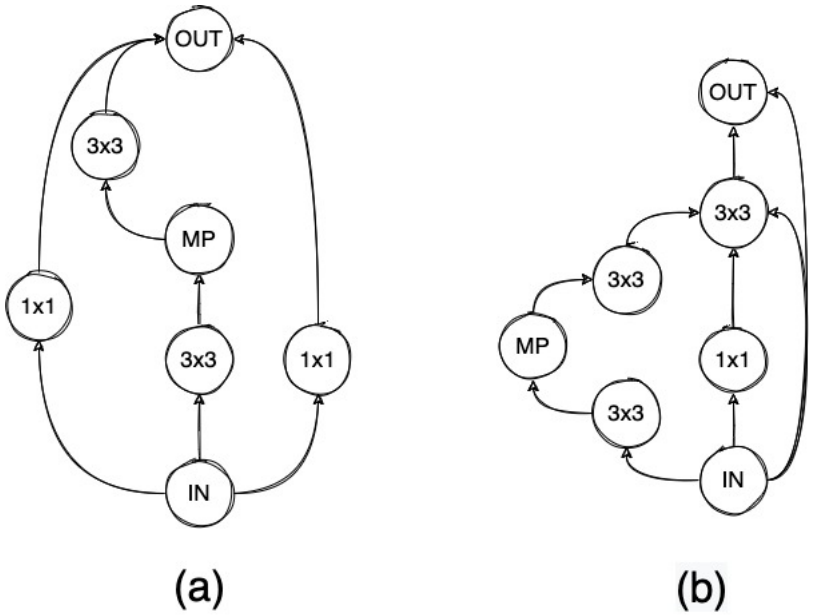

Figure 4.2: Architecture examples

4.2.2 Eliminate Isomorphism

There are models in our search space with different adjacency matrices and different labels, but they are computationally the same; this behavior we call isomorphism. Additionally, suppose there is an edge directly from the input node to the output node that

is not contributing to the results. In that case, therefore, we can prune these nodes and minimize the architecture size without changing the behavior of the whole architecture. Due to the larger size of the search space, it is computationally wasteful to evaluate each and every architecture without knowing the fact of isomorphic behavior in it.

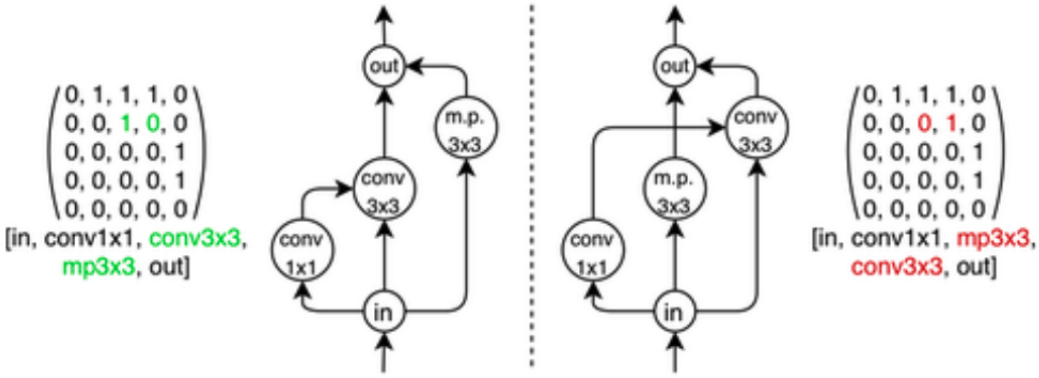

Figure 4.3: Two isomorphic graphs which are different but encode the same [Yin+19]

As a result, we use an iterative graph hashing approach [Yin19], which detects if two graphs are isomorphic or not. This algorithm conducts iterative isomorphism-invariant operations on the graph's vertices, including information from the nearby vertices and its corresponding label. Additionally, it generates a fixed-length hash that recognizes the computationally identical graphs that we call isomorphic graphs. This type of algorithm allows us to identify all unique graphs within the search space and create a representative graph representing all the other unique graphs. We do not have to compute all of them to minimize the expensive training and evaluation [Yin+19].

4.2.3 Hyperparameter Selection

The last but most crucial preprocessing component is hyperparameter selection. We have to choose how much learning rate, which optimizer, batch size, loss function, and how many training epochs we should get the better results. For this purpose, we have tried different combinations of hyperparameters on the same architecture, which is in the figure 4.2 (b)

Results of three optimzers on the different hyperparameters

Figure 4.4 has shown some results of three different optimizers on different following hyperparameter settings:

(a) Batch Size: **256**, Learning Rate: **0.01**, Epochs: **50**
(b) Batch Size: 256, Learning Rate: 0.01, Epochs: **100**
(c) Batch Size: 256, Learning Rate: **0.08**, Epochs: 100
(d) Batch Size: **128**, Learning Rate: 0.08, Epochs: 100
(e) Batch Size: **32**, Learning Rate: 0.08, Epochs: 100

Parameter			**Median Performance**		
Batch Size	Learning Rate	Epochs	SGD	RMS_Prop	Adam
256	0.01	50	0.2655	0.4323	**0.4942**
256	0.01	100	0.3336	0.4395	**0.5361**
256	0.08	100	0.4482	0.3580	**0.5085**
128	0.01	100	0.4831	0.3860	**0.4992**
32	0.01	100	0.5305	0.3639	**0.5649**

Table 4.1: Comaparing optemizers on different hyper-parameters

As shown in the table above as well as in the figure 4.4, Adam optimizer has the highest performance as compared to SGD and RMS-Prop for different hyper-parameters settings.

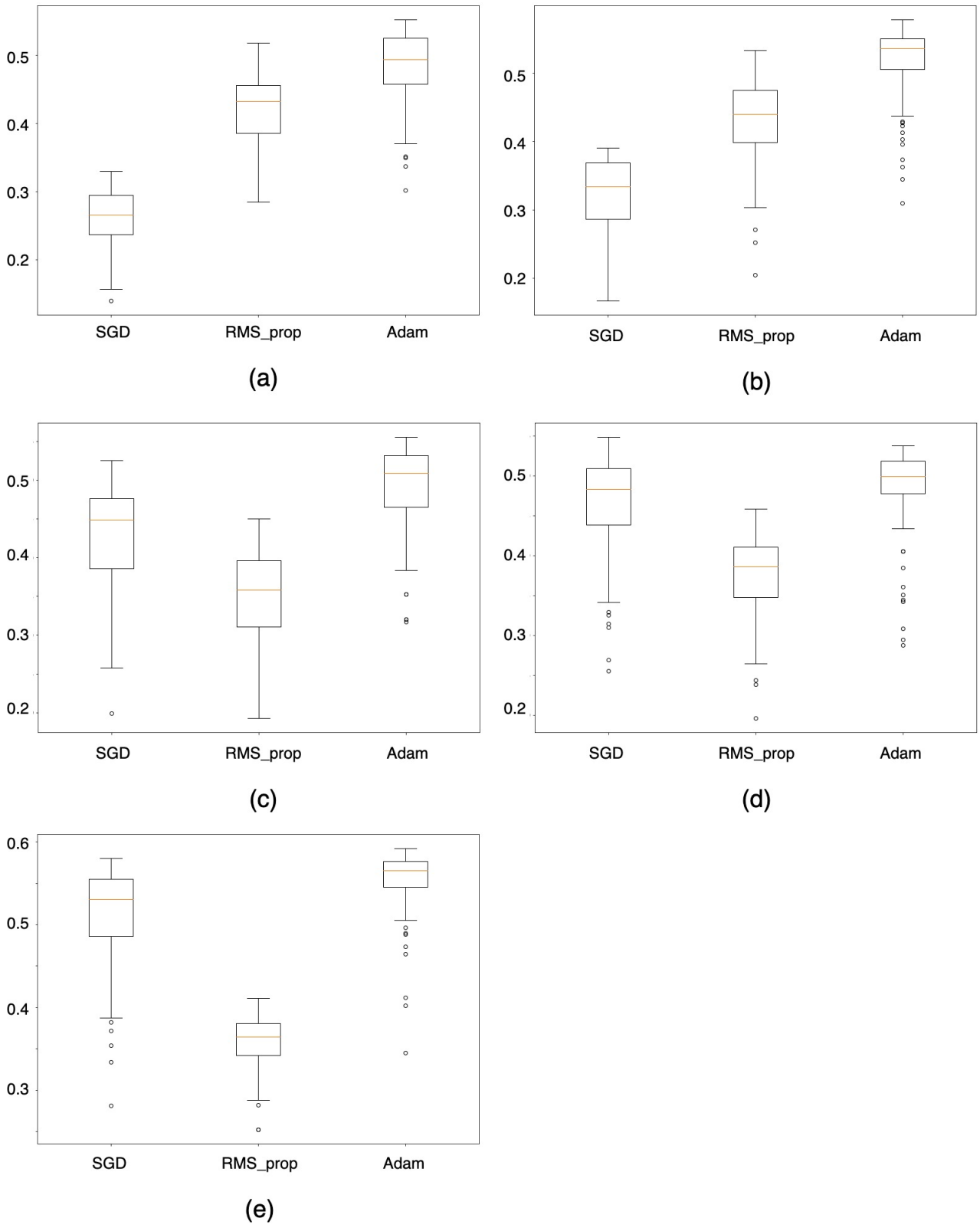

Figure 4.4: Results of optimzers on the different hyperparameters

Comparing the results of the adam optimizers on different batch sizes

After performing these experiments, it is shown that the Adam optimizer performed better than the other two. Therefore, for further preprocessing, we have to perform one final experiment on the Adam optimizer to determine the batch size we will use for the next experiments later on. In the figure 4.5, results have been shown for the batch size selection. And according to the results, batch size 32 has the better results, but it is very time-consuming to compute. So, we chose a 256 batch size, as it will computationally be less time-consuming and has satisfactory results. Therefore, we choose batch size 256 because it will be computationally faster, and we can iterate our experiments more frequently.

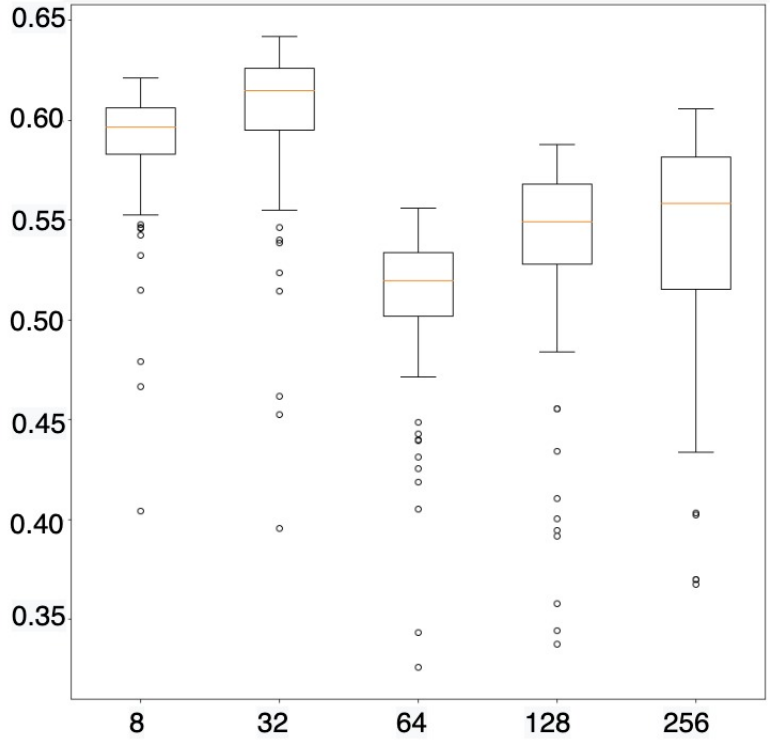

Figure 4.5: Comparison of different batch sizes on Adam optimizer

Final selected hyperparameters

After performing many experiments to select the right hyperparameters, therefore, following are the final hyperparameters that we will use in our further experiments.

Hyper-parameter	Value
Learning rate	0.08
Loss	Cross-Entropy Loss
Optimizer	Adam
Batch size	256
Training epochs	100

Table 4.2: Hyper-parameters used for training and evaluating the architectures.

4.3 Search Strategies implementation

After completing the preprocessing for our experiments, it is now time to discuss which experiments we performed and how it works. The first experiment is related to different search strategies that we performed on our search space.

4.3.1 Random Search

The first search strategy is random search; this is basically applied to reproduce the results of NAS-Bench-101 [Yin+19]. We used our method to randomly sample architectures from the given search space of a certain configuration. In this case, we randomly sampled 1500 encoded architectures and trained these architectures on the CIFAR-10 dataset with 100 epochs, including the hyperparameters as mentioned earlier in the table 4.2. Additionally, we ran this experiment three times so that we could also identify the variance.

Parameter	Value
Learning rate	0.08
Loss	Cross-Entropy Loss
Optimizer	Adam
Batch size	256
Training epochs	100
Sample size	1500
Number of Runs	3
Dataset	CIFAR-10
GPU	GeForce RTX

Table 4.3: Parameters used for training and evaluating the random search strategy.

The details on the results of the random search strategy are given in section results. And it took around 16 days to compute one run using GeForce RTX GPU. Due to time constraint, I could only run this for three times.

4.3.2 Regularised Evolutionary Search

In our experiments, we have used multiple evolution strategies. The first we will discuss is regularised evolutionary search. What is regularised evolution strategy? And how is it different from other evolution strategies? Answers to these questions we have already discussed in the section 2.5.2. In this section, we discuss more about its implementation. These are the following parameters we have used to run this approach.

We have implemented two types of regularised evolution strategies which are as follows:

First type of Regularised Evolution Strategy

Our goal is to evaluate a total of 1500 unique architectures, same as a random search because we have to compare our results with other search strategies on equal grounds. Therefore, we initially randomly sampled 800 architectures from our search space in this type. After that, we have to train and evaluate these architectures. Then in the next step, we picked the best architecture from the previously sampled architectures and sent it further towards the mutation. We chose the same mutation method from the publicly available algorithm of [Yin+19]. Furthermore, we mutated the best architecture 50 times, created new 50 architectures, then discarded 50 worst-performed architectures from the current generation. At last, we send these 50 mutated architectures and the remaining 750 architectures to the next generation.

And same procedure has been done for the next generations. The same total of 800 architectures and after training and evaluation, picked the best architecture from the current generation mutated it again 50 times and discarded the worst 50 architecture in each generation regularly. And after that again new 50 and 750 remaining architectures will make the next generation.

This process will continue until the last generation, which is 15, or will terminate if the next 3 generations will give the same result as it is not improving further. So it means this algorithm achieved some optima; it could be global optima or local optima.

Visualization of this procedure can be shown in figure 4.6.

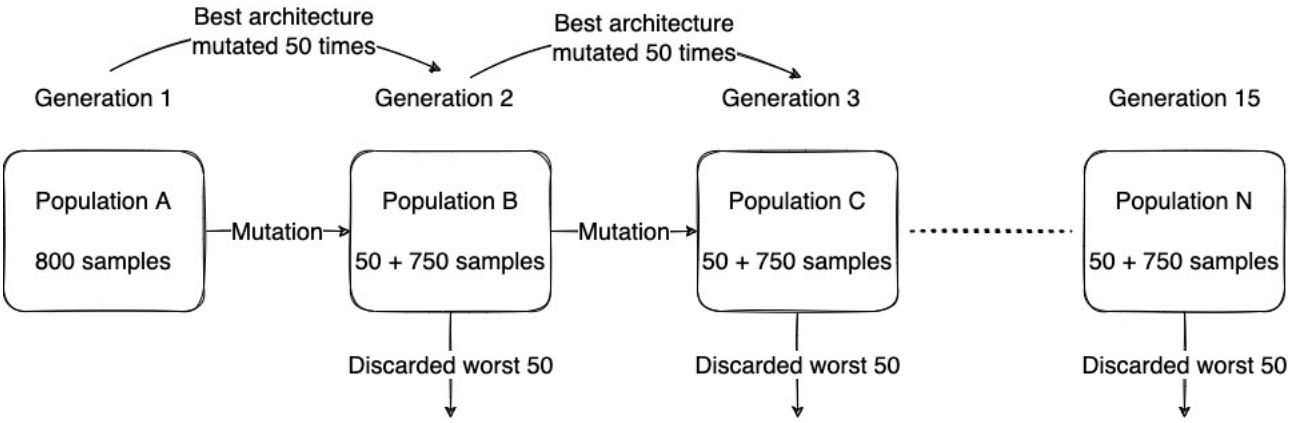

Figure 4.6: Flow chart of first type of Regularised Evolution Strategy

Parameter	Value
Learning rate	0.08
Loss	Cross-Entropy Loss
Optimizer	Adam
Batch size	256
Number of Runs	1
Training epochs	100
Samples in first generation	800
Number of mutations (each generation)	50
Number of Generations	15
Total evaluated models	1500
Dataset	CIFAR-10
GPU	GeForce RTX

Table 4.4: Parameters used for training and evaluating the first type of regularised evolution strategy

Second type of Regularised Evolution Strategy

Again, Our goal is to evaluate 1500 unique architectures, same as the other search strategies, because we must create a comparative study on common grounds. This time, we initially randomly sampled 200 architectures from our search space in this type. After that, we have to train and evaluate these architectures. Then in the next step, we picked the 100 best architectures from the previously sampled architectures

and sent it further towards the mutation [Yin+19]. Furthermore, we mutated these 100 architectures created 100 more after discarding 100 worst-performed architectures from the current generation. At last, we sent these 100 mutated architectures and the remaining 100 architectures to the next generation.

And same procedure has been done for the next generations. The same total of 200 architectures and after training and evaluation, picked the best 100 architectures from the current generation mutated it again and discarded the worst 100 architecture in each generation regularly. And after that, the new 100 and 100 remaining architectures will make the next generation.

This process will continue until the last generation, which is 14, or will terminate if the next 3 generations will give the same result as it is not improving further. So it means this algorithm achieved some optima; it could be global optima or local optima.

Visualization of this procedure can be shown in figure 4.7.

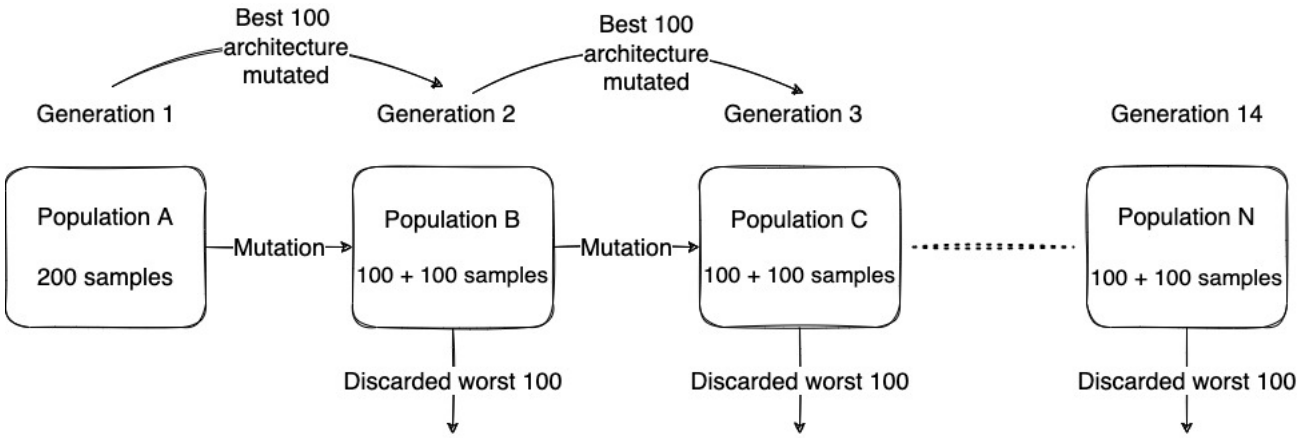

Figure 4.7: Flow chart of second type of Regularised Evolution Strategy

Parameter	Value
Learning rate	0.08
Loss	Cross-Entropy Loss
Optimizer	Adam
Batch size	256
Number of Runs	1
Training epochs	100
Sample size	1500
Samples in first generation	200
Number of mutations (each generation)	100
Number of Generations	14
Total evaluated models	1500
Dataset	CIFAR-10
GPU	GeForce RTX

Table 4.5: Parameters used for training and evaluating the second type of regularised evolution strategy

4.3.3 Genetic Algorithm

In this section, we discuss the implementation of the genetic algorithm (section 2.5.2) in our experiments. Again our goal is to evaluate a maximum of 1500 samples from our search space.

In the selection procedure of generation 1, we initially randomly sampled 200 architectures from our search space in this type. After that, we have to train and evaluate these architectures. Then in the next step, we picked the best two architectures from the previously sampled architectures and sent them further towards the crossover.

Crossover can be done in many ways; we can just swap half models with each other or put some limit on how much swapping can be done. In our experiment, we declared some limits. It will randomly make a chunk of that encoded matrix within that limit and swap it with the second best-performed encoded matrix. For example, if we have a 7×7 matrix multiple times to create multiple candidate children, we set the column ranges, i.e., range 1 within $(0, 3)$ columns and range 2 in $(4, 6)$. And the vertical limit is rows $(1, 3)$. It will be more understandable with the following figure 4.8.

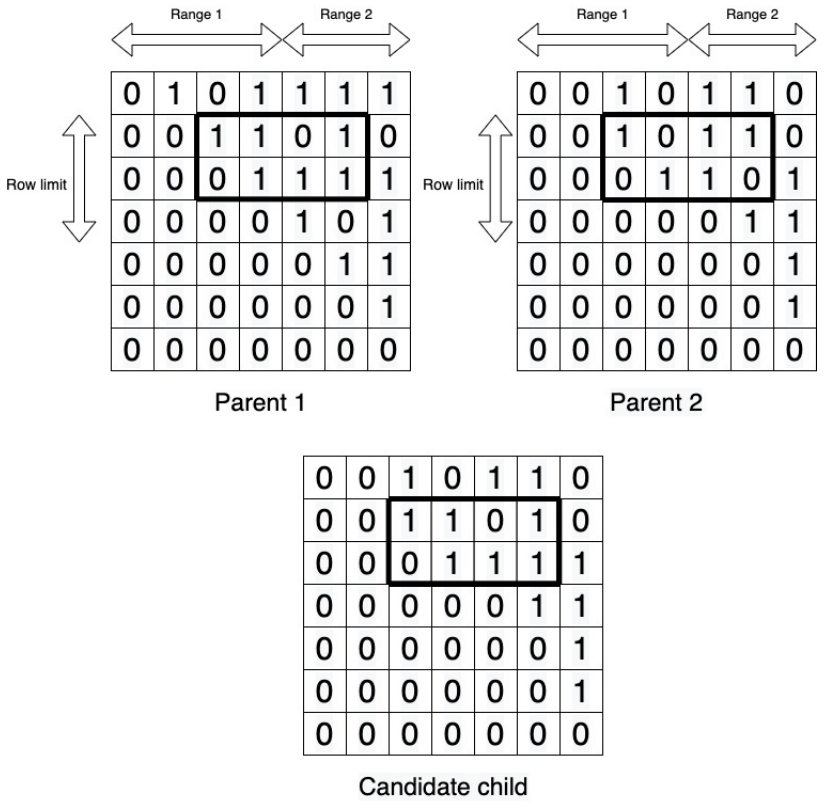

Figure 4.8: Crossover approach

In figure 4.8, parent 1 is the best-performed matrix from the previous generation, and parent 2 is the second-best. Then we have to apply crossover on that; for this purpose, we have to swap some values of parent 1 to parent 2. Therefore, we set some ranges row and column-wise to pick the first and last point of the section. Then, as in the figure, we randomly picked row 2 and column 3 as the first point and row 3 and column 6 as the second point and drew the rectangle. After that, we put this rectangular patch to parent 2 to the exact location as parent 1. Then we have the candidate child after this crossover.

In this procedure, there is a possibility that we can face a problem. Where both these patches are symmetrical. Then there will be a wasteful effort to apply crossover on that. To tackle this problem, a schema formed by Bradley Wallet et al., if A is the number of rows in the matrix, B denotes the number of columns in the matrix, of a location and its left or right neighbor is disrupted at the rate of 1/X and X=A*B+1. However, at the rate B/X, a schema comprising a place and its up or down neighbor is disrupted.

In general, one finds that the problem's representation is not rotationally invariant and that column-oriented schema is far more likely to be disturbed than row-oriented schema [Kit90].

After picking up two best-performed matrices from the first generation of 200, we have done 100 crossovers. After crossover, we have to mutate these 100 matrices same as the previous evolution strategies [Yin+19]. Furthermore, we mutated these 100 and created 100 more after discarding 100 worst-performed architectures from the current generation.

And same procedure has been done for the next generations. The same total of 200 architectures and after training and evaluation, picked the best two architectures from the current generation and sent them again to crossover phase for 100 times and then mutated it and discarded the worst 100 architecture in each generation regularly. And after that, the new 100 and 100 remaining architectures will make the next generation.

This process will continue until the last generation, which is 14, or will terminate if the next 3 generations will give the same result as it is not improving further. So it means this algorithm achieved some optima; it could be global optima or local optima.

Parameter	**Value**
Learning rate	0.08
Loss	Cross-Entropy Loss
Optimizer	Adam
Batch size	256
Number of Runs	1
Training epochs	100
Sample size	1500
Samples in first generation	200
Number of crossovers (each generation)	100
Number of Generations	14
Total evaluated models	1500
Dataset	CIFAR-10
GPU	GeForce RTX

Table 4.6: Parameters used for training and evaluating the genetic algorithm

4.4 Predicting results without training

In this section, we have predicted the results of random graphs without training them.

After training and evaluation are done of random search graphs, we stored its accuracy, precision, recall, f1, and loss in a file consisting of 1500 rows dataset.

There are more than 20 graph properties, but we will utilize only 5 of them for our experiment, i.e., number of nodes, number of edges, density, diameter, and average shortest path length. Because the graph attributes in our dataset are distributed throughout a wide range, we use the MinMax scaler to scale each graph property between a set of values $(0, 1)$.

In the next step, we divide the dataset into a train and test set of 90% and 10%, respectively. After that, train three different regressor algorithms, including Bayesian ridge regressor, Random Forest regressor, and Ada Boost regressor, on the train set with five graph properties as features and its related accuracy, precision, recall, and f1 as the target.

Each regressor algorithm is evaluated in the next step on the test set using an R-squared value to understand how well our data fits each regressor model.

5 Results

This chapter presents the results from the experiments as explained in the previous chapter. First, we start the presentation of our results with search strategies that we applied to our search space to find better architectures. And finally, we show the results of our performance prediction without rigorous training on the dataset.

5.1 Random Search performance

Our first visualization of the search strategies is random search. In which, we took 1500 samples randomly from the given search space. Then train and test three times to determine the different outcomes. As we can see in the figure-5.1, accuracy-wise, there is considerable fluctuation in Run_1. Interestingly, we got both best and the worst architectures in Run_1. Run_2 showed slightly better results as compared to Run_3.

5.2 Regularised Evolution Strategy performance

In regularised evolution strategy, we applied two types of evolutions that are discussed in detail in the section 4.3.2. And the results of these two evolution strategies are as follows.

5.2.1 First type of regularised evolution strategy

The first type of regularised evolution strategy's visualization is shown in figure-5.2. Generation 1 consists of 800 random samples, then generation by generation it mutates it further to improve the outcomes. Until the tenth generation, the median of the

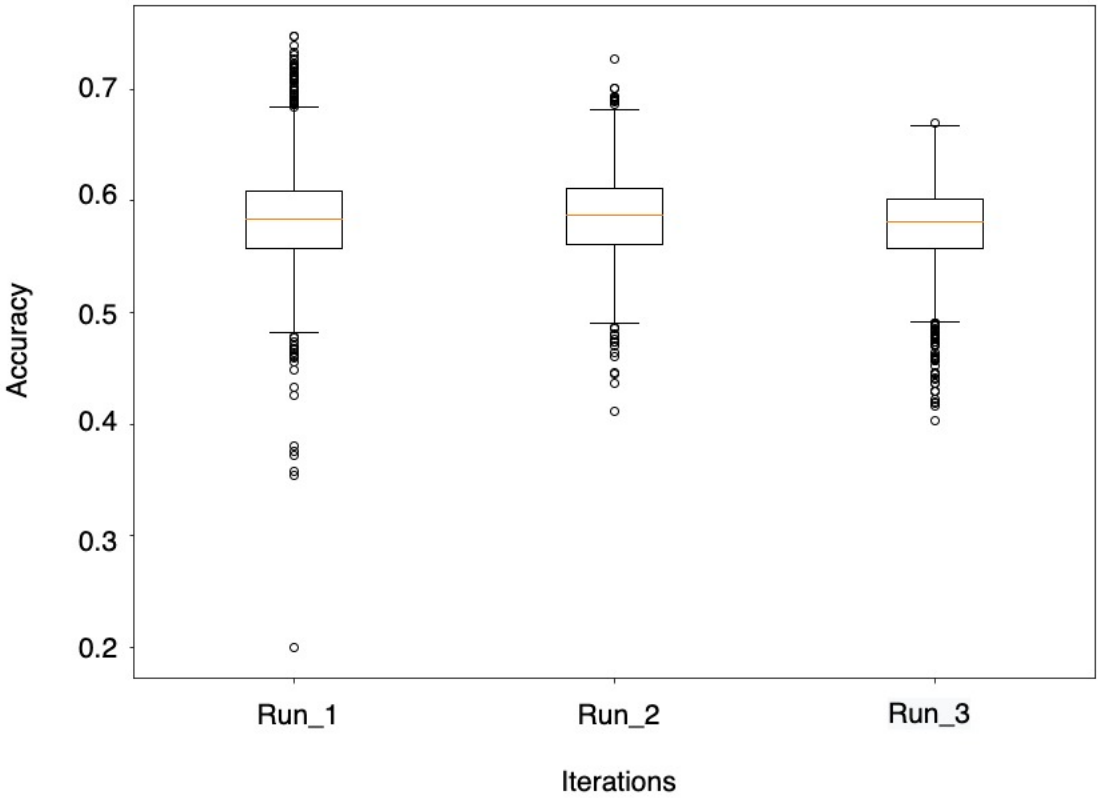

Figure 5.1: Three iterations of random search performances after training with 100 epochs and other hyperparameter settings, shown in the table 4.3.

outcomes is improving but after that, it does not show any improvement. This search strategy achieved the best accuracy (around 73%) in the eighth generation and worst in the third generation (around 36%).

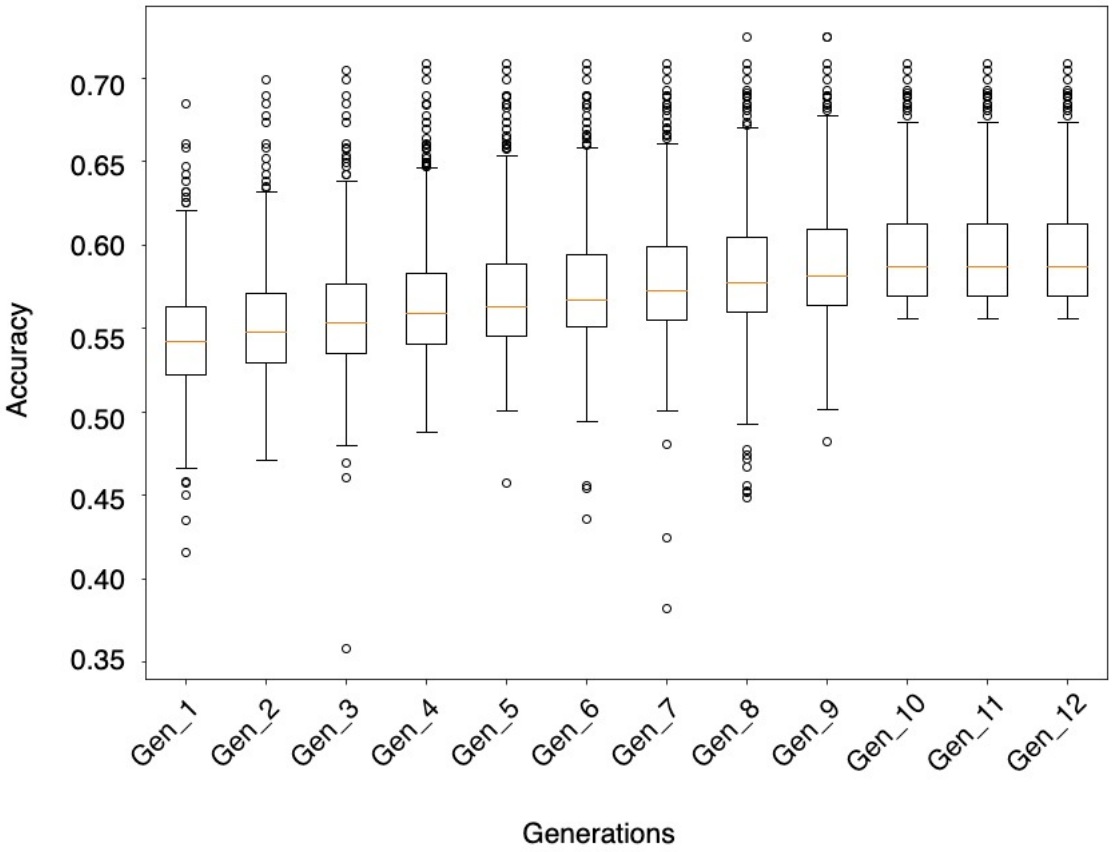

Figure 5.2: First type of regularised evolution strategy performance after training with 100 epochs and other hyperparameter settings, shown in the table 4.5.

5.2.2 Second type of regularised evolution strategy

The second type of regularised evolution strategy's visualization is shown in figure-5.3. As we can see, until the fourth generation, trend of the accuracy is increasing gradually, it does not show any improvement as the algorithm could not improve the architectures further. Additionally, we can notice that we could not improve the accuracy (around 74%) of the best architecture after the second generation.

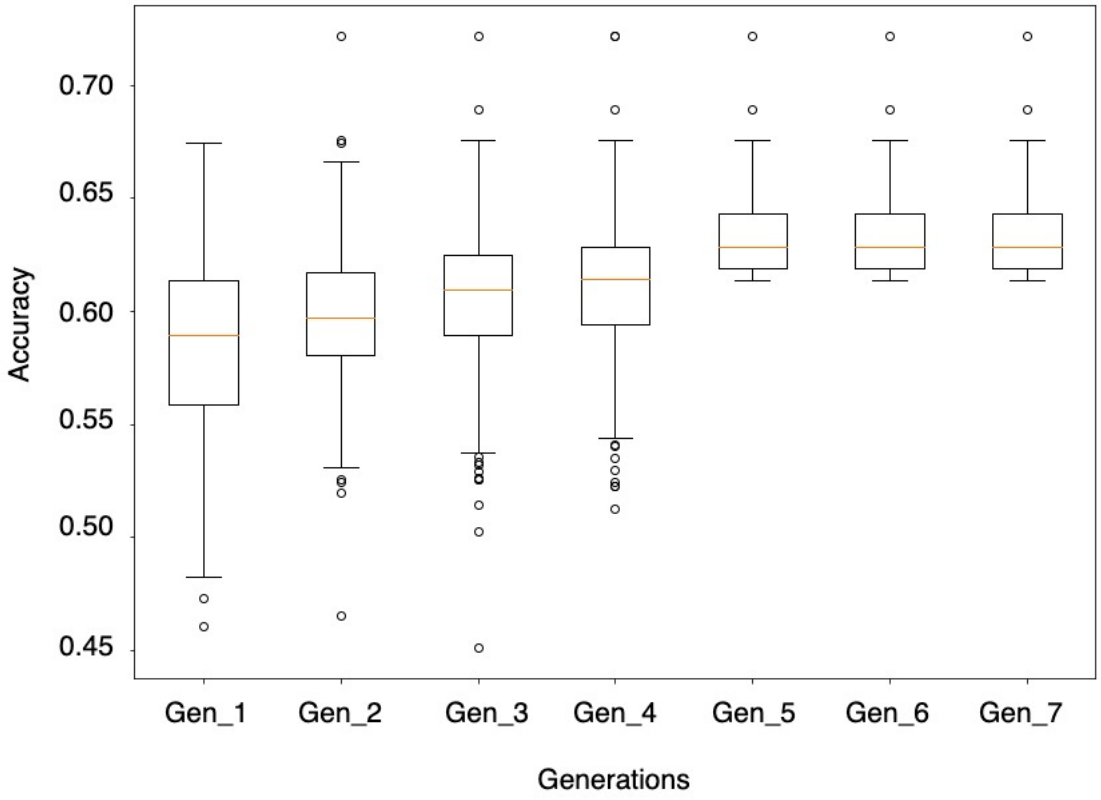

Figure 5.3: Second type of regularised evolution strategy performance after training with 100 epochs and other hyperparameter settings, shown in the table-4.5.

5.3 Genetic algorithm performance

The last box-plot visualization of the search strategies is the genetic algorithm. As we can see in figure-5.4, till the third generation, there is a gradual increase in the accuracy, and after that, it does not show any improvements as the algorithm could not refine the architectures any further.

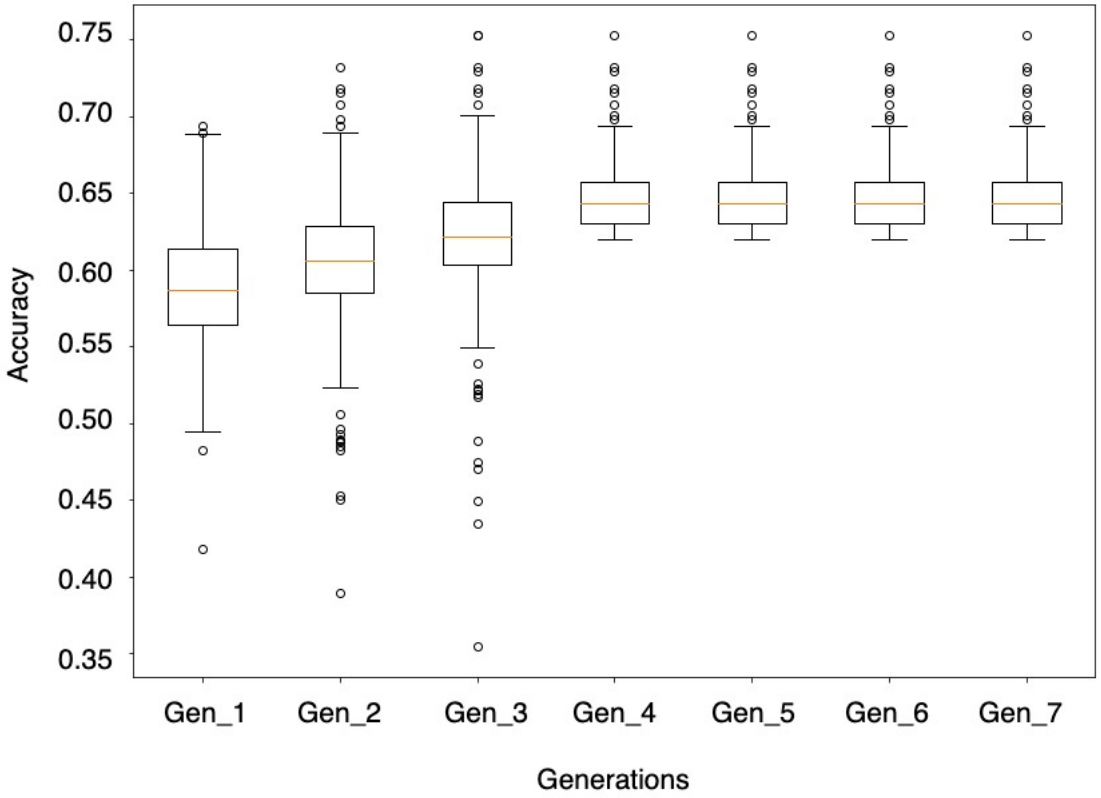

Figure 5.4: Genetic algorithm results after training with 100 epochs and other hyperparameter settings, shown in the table 4.6.

5.4 Comparative results of evolution strategies

Finally, we combined the results of all three evolution strategies in the following line-plot 5.5. As we can see, all of them showed different trends in their medians, and the first type of evolution strategy achieved a lower median compared to the other two. But genetic algorithm took lesser time to achieve local optima and showed the best median results.

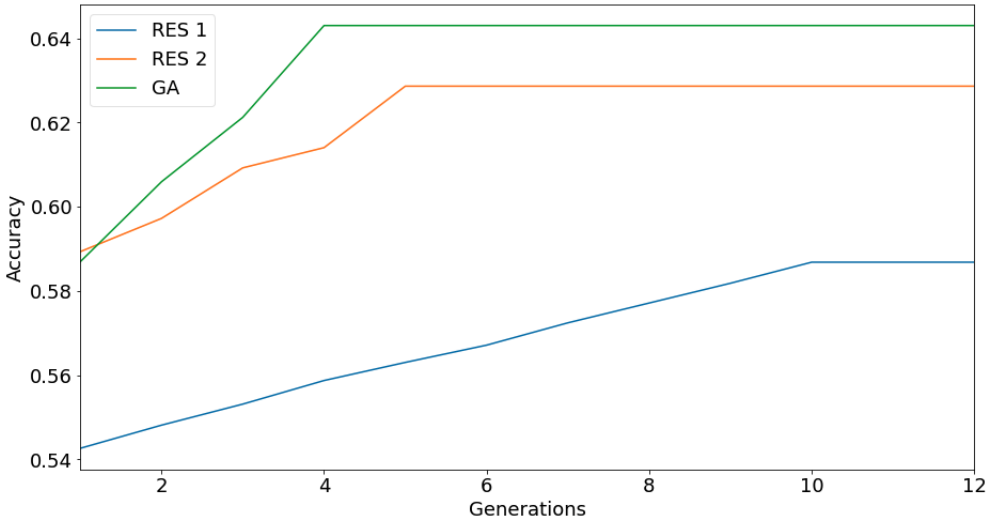

Figure 5.5: Comparative results of evolution strategies

5.5 Performance prediction

We created a dataset of randomly sampled architectures from our search space. After training and evaluating the architectures, we have filled the dataset with its performance metrics (accuracy, precision, recall, F1). Finally, we train three regressor algorithms on this dataset, and based on the resulting R-squared value, we determine if it is possible to predict a randomly structured architecture's performance.

For each target attributes, the following table shows the R-squared values from each regressor algorithm we used:

Target Attributes	Bayesian Ridge	Random Forest	AdaBoost
Accuracy	0.03556	0.14619	0.11592
Precision	0.13623	0.19597	0.22101
Recall	0.03556	0.14713	0.15443
F1-score	0.01363	0.09250	0.09823

Table 5.1: R-squared values from each regressor algorithm, for each target attributes help determine closeness of our data to the fitted regression line.

The R-squared values show how much our data is close to the regression line. If the R-squared value is higher and closer to one, it means it is closely fitted to the regression line and better fit for the model, and vice versa.

R-squared values achieved from Random Forest are better compared to the other two. However, these values are still low to draw any meaningful correlation between the graphs properties of the base adjacency matrix and target performance attributes.

6 Discussion

In this section, we describe our findings regarding the experiments. Results of different search strategies that we applied on neural architecture search (NAS) during the research are discussed briefly, along with the findings on the exploration of the possibility to predict the NAS results without retraining the architectures using regression.

6.1 Comparative study of different search strategies

Random Search strategy

In the first experiment, we applied a random search strategy to our search space to find for better architectures. We performed experiments using random search multiple times with 1500 samples in each run. As shown in the figure-5.1, there are some outliers in the results, with no significant improvement in the median of multiple executions. It is evident that the best architectures are found in the first run with no noticeable improvements in the subsequent execution.

First type of regularised evolution strategy

In the first type of regularised evolutionary search strategy, we performed this experiment for a maximum of fifteen generations. First twelve generations results are shown in figure-5.2. The prominent outcome of this experiment is the best accuracy achieved in the eighth generation and the worst accuracy in the third generation. However, the positive thing in this strategy is the median results keep on improving gradually until the tenth generation. After the 10th generation, the performance is identical because the same architecture from the previous generation is picked for mutation.

Second type of regularised evolution strategy

In the second type of regularised evolution strategy, we performed this experiment for a maximum of 14 generations. The results from first 7 generations are shown in Figure-5.3. It can be observed that the best architectures are found after the fifth generation, showing no viable improvement in the successive generations. Another considerable outcome is that from the second generation, the best architecture (accuracy-wise) remained the same throughout the experiment. Additionally, the worst architecture after mutation can be seen in generation 3. As opposed to the first type of regularised search strategy, this strategy found the optimal architecture early and with better performance.

Genetic algorithm

The experiment with genetic algorithm is also performed for a maximum of 14 generations. The results of the experiments from first seven generations are shown in figure-5.4 It can be seen that the best architectures using genetic algorithm are found in the fourth generation, after which there is no visible improvement seen in the median value for the accuracies of the found architectures. The best performing architecture can be seen in the thirds generation for the first time, which persists through out the experiment for the rest of the generations that follow. Additionally, the worst performing architecture is also found in the third generation. Compared to the regularised search strategies, Genetic algorithm is quickly able to find an optimal architecture, with slightly better performance than the second type of search strategy.

Overall comparison of evolution strategies

Figure-5.5 shows the overall comparative study of all the evolutionary search strategies (generation-wise medians) that are evaluated in our research. The most visible outcome is that the genetic algorithm (GA) achieved the best median accuracy, and the first type of RES achieved the least median accuracy. However, the first type of RES evolves till the tenth generation but very slowly as compared to the other two. On the other hand, GA evolves fastest to achieve the optimum results. Based on these results, it is safe to say that using this strategy, we can find an architecture with better performance in less time and less computing power.

6.2 Prediction based on graph attributes

In this part, we trained three regressor algorithms to find if we can employ graph properties of the architectures using adjacency matrix encoding, to predict their performance based on how well our data fits a model. As we can see in the table 5.1, the highest R-squared value achieved is 0.22 with AdaBoost regressor against Precision.

Among all four target attributes, Precision is the only metric with a consistent R-squared value of above 0.10, which means it can barely explain any of the variations in the response variable. On the other hand, F1 receives the lowest R-squared value for each regressor, meaning it fails to explain any of the variations in the response variable. It can depend on many things; for example, if we increase the size of the graphs and make them denser, the result may differ.

7 Conclusion

The main objective of this research, as described in chapter 1, is to compare different search strategies on neural architecture search. Chapter 4 briefly explained the whole methodology and procedure to perform our experiment, and at the end, chapter 5 presented the outcomes.

In the first phase of the research, we performed three search strategies to find the best-performing architectures on our search space: random search, two types of regularised evolutionary search strategies, and genetic algorithm. Random search lacked to provide better architectures, as it cannot evolve. However, if we run the random search multiple times (at least 20), we can observe better stability and variance. Moreover, based on our results of evolution strategies, the genetic algorithm showed faster to achieve its optimum median accuracy than regularised evolution strategy. The results depend on many aspects such as hyperparameters selection, problem statement, the structure of the search space, and many more. Therefore, we can consider these aspects for future work to achieve better and different results. Apart from these search strategies, we can also explore more techniques such as covariance matrix adaptation evolution strategy (CMA-ES) on continuous search space.

In the second phase of the research, the performance prediction experiment results identified that the target attributes accuracy, precision, recall, and f1-score have a very weak fit with all three regressors used for this experiment. The poor R-squared values are probably because lack of good quality features. This can be improved by increasing the initial adjacency matrix's size and increasing the chances of having dense and more connected graphs. This has not only a higher chance of having architecture with better performance but also a good R-squared value in the final experiment.

A Abbreviations

NAS	Neural Architecture Search
ES	Evolution Strategy
SD	Standard Deviation
NLP	Natural Language Processing
SLP	Single Layer Perceptron
MLP	Multilayer Perceptron
RS	Random Search
RES	Regularised Evolution Strategy
GA	Genetic Algorithm
ANN	Artificial Neural Network
ReLU	Rectified Linear Unit
ELU	Exponential Linear Unit
AutoML	Automated Machine Learning
DL	Deep Learning
TF-Record	Tensorflow Record
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory

B Code

The source code with implementation of all search strategies used in the research is available in the following repository:

`https://git.fim.uni-passau.de/munir/comparing-search-strategies-on-nas`

Bibliography

[Axe87] Robert Axelrod. "The evolution of strategies in the iterated prisoner's dilemma". In: *Genetic algorithms and simulated annealing* (1987), pp. 32–41 (cit. on p. 31).

[Bar18] Christopher S von Bartheld. "Myths and truths about the cellular composition of the human brain: A review of influential concepts". In: *Journal of chemical neuroanatomy* 93 (2018), pp. 2–15 (cit. on p. 5).

[BB12] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of machine learning research* 13.2 (2012) (cit. on p. 26).

[DY20] Xuanyi Dong and Yi Yang. "Nas-bench-201: Extending the scope of reproducible neural architecture search". In: *arXiv preprint arXiv:2001.00326* (2020) (cit. on pp. 1, 30).

[EMH19] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural architecture search: A survey". In: *The Journal of Machine Learning Research* 20.1 (2019), pp. 1997–2017 (cit. on pp. 23, 24, 32).

[HOT06] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets". In: *Neural computation* 18.7 (2006), pp. 1527–1554 (cit. on pp. 5, 15).

[HKV19] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019 (cit. on p. 23).

[Ji+21] Fang Ji et al. "Discerning the painter's hand: machine learning on surface topography". In: *Heritage Science* 9.1 (2021), pp. 1–11 (cit. on p. 5).

[Kit90] Hiroaki Kitano. "Designing neural networks using genetic algorithms with graph generation system". In: *Complex systems* 4 (1990), pp. 461–476 (cit. on pp. 23, 47).

[KRS22] Andreas Klos, Marius Rosenbaum, and Wolfram Schiffmann. "Neural Architecture Search based on Genetic Algorithm and Deployed in a Bare-Metal Kubernetes Cluster". In: *International Journal of Networking and Computing* 12.1 (Jan. 3, 2022). URL: `http://www.ijnc.org/index.php/ijnc/article/view/276` (visited on 01/03/2022). published (cit. on p. 32).

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105 (cit. on p. 35).

[LSY18] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "Darts: Differentiable architecture search". In: *arXiv preprint arXiv:1806.09055* (2018) (cit. on pp. 31, 32).

[MP56] WS McCullock and W Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity. Archive copy of 27 November 2007 on Wayback Machine". In: *Avtomaty [Automated Devices] Moscow, Inostr. Lit. publ* (1956), pp. 363–384 (cit. on p. 6).

[Men18] Ahmed Menshawy. *Deep Learning By Example: A hands-on guide to implementing advanced machine learning algorithms and neural networks*. Packt Publishing Ltd, 2018 (cit. on p. 10).

[MS95] Jacob MJ Murre and Daniel PF Sturdy. "The connectivity of the brain: multi-level quantitative analysis". In: *Biological cybernetics* 73.6 (1995), pp. 529–545 (cit. on p. 5).

[Nek+19] Vladimir Nekrasov et al. "Fast neural architecture search of compact semantic segmentation models via auxiliary cells". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 9126–9135 (cit. on p. 23).

[Pie20] Valdimir Pieter. *Parameters for the best convergence of an optimization algorithm On-The-Fly*. 2020. arXiv: `2009.11390 [math.OC]` (cit. on pp. 27, 28).

[Ras15] Sebastian Raschka. *Single-Layer Neural Networks and Gradient Descent*. 2015. URL: `https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html` (cit. on p. 15).

[Rea+19] Esteban Real et al. "Regularized evolution for image classifier architecture search". In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 4780–4789 (cit. on pp. 23, 28, 30).

[Ren+20] Pengzhen Ren et al. "A comprehensive survey of neural architecture search: Challenges and solutions". In: *arXiv preprint arXiv:2006.02903* (2020) (cit. on pp. 1, 2).

[Ros58] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 6).

[SM02] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127 (cit. on p. 23).

[WMS96] Bradley Wallet, David Marchette, and Jeffery Solka. "A Matrix Representation for Genetic Algorithms". In: (June 1996) (cit. on pp. 29, 31).

[Whi+20] Colin White et al. "A study on encodings for neural architecture search". In: *arXiv preprint arXiv:2007.04965* (2020) (cit. on p. 31).

[WLT21] Meng-Ting Wu, Hung-I Lin, and Chun-Wei Tsai. "A Training-free Genetic Neural Architecture Search". In: *Proceedings of the 2021 ACM International Conference on Intelligent Computing and its Emerging Applications*. 2021, pp. 65–70 (cit. on p. 32).

[Yin19] Chris Ying. "Enumerating unique computational graphs via an iterative graph invariant". In: *arXiv preprint arXiv:1902.06192* (2019) (cit. on p. 37).

[Yin+19] Chris Ying et al. "Nas-bench-101: Towards reproducible neural architecture search". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 7105–7114 (cit. on pp. 1, 3, 30, 31, 35–37, 41, 42, 44, 47).

[ZL16] Barret Zoph and Quoc V Le. "Neural architecture search with reinforcement learning". In: *arXiv preprint arXiv:1611.01578* (2016) (cit. on pp. 23, 31).

[Zop+18] Barret Zoph et al. "Learning transferable architectures for scalable image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8697–8710 (cit. on pp. 23, 35).

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, February 5, 2022

_____

Munir, Muhammad Arsal