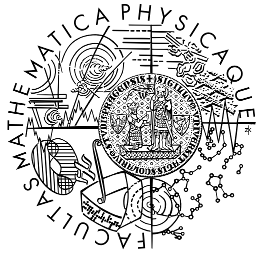


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Bc. Renáta Pivodová

Multi-objective Neural Architecture Search

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Martin Pilát, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I dedicate this work to my family and my partner. Also my great thanks belongs to supervisor of this thesis Martin Pilát for his support, mentoring and advice.

Computational resources were provided by the e-INFRA CZ project (ID:90140), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Title: Multi-objective Neural Architecture Search

Author: Bc. Renáta Pivodová

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Neural architecture search is a promising approach to automatic neural network architecture design, which can save a designer's work. The real world contains a lot of problems, which are time-consuming to solve even by neural architecture search techniques. A lot of these problems require architectures optimized according to different criteria such as quality, time of search, etc. In this work, we present two methods extending the CoDeepNEAT, a state-of-the-art neural architecture search algorithm. The Lamarckian CoDeepNEAT is the CoDeepNEAT enriched with weight inheritance implementation inspired by the Lamarckian theory of evolution. The Multi-objective CoDeepNEAT performs a multi-objective minimization of two chosen neural network objectives - the error rate and the number of floating point operations. Thanks to the base NSGA-II algorithm, the Multi-objective CoDeepNEAT searches for well-performing and fast networks. The methods are evaluated on the MNIST and CIFAR-10 datasets.

Keywords: neural architecture search multiobjective optimization lamarckism evolutionary algorithm CoDeepNEAT neural networks

Contents

Introduction **3**

1 Evolutionary Algorithms **5**
1.1 Biological Inspiration 5
1.2 Individual 5
1.2.1 Genotype and Phenotype 5
1.2.2 Fitness 6
1.3 Genetic Operators 6
1.3.1 Selection 6
1.3.2 Crossover 7
1.3.3 Mutation 9
1.4 Genetic Algorithms 10
1.5 Coevolution 10
1.6 Multi-objective EA 12
1.6.1 Pareto-based Methods 12
1.6.2 NSGA-II 13

2 Neural Networks **15**
2.1 Biological inspiration 15
2.2 Perceptron 15
2.3 Feedforward Neural Networks 16
2.4 Convolutional Neural Networks 17

3 Neural Architecture Search **21**
3.1 NAS Components 21
3.2 ENAS 24
3.3 LEMONADE 25
3.4 CoDeepNEAT 25
3.4.1 Development 26
3.4.2 Principles of CoDeepNEAT 27

4 Proposed Approach **31**
4.1 LamaCoDeepNEAT 31
4.2 Multi-objective LamaCoDeepNEAT 33

5 Experiments **39**
5.1 Used Implementation 39
5.2 Datasets 40
5.2.1 MNIST 40
5.2.2 CIFAR-10 41
5.3 Experiments with MNIST 41
5.4 Experiments with CIFAR-10 47
5.4.1 CIFAR-10 - Short Version 48
5.4.2 CIFAR-10 - Short Version, Higher Mutation Probability . . 50
5.4.3 CIFAR-10 - Long Version 54

Conclusion **59**

Bibliography **61**

List of Figures **67**

List of Tables **71**

A Attachments **73**

A.1 Hardware 73

A.2 Digital Attachements 73

Introduction

Artificial Intelligence (AI) is a beneficial field making work easier and more pleasing in many fields. An image recognition model can perform melanoma (a type of skin cancer) detection in medicine field. A recommendation system, which recommends to its users what kind of a movie they may like based on provided data, can be a core for streaming platforms in the movie industry. And a text generator can produce a summary of stock market news, which is then checked by an editor and published, in the journalism industry. But AI is helpful in daily life tasks too. An AI model can be a strong opponent in a chess game. A smartphone camera may hide your photographic ineptitude by automatically setting appropriate camera parameters. Artificial neural networks (ANN) are responsible for the biggest part of AI application success and the fast growth of this field. With still improving computer and memory resources, the development of ANN tools seems to be unstoppable.

But still, the process of creating an AI system based on ANN is complex and very demanding. The development workflow consists of several steps: data collecting, data preprocessing, neural network architecture design, neural network training, evaluating, etc. Some of the steps are repeated, of course, and the final cost of the whole procedure increases. Every attempt to simplify counts.

One of the successful attempts led to an establishment of a new computer science discipline Neural Architecture Search (NAS). NAS tries to search and produce a neural network architecture as a result of a problem defined by a developer. This way, the developer does not need to manually design a NN architecture based on their knowledge and findings, because NAS will do it for them. So it seems, that the automation of the design part of the model development is done. During NAS existence, different approaches have appeared, for example, neural architecture search based on evolutionary algorithms inspired by the natural evolution process.

In practice, using NAS for designing a neural network for hard and complex problems is still very time-consuming. Every network which is found during the search process must be tested, how much it is suitable for a given problem. This is usually done through NN training and evaluation, a straightforward, but demanding way. So applying some time-saving techniques is a spot-on proposal. We study common time-saving techniques appropriate for neural network search and implement weight inheritance based on our findings. This Lamarckian evolution theory-inspired approach does not waste already trained weights from the neural network, rather it passes them to the network's successor with very similar architecture and parameters, simply said.

As we said, NAS is used for the creation of deep neural networks, which can solve some tasks. But a lot of real-world problems can not be solved by optimizing only one chosen characteristic of the solution. For example, a buyer has several criteria for buying a car. What about the cost, car fuel consumption, comfort, etc. And if they can choose only one criterion, then which one is the best? The continuing increase of computer capability allows us to design a neural architecture search method focused on multi-objective optimization, which is exactly the content of this thesis.

To sum it up, in this thesis, we focus on studying existing state-of-the-art neural architecture search algorithms, methods for decreasing computational time, and multi-objective optimization techniques. Then based on our findings, we extend a chosen algorithm by adding two studied methods, which results in designing two proposed algorithms - one is based only on weight inheritance and the other is its multi-objective version. Both algorithms are tested and evaluated in experiments on two datasets.

In Chapter 1, we introduce concepts of evolutionary algorithms and in Chapter 2, we describe the basics of neural networks, mainly convolutional neural networks, which are both needed for understanding our proposed solutions. In Chapter 3, a neural architecture search is presented. We mention several NAS methods and NAS components. Also, we focus on a closer description of a chosen algorithm, which served as a base for our proposals. Chapter 4 introduces our proposed methods and extensions. And, finally, Chapter 5 shows a description and results of our experiments.

1. Evolutionary Algorithms

Evolutionary algorithms (EA) [1] is a field of computer science strongly inspired by nature and its mechanisms. Evolutionary algorithms mimic the real evolution of beings in a significantly shorter and simple way. The beautiful idea of EA allows us to find solutions to given problems without knowing anything about the way how to solve them in exchange for our computer resources.

1.1 Biological Inspiration

Generally speaking, evolution is a process working with a population of individuals. The primary task is to change an individual's characteristics during the process and find the most suitable traits for living in a given environment. These characteristics are also responsible for the individual's success in being selected for mating. Due to mating, characteristic traits are passed from parents to their offspring, but sometimes not in the same state as mutation may be applied. All these findings were observed and processed by Charles Darwin in his book *On the Origin of Species* [2].

1.2 Individual

In a computer world, a *population* of *individuals* is a set of solutions to the problem we want to solve, known as an *environment*. Every individual can be interpreted in two ways: as a *genotype* and as a *phenotype*.

1.2.1 Genotype and Phenotype

The genotype is an individual's full information, which is an object of evolution and heredity. On the other hand, the phenotype is a bundle of the individual's actual properties observable in the environment such as morphology or behavior. For example, a human's genotype is DNA and phenotype is a human's body. Graph genotype is a pair of an edge set and a node set and phenotype is the graph itself.

For many problems, it is a challenging task to find the right encoding from phenotype to genotype [3]. In other cases there is no need to look for an encoding and evolution can be applied directly to phenotype. One of the most used and easiest encodings is a *binary encoding*. An individual is encoded as a string of binary values. *Alphabet, integer* and a *floating point encodings* works in similar way.

A *neural networks encoding* is a complex field that can be divided into three types depending on what you want to evolve - weights, architecture, or both. A weights encoding is typically done in the floating point way where the phenotype of the network is a vector of weights. The architecture can be encoded as a matrix, for example. But in practice, encoding is usually more complex as it is more suitable for the implementation of the problem. We will discuss some of the specific solutions later.

1.2.2 Fitness

A *fitness function* is an objective function serving for evaluating how good or bad a solution (individual) is. After obtaining an individual's fitness, some evolutionary principles (e.g. parent selection) work with this value and therefore population development is fitness-driven. Artificial evolution is not blind nor completely random and respects Darwinian natural selection, which is also driven by the individual's survival and mating skills.

The fitness function is a problem (environment) specific tool. Designing appropriate function is not very straightforward for complicated and rich environments since it is not only about describing the ideal solution. A developer must not forget about computational efficiency and time of solution evaluation. Executing a time-consuming function for every individual in every generation may waste plenty of resources.

An example of a trivial problem and its fitness is finding a binary vector with the longest distance between two zeros. In such a case, one of the possible fitness functions returns a maximal number of ones between two zeros. An example of a more complicated problem is finding an artificial neural network classifying digits. Computing the fitness function includes training on a dataset and evaluating the proposed network. The fitness value then can be its error rate.

1.3 Genetic Operators

To make any evolutionary algorithm successful and heading towards the right solution, we need to define and apply so-called *genetic operators*. The genetic operators are subprocesses that imitate natural principles responsible for the development and reproduction of appropriate genomes in the real world. There are three main types of operators: selection, crossover, and mutation.

1.3.1 Selection

Selection [4] is an operator which selects preferred individuals from a population and creates a new subpopulation. That is done by transferring the individuals which were selected according to their fitness. In contrast with crossover and mutation, selection is independent of a chosen genome encoding (unless we count the fitness computation).

Selection can be used in two ways depending on what kind of the subpopulation we want to obtain and in which part of evolution we execute selection.

Parental selection is a type of selection comparing and choosing those individuals from the population, which will serve as parents. Their main goal is to reproduce and create a new population of possibly better children. Parental selection is a crucial part of every evolution.

In some cases, it is convenient to use **population selection** for selecting adequate individuals which will be propagated into the next generation. Typically, this selection unifies a population of parents and a population of newly created offspring, then picks out the fittest individuals and creates a fresh population for the next round of evolution. This is usually done as the last step of the iteration.

Population selection supports *elitism*. This phenomenon retains the fittest genomes unchanged throughout the whole process. For every generation, the best individual of the current generation is never worse than the best individual of the previous generation.

Elitism can be practiced unintentionally, meaning that passing the best parents into a new population relies only upon the probability of selection. Or one can define an exact number of the elite parents who are transferred into the new population before the selection part.

Examples of Selection

There are numerous implementations of selection operators. Let us describe those commonly used in *genetic algorithms*, which will be discussed in the next section.

Roulette wheel selection computes a probability p_i of being selected for every individual i. If f_i is a fitness of individual i and N is the number of individuals in a population, then p_i is obtained as:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}. \tag{1.1}$$

Then we need to divide an interval $[0, 1)$ into N intervals corresponding to computed probabilities. Finally, choose an individual by generating a random number from the same interval (i.e. $[0, 1)$). Repeat the last step until we have the desired number of individuals.

This method is named after its similarity to roulette wheels. Imagine that we divide a roulette wheel by drawing borders of areas. Every individual gets one area, which must correspond by its size to an individual's fitness value. Then we spin the wheel, throw a ball and wait for the result.

In Fig 1.1, there is a visualization of roulette wheel selection of five new individuals i_1,..., i_5 from six existing individuals j_1,..., j_6. Old individuals split the roulette wheel accordingly to their fitness-based probability of selection p_1,..., p_6. Red lines interpret five selected individuals. In this case, a newly selected group contains $i_1 = j_2, i_2 = j_1, i_3 = j_6, i_4 = j_4, i_5 = j_1$.

Tournament selection randomly chooses k genomes from a population and performs a tournament amongst them. Typically, k is a small number, like 2, 3, or 5. The winner is the fittest candidate (with some probability p) and moves to the new population. *Binary tournament selection* choose $k = 2$ candidates to the tournament. The winner of the binary tournament is the fitter individual with some high probability. Tournament selection with $k = 1$ is equivalent to *random selection*. Selection with $p = 1$ selects only the best candidates and is called *deterministic*.

Tournament selection mainly differs from roulette wheel selection in being independent of individual's exact fitness value. The tournament is also suitable for negative fitness values.

1.3.2 Crossover

After retrieving plausible candidates for their reproduction, it is time for *crossover* [5, 3]. Crossover, also called a *recombination*, is a genetic operator responsible for

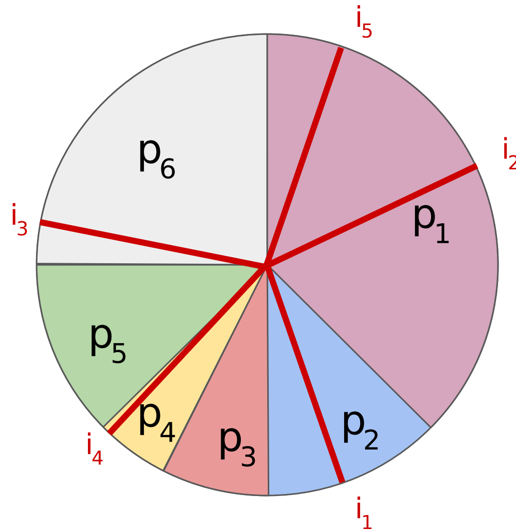


Figure 1.1: Roulette wheel selection of five individuals.

producing new individuals which are somehow related to origin individuals called *parents*. Typically, a combination of genetic information of two parents leads to obtaining one or two new children. This is analogous to a *sexual reproduction* in biology. Children can be also obtained by copying whole genetic information from one parent, in this case, we talk about an analogy to an *asexual reproduction*. In this case, crossover is not applied at all, therefore the *mutation* is very important. Methods from a field of *evolutionary programming* use no or little crossover.

Examples of Crossover

This operator differs for various genome encoding, so the number of existing crossover methods is enormous. The following examples are just the tip of the crossover iceberg, other examples are listed in [5].

K-point crossover is designed for the recombination of genomes represented by arrays of values. K-point crossover can be described in three easy steps. First, choose the value of k, usually, it is some small number like 1, or 2 (one-point, two-point crossover). Second, randomly generate k different indexes to individuals which defines where the individuals are meant to be split into $k+1$ subarrays. Finally, receive one or two children by an alternate combination of those subarrays.

An example of two-point crossover of parents P_A and P_B resulting in two children C_A and C_B with two crossing positions 2 and 4 is shown in Figure 1.2.

Uniform crossover is a very simple and widely employed crossover method thanks to its universal usage. The main idea can be illustrated as flipping a coin every time we want to decide which gene is inherited from what parent. This crossover can create two opposite children from two parents at once, similar to k-point crossover. Some modifications may be done, such as non-equal probabilities of gene choice.

In Figure 1.3, we can see uniform crossover of parents P_A and P_B with a prob-

ability of choosing a gene from the first parent set to 0.5. This example creates only one child C.

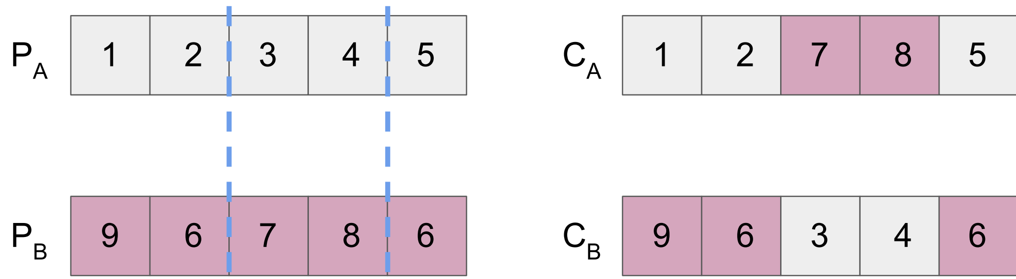


Figure 1.2: Two-point crossover with two children.

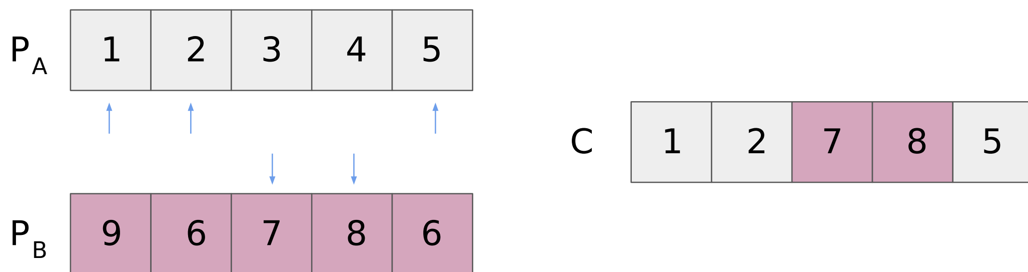


Figure 1.3: Uniform crossover with one children and probability 0.5.

Arithmetic crossover uses for the gene combination arithmetic operations. For example, children c may be obtained as a weighted mean of every gene of parents $p, q : c_i = ap_i + (1 - a)q_i$ where $0 < a < 1$.

1.3.3 Mutation

Mutation [6, 3] brings another randomness into the evolutionary process. Operators randomly change some characteristic traits of an individual and this makes mutation responsible for maintaining the genetic diversity of a population. Exploration is supported and individuals are not too similar to each other. Hence, this behaviour helps the population by preventing it from being stuck in a local optimum. The mutation operator is very powerful and in some evolutionary algorithms fields, e.g. *evolutionary strategies* [7], crossover is completely suppressed at the expense of mutation.

Examples of mutation

Bit-flip mutation is a mutation specifically designed for binary individuals. Every gene is changed to another value with some small probability. Mutation of position one and three in individual P creates individual P' as shown in Figure 1.4.

Figure 1.4: Bit-flip mutation of an individual.

Other types of mutation are usually very customized for a given genome encoding. But typically, when an algorithm works with individuals composed of numbers (floats or integers), mutation designs can be divided into two categories. **Biased mutation** takes into account the original gene value when it computes the new value. A simple example is a mutation, which takes the original value and adds to it a random number from *normal distribution*: $g_{t+1} = g_t + \mathcal{N}(0, \sigma)$. On the other hand, **unbiased mutation** generates a new random gene regardless of its original value.

1.4 Genetic Algorithms

Genetic algorithms (GA) are a subclass of evolutionary algorithms and they are probably one of the simplest approaches of EA [4]. Therefore the original genetic algorithm proposal is nowadays called *simple genetic algorithm* (SGA) [8]. The SGA works with the simplest encoding and the minimal set of genetic operators. This approach was proposed by Holland in the 1960s [9].

The SGA uses the mentioned **binary encoding**. An individual's genotype is a string of k binary values. Formally for individual x: $x \in \{0, 1\}^k$. Individual's fitness is computed by some **real-valued fitness function** f mapping individuals space to the set of real numbers: $f : \{0, 1\}^k \to \mathbb{R}$.

The SGA uses **roulette-wheel selection** for selecting parents from a population. Parents are then mating via **one-point crossover** with crossover probability p_c. This probability determines if parents undergo crossover as we do not want to modify all individuals in the population. The mutation method chosen for the SGA is **bit-flip mutation** with probability p_m which decides whether a gene in the genome is flipped to the other binary value.

See Algorithm 1 for the description of the SGA.

1.5 Coevolution

During time, scientists discovers new ideas and bring upgrades to mentioned methods. A lot of improvements are inspired by nature and biology, just like the main idea of evolutionary algorithms. One of them is the principle of *coevolution* [10].

In nature, it can happen that two (or more) species reciprocally affect their evolution as they interact together. For example, bird-pollinated flowers evolved coloration distinctive to hummingbird vision. Usually, the influence of the other species affects natural selection. The same goes with coevolution in computer science, where another species' impact is implemented in selection through fitness

Algorithm 1 Simple Genetic Algorithm

Require: fitness function f, crossover probability p_c, mutation probability p_m, population size m, maximal number of generations g

```
t ← 0
randomly_initialize(Population(t))
evaluate(Population(t), f)
while isNotTerminanted() and t < g do
    Parents(t) ← roulette_wheel_selection(Population(t))
    Children(t) ← one_point_crossover(Parents(t), p_c)
    mutate(Children(t), p_m)
    evaluate(Children(t), f)
    Population(t + 1) ← Children(t)
    t ← t + 1
end while
```

computation [11]. The rest of the evolution process runs independently on other species.

Coevolution is widely used for complex tasks, which is better to decompose into easier subtasks. Then populations of solutions for every subtask are treated and simultaneously evolved as different species. An example, which will be discussed later in section 3.4, is coevolution of structures of convolutional neural networks - *blueprints*, which defines an architecture of a neural network, and *modules* representing convolutional or fully connected layer.

Cooperative vs. Competitive Coevolution

Depending on their fitness computation, coevolution methods can be divided into *cooperative coevolution* [12] and *competitive coevolution* [13]. The cooperative mode evaluates an individual by concatenating it with (usually) the best individual from other species in some defined way. At the end of evolution, the fittest individual with its collaborators from other species gives the best solution to the problem.

Competitive coevolution evolves competing species. The competitive fitness is based on direct competition among individuals selected from the independently evolving populations. For example, $fitness_a = 1 - fitness_b$, for $fitness \in (0, 1)$. In other words, when one is winning, the other one is losing and otherwise. The typical case of competitive coevolution is coevolution of predator and prey, where the fitness of the predator can be computed as the time to catch t_c and fitness of the prey as $1 - t_\mathrm{c}$.

Species

Another attempt of scientists to apply biologically inspired phenomena is an implementation of individual *species* [14]. This mechanism fights against selection, which usually leads to a loss of diversity in a population. Selection of candidates for a mating pool is heavily dependent on the fitness of individuals. To decrease this dependency, it is possible to divide the population into species according to how individuals are similar to each other.

To preserve genome diversity, genetic operators work only with individuals from given species. The size of the species population is the bigger the better performance individuals give, in practice. So individuals of better species have a bigger probability to survive into the next generation.

The difference between species and coevolution is that individuals in all species use the same fitness function. The fitness function does not evaluate concatenated individuals from all species as it does in coevolution.

Species in a population allows us to add improving principles and methods to evolution. A well-known mechanism is called *fitness sharing* [15]. Fitness sharing is inspired by the idea, that individuals of the same species have to share resources (e.g. food), therefore the resource is the fitness value in the case of evolutionary algorithms. Typically, the shared fitness f_i' of an individual i is calculated as:

$$f_i' = \frac{f_i}{\sum_{j=1}^{N} sh(d_{ij})}, \tag{1.2}$$

where f_i is the fitness value of the individual i, sh is a *sharing function* measuring the similarity level of two individuals, d_{ij} is the distance between individuals i, j and N is the number of individuals in the given species. Individuals in densely populated regions are given a lower fitness value than comparably good solutions in sparsely populated regions. In effect, sharing tends to encourage the search in unexplored areas.

1.6 Multi-objective EA

Plenty of problems do not settle for solutions optimized for only one objective. An example from the real world is buying a PC. The cost, computer performance, or the type of GPU, etc. are decisive properties for a buyer and they typically want to find the compromise of all important features. Multi-objective evolutionary algorithms (MOEAs) were designed to solve this type of problems [16, 17].

In some cases, a sufficient solution is to compute fitness function $f(x)$ as a weighted sum of objectives $o_1, ..., o_n$ (or use other scalarization methods) $f(x) = \sum a_i o_i$, where $a_1, ..., a_n$ are defined weights and this way we are back to single-objective optimization.

From now on, when we talk about multi-objective optimization, we always mean minimization.

1.6.1 Pareto-based Methods

A common approach is to search for *Pareto-optimal* solutions [18]. The set of these Pareto-optimal solutions is called *Pareto front* and for each solution, it is true that it is *not dominated*[1] by any other solution in the search space. No Pareto solution can decrease any objective without increasing other objectives. Since the set of Pareto-optimal solutions is not finite, we try to approximate it.

[1] A solution S dominates a solution S' if S is in all criteria at least as good as S' and at least in one criterion S is clearly better.

We will take a closer look at how these algorithms work with a popular method in the next subsection.

A visualization of the Pareto front in the search space of solutions with objectives f_1 and f_2 is shown in Figure 1.5.

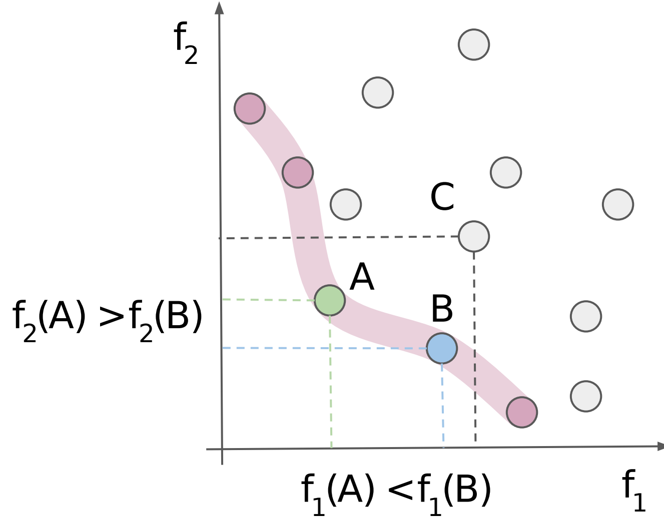


Figure 1.5: The visualization of Pareto front.

1.6.2 NSGA-II

NSGA stands for *non-dominated sorting genetic algorithm* [19]. It is one of the simplest examples of multi-objective evolutionary algorithms and for its popularity NSGA has ensured its further development and improvements (NSGA-II [20], NSGA-III [21, 22]).

Implementation

NSGA-II implements multi-objectiveness mainly in environmental selection.
When it comes to selecting individuals, who will be transferred into the next generation, parents and offspring populations are joined together creating one population (elitism is done in this part). This large merged population is then divided into *non-dominated fronts*. The first front is a set of all non-dominated individuals from the population, the second front is a set of all non-dominated individuals from the population without the first front, etc. Next-generation individuals are taken from the fronts with the lowest number.

When we want to select a number of individuals smaller than the size of the front with the lowest number, we need to decide which individuals will be transferred and which will die. This decision is left to *secondary sorting criterion* and the original NSGA-II uses *crowding distance* (another example of a secondary criterion is *hypervolume contribution*). Crowding distance is essentially based on a distance between the two closest neighbours of the individual. The bigger distance the better because it means that such a solution is in a sparsely populated area and improves the coverage of the Pareto front. For this reason, solutions at the ends of the front get infinite distance which secures their place in the selection.

So, when comparing two individuals, first a *rank* of the individual (number of the front to which it belongs) is considered (smaller is better). In the case of the same rank, their crowding distance is considered (bigger is better). No fitness is computed.

The steps of NSGA-II can be summarized as follows:

1. Create a new population by merging the parent population and the offspring population.
2. Apply non-dominated sorting algorithm 2 on the population, which creates non-dominated fronts.
3. Generate a next-generation population and fill it with individuals from fronts with the lowest number until the required size is reached. If needed, use crowding distance calculation for selection from the last used front.
4. Generate new offspring from the next generation population.

Algorithm 2 Fast Non-dominated Sorting Algorithm

```
Require: a population P
  for each p ∈ P do
      for each q ∈ P do
          if p dominates q then
              S_p = S_p ∪ q
          else if q dominates p then
              n_p = n_p + 1
          end if
          if n_p = 0 then
              Front_1 = Front_1 ∪ {p}
          end if
      end for
  end for
  i = 1
  while Front_i ≠ ∅ do
      T = ∅
      for each p ∈ Front_i do
          for each q ∈ S_p do
              n_q = n_q − 1
              if  thenn_q = 0
                  T = T ∪ {q}
              end if
          end for
          i = i + 1
          Front_i = T
      end for
  end while
```

2. Neural Networks

Similarly to evolutionary algorithms, the idea leading to creation of neural networks (NN) comes from biological inspiration. The first model was proposed by Rosenblatt in [23]. Even though neural networks differ from today's findings in neurology, their theoretical background is very strong. They are still popular and useful for solving different tasks from the computer world, like time series prediction, image classification, etc.

2.1 Biological inspiration

A neural network imitates the behaviour and structure of an animal brain. Simply said, the natural nervous system is built from neurons, electrically excitable cells, connected by synapses with other neurons. Biological neurons communicate by sending electric signals via these synapses. Artificial neural networks are made from *neurons* too, but artificial neurons are processing units. They are also linked by synapses, which transmit a signal from one neuron to another. As information flows through the whole network, every time it visits a new neuron, it is changed a bit. This results in the transformation of the initial information into the solution to the given problem, ideally.

2.2 Perceptron

In this chapter, we will show how a neuron works and computes on an easy example - a simple model containing only one neuron called *perceptron*, also known as *McCulloh-Pitts neuron* [24].

An essential characteristic of every neuron is its *weights*. Weights are a vector of real numbers, which are used for the transformation of retrieved signals. The developer's goal is to train the neural network to give plausible solutions for the given problem. It is done by *learning*, the process of iterative changing weights, which is usually based on some version of *gradient descent algorithm* [25].

An *activation function* is a non-linear function that maps a weighted summary of retrieved signal into defined space, i.e. binary values. It is a NN defining parameter, which always needs to be defined.

Network Calculation

Neuron computation can be formally described as follows: for a given input vector $x = (x_1, x_2, ..., x_n)$, a neuron constitutes a function

$$y = \phi(\sum_{i=1}^{n} w_i x_i + b), \tag{2.1}$$

where $w = (w_1, w_2, ..., w_n)$ is a neuron weight vector, b is a neuron *bias* and ϕ is an activation function.

Perceptron aims to perform binary classification. For this purpose, the best choice for the activation function is a function defined as:

$$\phi(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

A visual interpretation of a perceptron computation is in Figure 2.1.

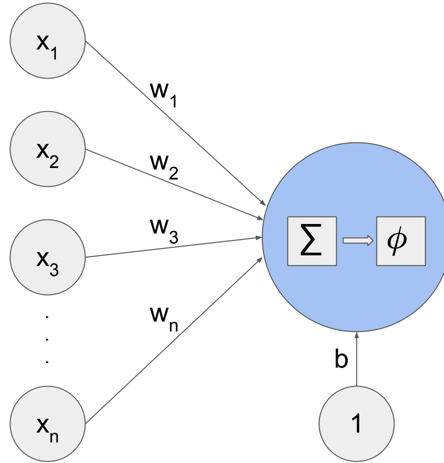


Figure 2.1: The scheme of a perceptron.

2.3 Feedforward Neural Networks

One neuron alone is not a very effective model for solving complex problems, due to its non-ability to learn patterns that are not linearly separable. It is better to use neurons as elementary units of some network. *Multilayer perceptron* (MLP) is a model built from at least three *layers* of neurons [26]. Those layers are called *input, hidden* and *output layers.* Thanks to the input layer, which transforms input data into a space, where it becomes linearly separable, MLP can distinguish non-linearly separable data.

Feedforward neural network (FNN) is a class of neural networks, to which MLP belongs [27]. They are called *feedforward* because of their architecture - neural connections do not form any kind of cycle. This means input data pass only in one direction, never backward. Generally, any acyclic graph can be viewed as a feedforward network.

Network Calculation

Since FNN contains more than one layer, networks compute their output a bit differently. Neural weights compose matrices W which represent the weights between consecutive layers. For every weight matrix, its size is defined as *a size of the next layer* $\times$ *a size of the current layer.* FNN then computes the output of every network, from the first hidden layer to the output layer, as follows:

$$y = \phi(Wx + b), \tag{2.3}$$

where $b \in \mathbb{R}^{|next\ layer|}$ is a vector of biases and $x \in \mathbb{R}^{|current\ layer|}$ is a vector of previous layer results. ϕ is a non-linear activation function, which can be defined

differently for every layer in FNN. Commonly used functions are sigmoid, softmax, or ReLU (for closer description and other functions look in [28]).

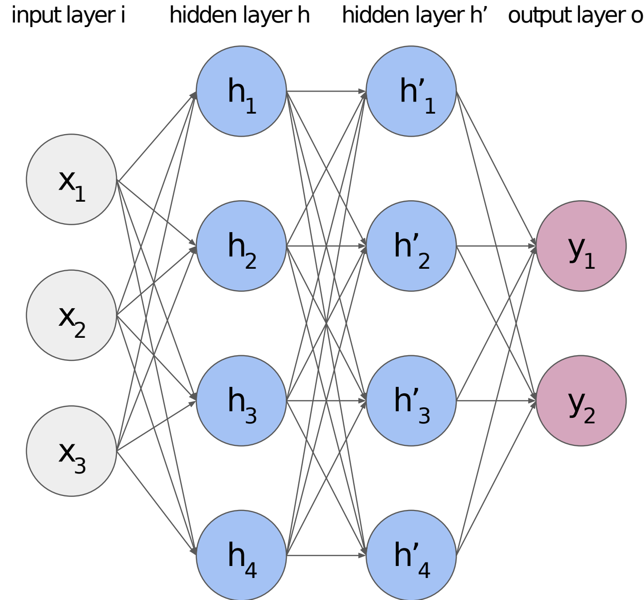


Figure 2.2: The visualization of a feedforward neural network with two hidden layers.

An example of a FNN with an input layer i, two hidden layers h, h', and an output layer o is shown in Figure 2.2. We will denote weight matrices as $W_{h,i}$ for weights from the input layer to the first hidden layer, similarly, we get $W_{h',h}$, $W_{o,h'}$. Biases are $b_h, b_{h'}, b_o$ and activation functions $\phi_h, \phi_{h'}, \phi_o$.

We then can compute the final network output vector y for input vector x as:

$$h = \phi_h(W_{h,i}x + b_h), \quad (2.4)$$

$$h' = \phi_{h'}(W_{h',h}h + b_{h'}), \quad (2.5)$$

$$y = \phi_o(W_{o,h'}h' + b_o), \quad (2.6)$$

where h and h' are vectors of intermediate results from corresponding hidden layers.

2.4 Convolutional Neural Networks

In this section, we will focus more on neural network architecture called *convolutional neural networks* (CNN) as evolving CNNs is crucial for this master thesis [29]. The first modern CNN was introduced in LeCun's paper [30] in 1998. This architecture gained its popularity thanks to its ability to effectively solve image-related tasks (e.g. image recognition), which use image pixels as input data. CNN contains at least one *convolutional layer* that supports mathematical operation *convolution*.

Convolution

Convolution is a mathematical operation (denoted by the symbol $*$) operating with two functions and producing one new function which shows, how much is

the first function affected by the second function. For two functions x and w is convolution $x * w$ defined as:

$$(x * w)(t) = \int x(t-a)w(a) \, da. \tag{2.7}$$

Function w is called *kernel*. If we use convolution in image processing, a function x is then an input image, and function w is some kind of filter. This terminology also applies to convolution in CNNs. In a world of weight matrices, we define convolution as:

$$(K * I)_{i,j} = \sum_{m,n} I_{i-m,j-n} K_{m,n}. \tag{2.8}$$

When processing images, the input pixel is usually not only one value but a whole vector of values called *channels*. The same goes for output pixels, which are composed of *output channels*. This new data information needs to be processed by convolution too, so the kernel is actually a four-dimensional tensor.

Simply said, convolution emphasizes or removes certain features from the input image. The value of an output pixel is computed as a weighted sum of the pixel's neighbourhood in the input image. Kernel defines the weights used in the sum. This method is used, for example, in image smoothing or edge detection.

The convolutional kernel may be also referred to as *convolutional filter*. Convolution is in fact a type of *filtering*, which is an operation computing an output pixel from its original neighbourhood. When the convolutional layer filters images, the kernel extracts its visual features, like edges. For this reason, the kernel is also called *feature detector*.

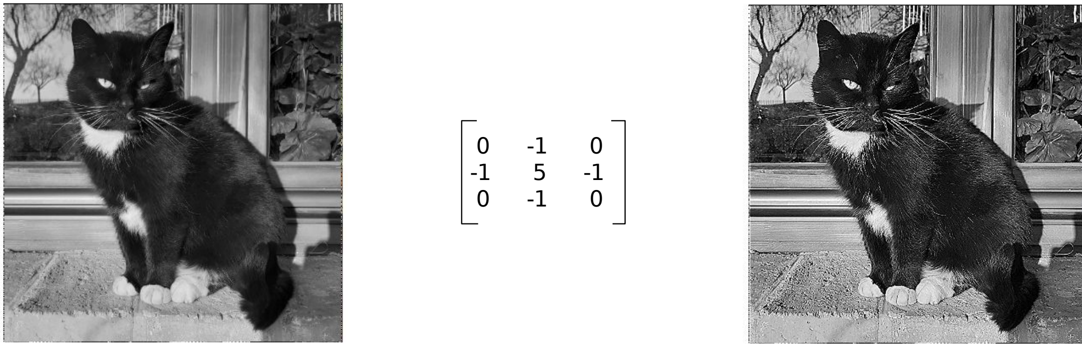


Figure 2.3: An example of image convolution with sharpening kernel of author's cat photo. Produced with a website app available on [31].

Convolutional, Dense and Pooling Layers

As mentioned before, the convolutional network contains at least one convolutional layer. The design of this layer requires a definition of the following parameters: *width W* and *height H* of the kernel, *number of input, output channels C, O*. Then we get the kernel of size $W \times H \times C \times O$.

The output size can be also controlled by two other parameters. *The stride* defines what (in what position) the input pixel should be used in calculating the output pixel. For example, the output is half the size, if the stride is 2.

The padding scheme acquires two values: valid or same. *Valid padding* uses only valid (existing) input pixels, which causes the result to be smaller. *Same padding* keeps the size of the result the same as the size of the input, but it uses zero pixels for evaluation of edge pixels.

In Figure 2.4, an example of convolution with a filter of size 2 × 2 is shown. Valid padding and stride 1 cause that the size of the output matrix is smaller than the original matrix.

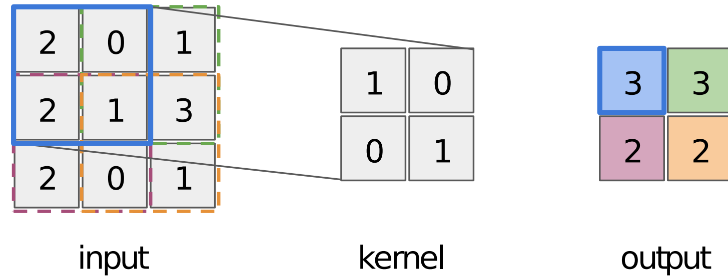


Figure 2.4: An example of pixel matrix convolution with a kernel of size 2 × 2, valid padding and stride 1.

Dense layer also known as *fully-connected layer* connects every neuron to every neuron in the next layer. Dense layers are usually used for the classification/regression part in CNNs, therefore they are used as the ending layers in the architecture.

CNN contains not only convolutional layers and fully-connected layers. Another component is so called *pooling layer*. The pooling layer reduces the spatial size of intermediate results and decreases the number of the network's parameters. The pooling process divides the image into the same size regions. Then it performs some operations on every region producing only one value. *Max pooling* computes the output as maximum value of the region, *average pooling* computes average value of the region [32].

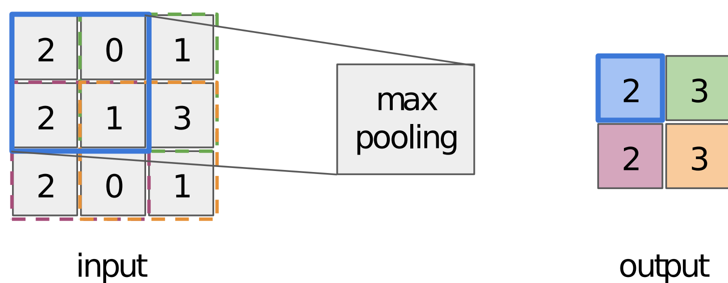


Figure 2.5: Max pooling with neighbourhood of size 2 × 2.

3. Neural Architecture Search

Despite neural networks being widely used for finding solutions to computer science problems, there is still one big part of work with neural networks scientists struggle with. Designing an appropriate neural network is still a very challenging task. NN architecture describes what number of layers and neurons a network contains, how are these neurons connected, what activation function is used, etc. There is plenty of parameters designers have to deal with, but luckily, weights do not belong to them because their values are found during the training process later.

Historically the first approach is manual architecture design by hand. This approach requires a designer's deep knowledge, creativity and time. As much as it may seem like looking for a needle in a haystack, this method has yielded many useful principles and architectures that have come into widespread use. For example *residual connections* [33], a model called *SqueezeNet* [34] and previously mentioned *convolutional layers* [29].

As time passed, the idea of automatic architecture search become more realistic. This technique is called *neural architecture search* (NAS) [35, 36]. It is a rapidly evolving field, nowadays, and the very successful fruit of this method is e.g. the *EfficientNets* family of architectures [37].

Neural architecture search process is illustrated in Figure 3.1. A search strategy selects an architecture A from a predefined search space $\mathcal{A}$. Then a performance estimation strategy evaluates proposed architecture A and the result is given back to the search strategy.

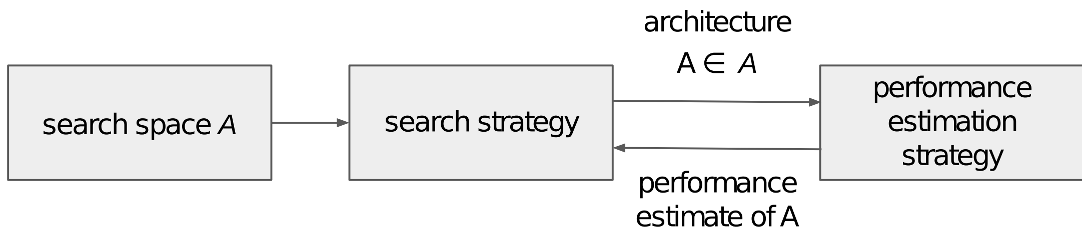


Figure 3.1: The abstract illustration of neural architecture search.

3.1 NAS Components

Search Space

The search space defines the type of NNs we want to find and use later. Since this space can be enormous and the search could be infinite, the search space needs to be limited. Limits can be easily set by creating task-specific space, for example, the space of convolutional networks for searching solutions for image recognition. Or we can be more concrete and use search space based on state-of-the-art architectures. This approach was used by authors of NAS-Bench-101 [38], who used search space inspired by ResNet [33] and Inception [39].

To sum up, this part of the NAS workflow is still heavily influenced by scientists, who define and set space limits. If they create a very specific and small

search space, the rest of the NAS parts can be less time-consuming, but also the probability of finding novelty solutions decreases.

White et al. [35] divides popular search spaces into several types.

1. **Macro search spaces** can be of two types. The first type contains whole neural architectures represented as a single directed acyclic graph (this approach was very popular with the first NAS attempts). The second type of macro search spaces focus only on macro-level hyperparameters.

2. **Chain-structured search spaces** are made of a simple sequential chain of operation layers.

3. **Cell-based search spaces** work with knowledge gained from manual design: in some state-of-the-art networks, some blocks of layers are repeated multiple times. So cell-based space contains only these blocks called cells. Cells are then stacked and together create the final network.

4. **Hierarchical search spaces** consist of multiple searchable levels of motifs, where the motif is often a directed acyclic graph of lower-level motifs.

Search Strategy

The way we should explore the search space to find the optimal solution is defined by a chosen *search strategy*. Search strategies can be naturally divided into two categories: black-box optimization techniques and one-shot techniques. The difference between these two categories will be shown in specific methods, which gained their popularity over the years of NAS research.

Random search is usually a first attempt to search for any solution. Even though it is generally not very effective, the simplicity of the idea and implementation are tempting - architectures are selected randomly from the search space. Selected networks are then trained and the one with the best accuracy is the result.

Reinforcement learning (RL) uses a neural network[1], called *controller*, to find architectures in the search space as its action [41]. The controller is then rewarded based on the found network validation accuracy. Finally, the controller updates its searching strategy (NN parameters) to maximize its reward and is ready to look for another architecture. See general RL NAS Algorithm 3.

A popular search strategy based on reinforcement learning is *Efficient Neural Architecture Search* (ENAS) described in Section 3.2, for example. A closer description of reinforcement learning based NAS and other possible approaches to it can be found in this survey [42].

Evolutionary algorithms were historically the first approach to neural architecture search, but the usage has changed since then [43]. They used to simultaneously optimize both the neural architecture and the weights i.e. they performed *neuroevolution* [44]. Today, EAs are rather used only for the optimization of architectures that are then trained with SGD-based methods.

Evolutionary algorithms were closely described in Chapter 1. Evolutionary NAS algorithms work with neural architectures as individuals. In every generation, a group of parent architectures is sampled from a population using a selection

[1] Usually a *recurrent neural network* [40].

Algorithm 3 General Reinforcement Learning NAS Algorithm

Randomly initialize controller weights θ.
for $t = 1, ..., T$ **do**
  Use the controller policy $\pi(a, \theta)$ to find an architecture a.
  train(a)
  evaluate(a)
  Update controller's parameters θ by performing a gradient update.
end for

operator according to their fitness. The fitness is usually computed as network accuracy or error rate after the network was trained. Parents are then combined and mutated as crossover and mutation rules say. This results in the birth of new and ideally better neural architectures.

Evolutionary NAS algorithms are very variable. Scientists can choose from a wide range of existing genetic operators and individual encodings or they can simply use tailored procedures. Also, the possibility of implementing other evolution principles such as elitism, and coevolution, contributes to the diversity of evolutionary-based methods.

Other black-box optimization techniques include **bayesian optimization** methods [45] or **Monte Carlo tree search** [46]. All of the black-box methods rely on iterative sampling of architectures from the search space, training them, and updating the search strategy based on the network's success. But thousands of networks can be trained, in practice, so the time complexity becomes a big issue here.

Simple description of *one-shot* NAS techniques follows [47]. Its creation was inspired by avoiding training each architecture from scratch. The backbone of these techniques is to find only one large neural *supernetwork* or *hypernetwork*[2]. After "one-shot" training of the supernetwork, the search space is created from subnetworks, which are not evaluated again, because they inherited their weights from corresponding parts of the supernetwork.

Performance Estimation

As mentioned before, most search strategies involve performance estimation based on training and evaluating found neural networks, which can significantly increase the computational cost. One option is to reduce training demands like a lower number of train epochs or subsampling of the input dataset.

Another possibility is to use speedup techniques for NAS algorithms. **Performance prediction** is a technique based on predicting the performance of the NN before it is fully trained. The prediction is handled by any function called *performance predictor*. **Multi-fidelity algorithms** try to approximate the objective function by a lower-fidelity and cheaper version parameterized by the *fidelity parameter*. **Weight inheritance** fights against discarding gained knowledge of already trained networks and saves some time. Weights of trained architectures are transferred to similar newly found networks.

[2]These are not two different names for the same thing. A hypernetwork is NN which generates the weights of other neural networks.

Multi-objective Search

Until now, we introduced methods that are based on searching only for the most accurate architectures by maximizing accuracy or minimizing error rate. But every neural network has other features which can be also optimized. Focusing on minimization of a size of a network or a number of trainable parameters can lead to desired time-consumption decrease because of faster training. Other interesting objectives are for example memory consumption and inference time. These features are not powerful enough to use as the only objective as we still want to reach a capable network. One approach called *multi-objective search* aims for all chosen objectives in one searching process.

3.2 ENAS

The Efficient Neural Architecture Search (ENAS) is an approach based on reinforcement learning search strategy [48]. It uses a *long short-term memory* controller for searching for neural network architectures. When the controller finds a new *child model*, the model is then trained to minimize cross-entropy loss and evaluated. The controller uses the measured performance as a guide for finding better models. The controller itself is trained with a policy gradient to select a NN maximizing the expected reward. This process is repeated for many iterations.

The ENAS uses a unique representation of search space, which they view as a single *directed acyclic graph* (DAG). The nodes of DAG are local computations and edges represent the flow of information. The found child models are then presented as subgraphs of this larger graph. An example of ENAS DAG search space is shown in Figure 3.2.

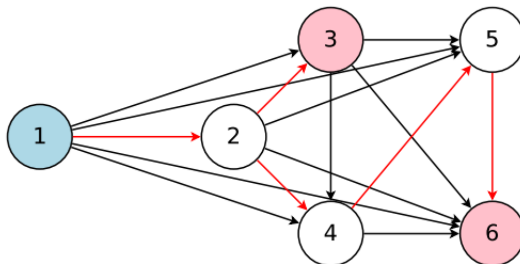


Figure 3.2: The visualization of the directed acyclic graph search space. The red connections define a found child model, which has input in node 1, and nodes 3 and 6 are output nodes.

The DAG search space allows implementing the main contribution of ENAS *parameter (weights) sharing* between found child models, which makes ENAS performance faster. So ENAS works with two learnable parameters: the parameters of the controller and the shared parameters of the child models.

In the paper, authors propose ENAS for convolutional neural networks and recurrent neural networks. Both types require different approaches to designing its elementary cells, but the common base process can be simply described as: the controller samples a subgraph from the DAG and its parameters, such as operations used in the nodes and connections.

3.3 LEMONADE

The Lamarckian Evolutionary Algorithm for Multi-Objective Neural Architecture Design (LEMONADE) is an evolutionary algorithm for multi-objective architecture search with Lamarckian inheritance mechanism [49]. The LEMONADE evolves neural networks optimizing two objectives, one objective maximizes network performance and one objective penalizes resource consumption.

The evolution works with the population of neural network encodings. The sampling process is divided into two stages. In the first stage, the parents are sampled from the population based on their *cheap* objective values such as model size. After sampling, the creation of children using special network operators happens (the operators are described in the next paragraph). In a second sampling stage, the second round of the selection chooses a subset of children based again on the cheap objectives. This subset of child networks is then evaluated on some expensive objectives. By this two-staged sampling strategy, the LEMONADE generates and evaluates more children that have the potential to fill gaps in the current Pareto front approximation. The next generation is a new Pareto front computed from the current population and generated children.

The LEMONADE network operators are used in the evolution as the mutation operators. The operators can be divided into two classes, namely *network morphism* and *approximate network morphism*. The network morphism is an operator, which enlarges NN but preserves the network performance (the network remains the same). The LEMONADE network morphism operators are:

- insert an identity block, which does not change the network performance,
- increase the number of convolution filters,
- add the skip connection.

As was said, all network morphisms increase the network capacity. But during multi-objective optimization, operators, which can decrease the size of NN architecture are needed. The proposed operators with this ability are grouped into a class called approximate network morphism. The LEMONADE specifically uses:

- remove a layer or a skip connection,
- prune a convolutional layer,
- substitute convolution by a depthwise separable convolution.

All modified objects are chosen randomly.

3.4 CoDeepNEAT

The Coevolution Deep NeuroEvolution of Augmemting Topologies (CoDeepNEAT or CDN) is a method from a class of evolutionary neural architecture search algorithms introduced by Miikkulainen et al. in 2018 [50].

3.4.1 Development

The CoDeepNEAT is a descendant of a different NAS method - *NeuroEvolution of Augmenting Topologies* (NEAT), which carries the core idea of all NEAT-based methods [51]. NEAT is a genetic algorithm and it is based on some new principles: usage of historical markers and complexifying. As the name says, NEAT is a kind of neuroevolution, so it evolves both weights and architecture.

NEAT

NEAT uses direct encoding. A neural network is represented as a simple list of connection genes (edges). Each connection gene specifies its input node, an output node, *an enable bit* (whether the edge is enabled or disabled), and *an innovation number* crucial for properly working genetic operators. These numbers mark the original ancestor of each gene. See Figure 3.3, where the third gene is disabled, so the connection that it specifies (between nodes 2 and 5) is not expressed in the phenotype.

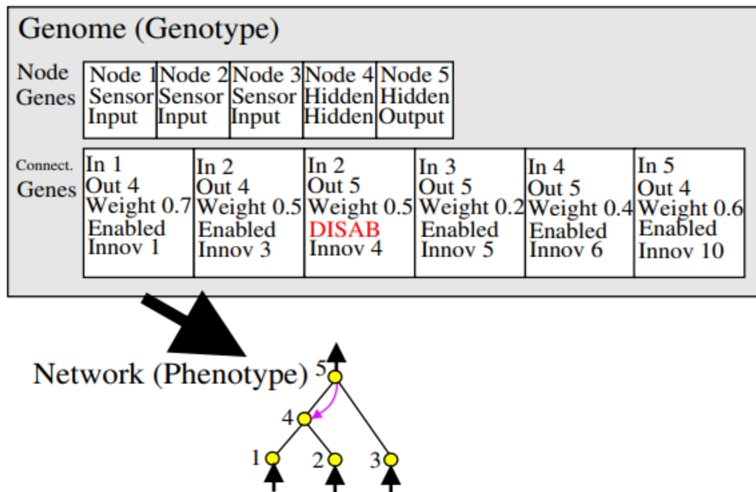


Figure 3.3: A genotype to phenotype mapping example. Source: [51].

The mutation can change any network in multiple ways: weight mutation, adding a node, adding a connection. Adding node or connection and its influence on both genotype and phenotype is shown in Figure 3.4. The top number in every connection gene is the innovation number. New genes are assigned new increasingly higher numbers.

In order to perform crossover which does not create a nonsensical topology, the innovation numbers are used. During crossover, the connections in both parents with the same innovation numbers are lined up. Other genes, which have different innovation numbers, are inherited from the parent with better fitness value. The process of crossover is visualized in Figure 3.5.

To protect novelties with low fitness between more optimized networks, NEAT divides the population into species. Individuals then compete only with individuals of the same species. The speciation is based on topology similarity.

When NEAT was firstly introduced, it performed a faster search than any other neuroevolution technique. These results led to the enrichment NEAT-family with new algorithms such as *HyperNEAT* [52], *DeepNEAT* or *CoDeepNEAT* [50].

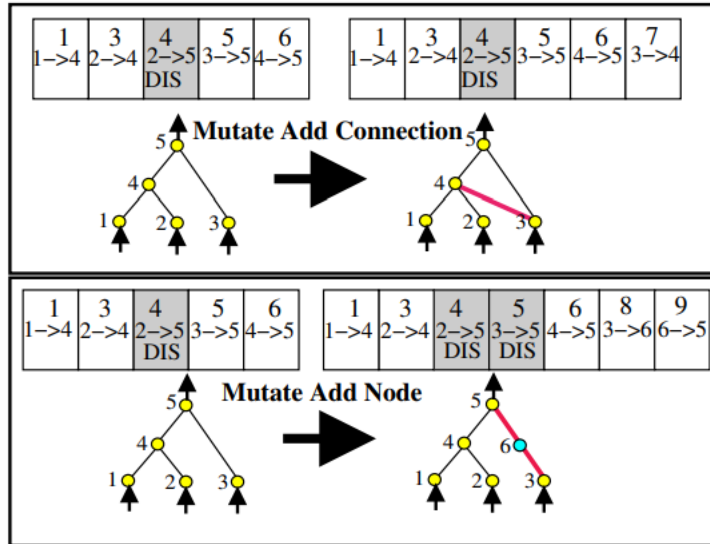


Figure 3.4: The two types of structural mutation in NEAT. Source: [51].

From NEAT to DeepNEAT

Previously described NEAT's strong point is evolution of small neural networks, but it is not as practical for deep neural networks. Some changes were required to adapt the NEAT core idea to evolution of bigger and more complex architectures. This attempt results in designing DeepNEAT, which evolves topologies and hyperparameters of DNNs.

The DeepNEAT backbone is basically the same as NEAT's, species and innovation numbers (historical markings) are used as well, but the major difference is in genome encoding. Instead of representing a network as a list of connections between neurons, DeepNEAT works with lists of connections between *layers* of NN. Each layer is defined by hyperparameters such as type of layer (e. g. convolutional layer, fully-connected layer), number of neurons, and activation function.

Since DeepNEAT does not perform weight evolution, every solution needs to be trained to compute its fitness. Also, the algorithm has to deal with possibly different input and output sizes of connected layers. Implementing downsampling or adding a merge layer etc. solve this problem.

3.4.2 Principles of CoDeepNEAT

In this subsection, the main ideas of Coevolution DeepNEAT algorithm are described. The word *coevolution* in CoDeepNEAT refers to defined algorithm being in fact coevolutionary algorithm evolving two populations: the population of *blueprints* and the population of *modules*. The motivation for cooperative coevolution comes from phenomena appearing in successful deep neural networks - some architectures are created from modules, which are repeated multiple times.

Populations and Evaluation

Modules are small structures of connected layers composed from at least one layer. They are always divided into species as module speciesism is crucial for blueprints. *Blueprints* are backbones of evolved neural networks. They are graphs of nodes,

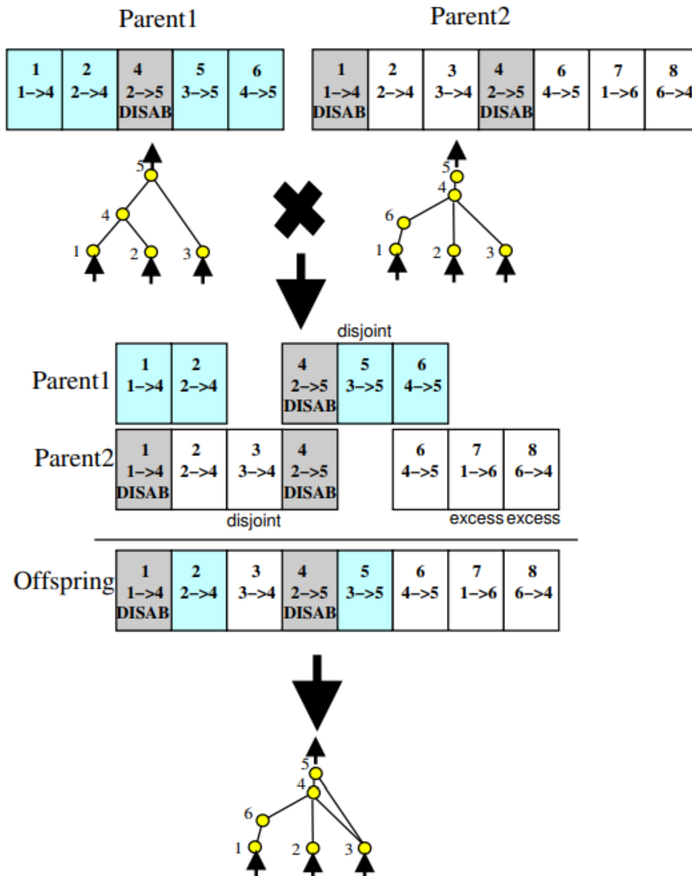


Figure 3.5: NEAT crossover of two networks. Source: [51].

which are actually pointers to module species. Initialization of these populations is very simple, as it is also in NEAT. All modules are initialized with random values and assigned to a single species. Every initialized blueprint has a minimal graph. Initializing values with defined hyperparameters (module/blueprint size, optimizers, dropout, etc.) is also possible.

These two populations evolve separately. Before the application of genetic operators (selection, crossover, mutation), fitness, which depends on the other population, needs to be calculated. For this reason, modules and blueprints are combined in a way, that every pointer node in the blueprint is replaced with a random module of the corresponding specie. If the blueprint contains multiple nodes, which point to the same module specie, they are replaced with the same module. This leads to the creation of a bigger working neural network. The assembled network is then trained, and evaluated and its fitness is passed to its original blueprint and modules. After evaluation of all assembled networks, the final blueprint and module fitness are computed as the average fitness of all assembled networks they were part of.

See Figure 3.6 for visualization of network assembly. Nodes in the blueprint B point to module species 1 and 2. From these species, modules M and N are randomly sampled, which are then put into the blueprint in corresponding places. Finally, the result of this procedure is the assembled network A.

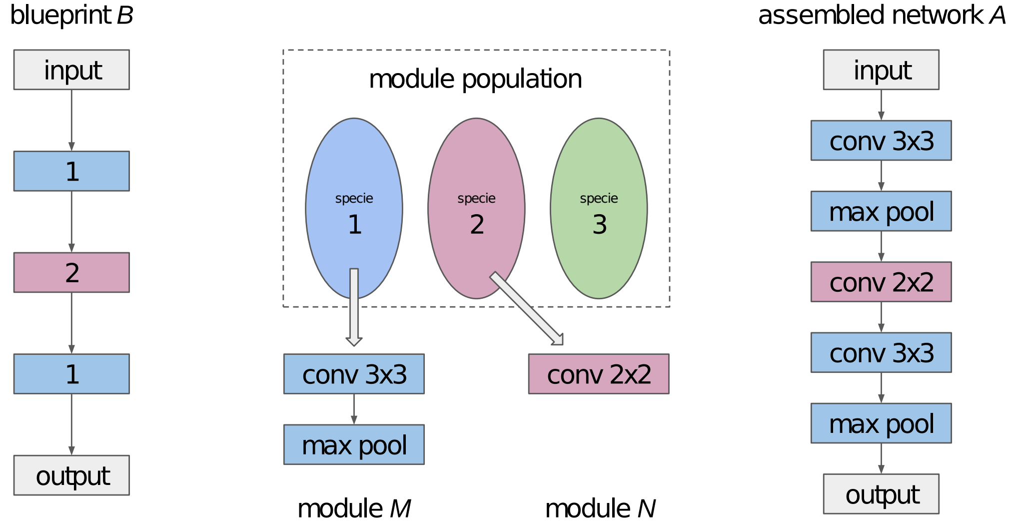


Figure 3.6: CoDeepNEAT assembly of a neural network from a blueprint and modules.

Evolution

Since both module and blueprint genomes can be represented as graphs, the same crossover and mutation methods can be used, but still, light changes need to be done. CoDeepNEAT operators are based on and heavily influenced by NEAT genetic operators.

CoDeepNEAT uses uniform crossover over the graph nodes with a fixed probability. For modules, it means their definition (parameters) are combined into resulting offspring. So in practice, a mutation process needs to be implemented for every module type, because each of them may have unique parameters. Crossover in blueprints creates new offspring by merging their genomes and random decision over the same genes in both parents. Blueprint crossover is very similar to NEAT/DeepNEAT crossover.

Mutation is based on structural changes in graphs: node addition or removal, edge addition or removal, or node replacement (changing node features). Another type of mutation can be implemented such as optimizer mutation or node species mutation for blueprints.

During evolution, a defined percentage of the fittest individuals in both populations are transferred into the new generation, so CoDeepNEAT uses elitism and ensures the preservation of the best solution.

4. Proposed Approach

In this chapter, we will describe the proposed approach to multi-objective neural architecture search. The approach is based on some of the related work and processes introduced in previous chapters.

We chose to study and extend evolutionary-based neural architecture search, namely the CoDeepNEAT (Section 3.4). At first, we tried to speed up the CDN by implementing a weight inheritance between neural networks, which was inspired by Lamarck's evolution theory commonly used in evolutionary algorithms. This part of the work involved studying neural network weights and their behaviour, proposal and implementation of the weight inheritance. Multi-objective optimization in CoDeepNEAT is built on the usage of the popular algorithm NSGA-II (see Section 1.6.2, [20]). This extension lets us optimize neural networks in two chosen objectives: the error rate of the trained neural network and the computational complexity of the network. Implementing the NSGA-II into the CDN has been accompanied by a complex study of the principles of both main algorithms.

We propose an implementation of Lamarckian CoDeepNEAT called *LamaCoDeepNEAT* (LamaCDN), which is based on NN weight inheritance, and its multi-objective version using NSGA-II called *Multi-objective LamaCoDeepNEAT* (MOLamaCDN).

4.1 LamaCoDeepNEAT

LamaCoDeepNEAT is an extension of the CoDeepNEAT algorithm, which honours the law of *lamarckian evolutionary principle* introduced in 1809 [53]. Lamarckism[1] is based on the principle that parents can transmit their physical characteristics developed during their lifetime to their offspring. Compared to Darwinism in evolutionary algorithms, which transfers unchanged characteristics, Lamarckian EA pass the parent's traits as they were at the time of mating. The classic example used to explain the concept is a giraffe with an elongated neck. Over a lifetime of straining to reach high branches, the giraffe developed an elongated neck, which was then inherited by its children.

This principle can speed up the evolution of chosen traits since a child continues to evolve the trait in the state, in which the parent passed it to them. So the child does not waste time repeating the parent's progress. The LamaCoDeepNEAT transfer the learned weights of a parental neural network to an offspring neural network, which saves time during training of the new NN because the network does not start with the random weights.

Weight Transfer between Modules and Networks

Since neural networks are not taken as individuals of the population in the LamaCDN but only as objects used for fitness calculation, the weight inheritance is realized on the population of modules. At first, a part of weight inheritance process is done during composition of a neural network by combining a blueprint

[1]Named after its author - French biologist Jean Baptiste Lamarck.

and modules. The building modules pass their layer weights to the assembled network. The assembled network does not have initialized weights (by for example the *Glorot uniform initializer* or the *He normal initializer* [54]), as it does in the CoDeepNEAT, but inherited (except for the networks created from the initial populations, because these individuals have not been trained). When modules transfer their weight matrices onto the model, the weight matrices of connected modules may have been incompatible, meaning that the number of output channels of the previous module is not the same as the number of input channels of the following module. The number of input channels needs to be changed. So the weight matrix inherited from the following module is resized (the way of resizing is described in the next paragraph). See Figure 4.1 for the visualization of weight inheritance during NN assembling.

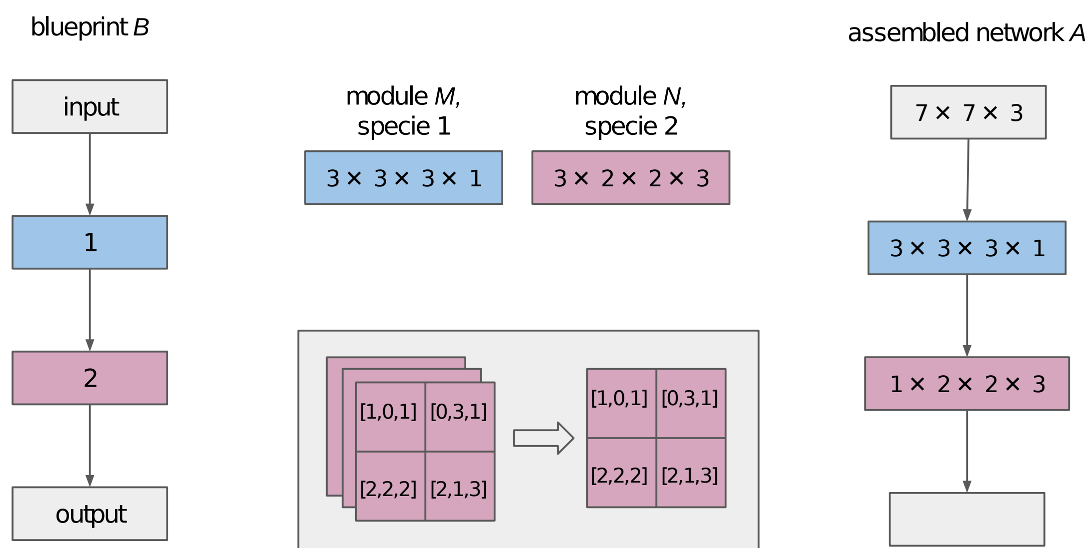


Figure 4.1: The visualization of weight inheritance during the assembling of a neural network from the blueprint B and the modules M, N. The dimensionality of modules weight matrices is shown as a number of input channels × kernel size × kernel size × number of output filters. The weight matrix of module N is shrunken in the first dimension in order to make it compatible with module M.

After the training of the assembled network during evolution, the network weights are passed on to the corresponding modules. One module can be present in multiple assembled networks. When this happens, the module inherits the weights from the network with the best fitness.

Weight Inheritance during Mating

A simple rule for weight inheritance is applied during the crossover: a new module inherits the neural weights of the fitter parent. One must be careful about the possibility of different kernel sizes and a number of filters[2], which define the resulting size of the weight matrix. Divergence can be caused by both crossover and mutation. When the offspring kernel size or the number of filters is smaller

[2] *(Output) filter* is a different name for *output channel* denoting a dimensionality of an output space. This name is widely used among the TensorFlow community.

than the parent's values, the parent's weight matrix is reduced and then passed to the offspring. Otherwise, we add new weights initialized with zero value. We have chosen to add zero values because it does not change the performance of the layer and the layer remains to compute the same function as it did before the weight matrix transformation, but the weight matrix dimension is changed as we request. A module mutation is done by changing some of the module parameter values. The weight matrix needs to be changed when the kernel size or the number of filters is mutated, which is done the same way as it is during the crossover.

4.2 Multi-objective LamaCoDeepNEAT

The Multi-objective LamaCoDeepNEAT is the LamaCDN enhanced by mechanisms supporting multi-objective optimization. The main mechanism is the NSGA-II algorithm presented in Section 1.6.2.

A version of multi-objective CoDeepNEAT has been already used in designing LEAF (Learning Evolutionary AI Framework), the evolutionary AutoML framework[3] [55]. In LEAF, multi-objective optimization is used to maximize the performance and minimize the complexity (e.x. a number of parameters) of the evolved networks simultaneously. The ranking of blueprints and modules is computed from successive Pareto fronts generated from both objectives, assembled networks are ranked similarly. Although the paper includes algorithms of the multi-objective version of the CDN and the Pareto front calculation, the mechanisms are too poorly described to be replicated without further information.

Multi-objectiveness in MOLamaCDN

In our proposal of the MOLamaCDN, non-dominated sorting is used on blueprints, modules, and assembled networks. But there is a difference between sorting blueprints/modules and assembled networks in purpose and implementation details such as secondary criterion.

Firstly, let's look at the objectives which are being optimized during the evolution. Since our goal is to decrease the time-complexity of evolved neural networks, we chose to optimize the number of floating point operations (FLOPs[4]), which is a commonly used measurement for neural network models and it makes our proposal more comparable with existing NAS methods. The FLOPs value for a given convolutional network is calculated as a sum of FLOPs of each layer. The FLOPs of the convolutional layer can be obtained as [56]:

$$\text{FLOPs} = 2fko, \tag{4.1}$$

$$o = (i - k) - 1, \tag{4.2}$$

where f is the number of convolutional filters, k is the kernel shape, o is the output size and i is the input size. But one module can be used to process input data with different sizes during one evolution iteration. For example, an assembled

[3] One of the authors of the LEAF is an author of the CDN Risto Miikkulainen.

[4] Do not confuse with floating point operations per second (FLOPS).

network contains the same module at two positions (the original blueprint has two nodes, which point to the same module specie). Therefore input size is a non-defining parameter for a module and output size, which depends on the input size, is removed from the FLOPs calculation for modules in MOLamaCDN for this reason. Then we get:

$$\text{FLOPs} = 2fk. \tag{4.3}$$

A blueprint FLOPs value is calculated as a FLOPs value of its assembled neural network, which we can obtain with implemented Tensorflow functions. Minimization of FLOPs may lead to a decrease in the size of evolved architectures and the number of trainable parameters. Another benefit of optimizing FLOPs is its undemanding and relatively supported calculation.

The other objective is the error rate of the trained network, so both objectives need to be minimized. The error rate is calculated as $1-$ sparse categorical accuracy of the model. We use sparse categorical accuracy implemented in Tensorflow [57]:

```
from . import backend as K
def sparse_categorical_accuracy(y_true, y_pred):
return K.mean(K.equal(K.max(y_true, axis=-1),
                      K.cast(K.argmax(y_pred, axis=-1),
                      K.floatx())))
```

But unlike the calculation of the number of FLOPs, calculating the error rate of the network is the most time-consuming part of the whole evolutionary NAS, especially because of the network training.

Both of the objectives are computed for every assembled network and then passed to every object, which was used for the creation of the assembled network, similarly as it is done in single-objective methods. Error rates are transferred to both blueprints and modules, but FLOPs are passed only to blueprints since the final number of FLOPs of modules is independent of assembled network and its architecture. The module's number of FLOPs is computed from its parameters, which do not change during the network assembling but only change during the mutation. So every module mutation is accompanied by the number of FLOPs update.

Transfer of Objective Values

Passing calculated objective values from the network to the modules and the blueprints is not as straightforward as it is in the single-objective CDN. We have to figure out, how to transfer a pair of values gained by an assembled network and combine it with values from other assembled networks. We have proposed and compared two methods of objectives transmission. The simpler one is inspired by the single-objective CDN - after training, the assembled network error rate and FLOPs are assigned to the blueprint and all modules used for the creation of the network. After evaluation of all networks in the current generation, each blueprint calculates its final objective values as an average of all assigned values:

$$o = \frac{\sum_i^n o_i}{n}, \tag{4.4}$$

where o is the calculated objective of the blueprint/module, n is the number of assembled networks based on the blueprint/module and o_i is the objective value of such ith network. The module calculates this way only error rate, FLOPs stays the same for the reason, which is described in the previous paragraph. The advantage of this approach is its simplicity and low time complexity. But it completely discards the information, that objective values create pairs depending on their original network, which may be useful information. Error rate and FLOPs are quite contradictory qualities and a network with a low error rate probably has a high FLOPs value (and vice versa).

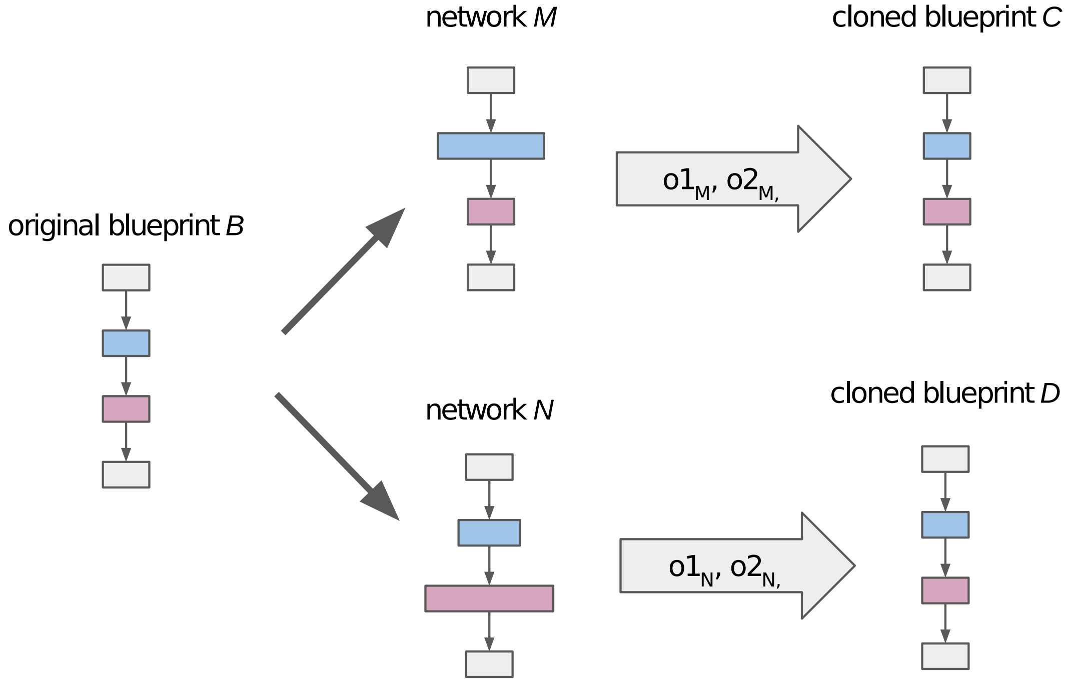


Figure 4.2: The visualization of the cloning of blueprint. The original blueprint B creates two assembled networks M, N and two clones C, D. After training, each network passes its calculated objective values $o1$ and $o2$ to the relevant blueprint copy and copies become part of the population.

The other proposed method, which is actually used in the MOLamaCDN, can be divided into two phases.

In **the cloning phase**, we create a new copy of every blueprint used in the currently evaluated network. Assign to these copies gained objectives values of the network. Put the copies in the same species as its original blueprint and for each one assign the number of siblings - how many blueprint copies of the current blueprint there are. Remove the original blueprint from the population. A simplified view of the cloning phase with blueprints is in Figure 4.2 showing creation of two copy blueprints. Similarly, create copies of all modules. Assign the error rate and trained weights and remove the original modules too. The advantage of this approach is that the weights are passed on to modules with corresponding objective values, so the module do not need to combine weights from all genomes it has appeared in and lose some information. This way, module and blueprint populations temporarily increase their size, which will be decreased back to normal in the next phase.

Algorithm 4 The Cloning Phase

Require: a population P, a set of assembled networks A

```
  Q = {}                                        ▷ A set of clones.
  R = {}                                        ▷ A set of individuals which have
                                                  been cloned.
  for each a ∈ A do
      for each p ∈ P and assemble(p, a) do      ▷ The individual p have assem-
                                                  bled the network a.
          q = copy(p)                           ▷ Copy all p attributes including
                                                  species.
          q.flops = a.flops
          if q.type is "blueprint" then
              q.error_rate = a.error_rate
          else
              assign_weights(q, a)
          end if
          Q = Q ∪ q
          R = R ∪ p
      end for
  end for
  P = (P ∪ Q) \ R                               ▷ Update P by merging it with
                                                  clones from Q and by removing
                                                  individuals from R
```

In order to create **the mating pool**, we sort the current population with clones and divide them into fronts by fast non-dominated sorting algorithm (Algorithm 2) known from NSGA-II. From the fronts, we choose a group of the best individuals, which will be later used as parents and moved straight to the new population. This way elitism is implemented in the MOLamaCDN. Every candidate needs to be checked, whether it has no other sibling in the mating pool (i.e. an individual which was cloned from the same original individual). If the group already contains one of the siblings, who belongs to a better front, the others are completely removed from the population. If both siblings belong to the same front, the algorithm chooses randomly. When it comes to the situation, where we have to choose a number of candidates smaller than the size of the front, we decide by the number of individual's siblings (which is our secondary criterion) and if compared values are the same, we look at the crowding distance (this apply to the siblings in this phase too). The usage of the number of siblings as a secondary criterion indicates our preference for the number of times an individual was selected to form an assembled network over front coverage.

During **the offspring creation**, we fill the rest of the new population with offspring of individuals from the previously selected mating pool. The mating individuals are chosen by tournament selection, which compares their rank, number of siblings, and crowding distance. Offspring are created by the crossover and mutation methods used in the original CoDeepNEAT implementation [58].

The comparison of these two methods has shown, that the first method with averaging evolved the population of networks with similar objective values. The method with cloning usually came with a wider variety of objective values,

so it is better for preserving the diversity of individuals and reaching extremes of optimized objectives. On the other hand, the implementation of cloning is more complex and it demands more memory and time.

Algorithm 5 The Mating Pool Creation

```
Require: a population P, a number of parents to select x
  M = {}                                      ▷ A mating pool.
  fronts = nondomsort(P)                      ▷ Use sorting Algorithm 2.
  for each f ∈ fronts do
      if x == 0 then
          return M
      end if
      if size(f) ≤ x then
          for each p in f do
              if siblings(p) not in M then    ▷ No p's siblings have been selected.
                  M = M ∪ p
                  x = x − 1
              end if
          end for
      else
          crowding_distance(f)                ▷ Compute crowding distance.
          sort(f)                             ▷ Sort f by number of siblings and
                                                crowding distance in decreasing or-
                                                der.
          for each p in f do
              if x == 0 then
                  return M
              end if
              if siblings(p) not in M then
                  M = M ∪ p
                  x = x − 1
              end if
          end for
      end if
  end for
```

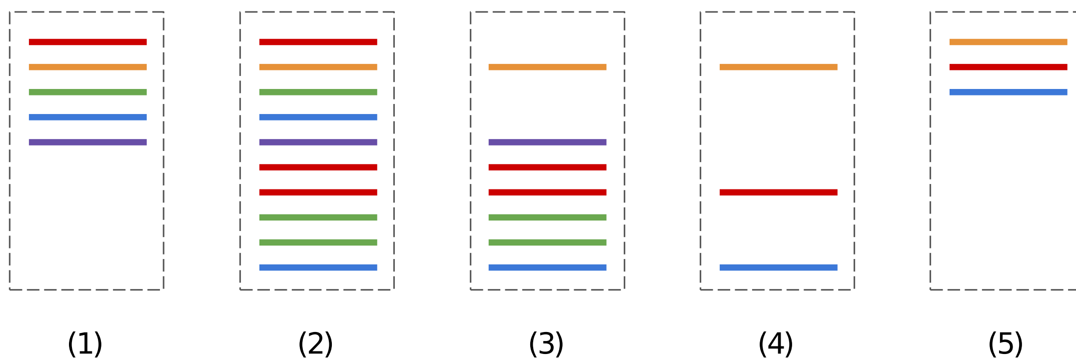


Figure 4.3: The effect of cloning on the population of individuals. Picture (1) shows a new population of five individuals (red, orange, green, blue, and purple). Picture (2) shows the cloning phase, where the red, green, and blue individuals are copied. The original individuals are then deleted in (3). From this population are then selected the best individuals, in the picture (4) those are the orange individual and the red and blue clones, the rest is removed. The final stage of the population (5) is ready for mating.

5. Experiments

In this section, we describe the experiments with LamaCDN and MOLamaCDN that we have conducted. We introduce and describe the CDN implementation and packages used for our implementation and the datasets used in the experiments. The results of experiments are then presented and compared with each other and with some other known models.

The evolutionary experiments need a lot of time to evaluate and they are usually launched in multiple runs. The final described model is usually the one of all results, which reached the best score and has the best performance. We present its objectives: fitness (i.e. error rate) and FLOPs value, and the total number of network parameters (trainable and non-trainable).

We decided to check and test our two proposed methods on two datasets - MNIST [59, 60] and CIFAR-10 [61, 62]. Also, we needed to run the implementation of the CoDeepNEAT we used, since it has no official experiment results, which we could use for comparison. The results from the original CDN paper cannot be compared either, since we did not use the original implementation as a base for our method because authors did not publish the code. MNIST experiments were run with one set of hyperparameters and CIFAR-10 was run with three sets of hyperparameters. The parameters are described in the next section. To sum up, we prepared and conducted twelve experiments in total.

We did not use any architecture dataset for limitation NAS search space like NAS-Bench-101 [63]. NAS-Bench-101 contains over 5 million trained models, which were evaluated on the CIFAR-10 dataset. Since our methods contain weight inheritance, training a network is an important part we do not want to skip.

5.1 Used Implementation

First, we needed to find an implementation of the CoDeepNEAT, because we used it as a base algorithm for our proposed approach. We applied multiple changes and expansions to transform the original CoDeepNEAT into the Lamarckian CoDeepNEAT, and later into the multi-objective version of LamaCoDeepNEAT. Unfortunately, CDN authors did not publish their paper [50] with the relevant code. Releasing code is highly recommended since without the full original code and used hyperparameters it is nearly impossible to reproduce the NAS methods. This and other recommendations to developers are mentioned in [35]. In 2020, a paper introducing a CDN implementation using Keras and its source code was published by Bohrer [64, 65]. The paper is a useful source for writing your own code since it contains many implementation details, which were not mentioned in the original paper. However, we decided not to use this code, because it is built on old methods of using the Keras and Tensorflow packages and the versions of the used packages are outdated.

The code, which we actually used as a foundation, is a part of Paul Pauls's Tensorflow-Neuroevolution framework (abb. TFNE) [66]. TFNE works with Tensorflow version 2.x and it is quite a robust and sophisticated framework. And yet,

TFNE is open to modifications, properly documented[1] and user-friendly, which makes it easier to work with it. As mentioned before, the whole code is written in Python 3.7 [67], which is a popular and supported language for machine learning model development. A significant part of the code uses Tensorflow - machine learning platform [68], especially for work with neural networks, their assembling, training, and evaluating fitness function and weight inheritance. Informations about our source code can be found in Attachement A.2.

5.2 Datasets

The chosen datasets, the MNIST dataset and the CIFAR-10 dataset, are well-known in the machine learning community. Both of them are image datasets suitable for image classification [69, 70]. Their main perks are their quality, size, and availability.

5.2.1 MNIST

The MNIST database (Modified National Institute of Standards and Technology database) [59] was first introduced in 1998 by LeCun. It is a dataset of handwritten and annotated digits. It has a training set of 60 000 examples and a test set of 10 000 examples. The dataset has been used also for tasks like image clustering or image generation so far. A few MNIST database variants have occurred during its existence, such as Sequential MNIST for sequential image classification [71] or Moving MNIST for video prediction [72]. The Moving MNIST contains short video sequences, where two digits move independently around the frame, bounce off the edges and intersect with each other.

The MNIST database does not need any preprocessing - all of its images are size-normalized and centered in a fixed-size image. Every image has a size of 28×28 pixels, the shown digit is centered in the image and it has 20×20 pixels size. The database contains black-and-white pictures only.

Figure 5.1: Sample images from the MNIST dataset, from [73].

[1]The documentation can be found on [58].

5.2.2 CIFAR-10

The CIFAR-10 dataset (Canadian Institute for Advanced Research) [61] is a dataset of photo images, which can be split into ten groups (which explains the name CIFAR-10) by displayed object. The categories are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Every class consists of 6 000 images, so the database contains 50 000 training images and 10 000 test images in total. All images are colour photographs with resolution 32×32 pixels and 3 channels (RGB).

5.3 Experiments with MNIST

The results of experiments with the MNIST dataset are presented in this section. First, the parameters of experiments are described. Descriptions and obtained results of runs of all methods (CoDeepNEAT, LamaCoDeepNEAT, and Multiobjective LamaCoDeepNEAT) follow.

Parameters

We used the same experiment hyperparameters values as the authors used in Keras-based CDN implementation [64], but we have slightly changed some of them (some of them are not mentioned in the article). The original CDN proposal [50] was not evaluated on the MNIST dataset.

All of the MNIST experiments run for 40 generations. The populations contain 10 individuals, 10 blueprints, and 30 modules. Every blueprint is used for network assembling once per generation. The networks are trained on the full training dataset (60,000 images) for 4 epochs. A maximal number of module/blueprint species is set to 5, except for the maximal number of blueprint species in MOLamaCDN, where it is set to 1, so the speciation is practically not used. The genetic operator parameters for modules are as follows: the mutation probability is 0.8, and the crossover probability is 0.2. In every generation, the two best modules are carried unchanged into the next generation. The blueprint mutation probability is set to 0.3 and the blueprint crossover probability is set to 0.1.

Other parameters used for MNIST experiments are closely described in the following tables. Table 5.1 contains parameters related to produced networks. The table says that every assembled network contains convolutional layers with one dense or flatten layer used as an output layer. Adam optimization is used for network optimization. It is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments [74]. Tables 5.2 and 5.3 show the parameters of mentioned layers. The dense layers are strictly defined with 10 units and a softmax activation function since they are used only as the output layer. The convolutional layers can be more diverse. Some of the parameters, e.g. merge method or activation function, can be randomly chosen during initialization and they can become the object of mutation during evolution. Table 5.3 also contains parameters related to dropout and max pooling layers, which may become a part of a module. We have run every method experiment 15 times.

Parameter	Type	Options
Datatype of network phenotype	Fixed	[”float32”]
Available modules	Fixed	[”Convolutional”]
Available optimizers	Fixed	[”Adam”]
Output layers	Random choice	[”Flatten”, ”Dense”]

Table 5.1: Experiment hyperparameter table.

Parameter	Type	Options
Units	Fixed	[10]
Activation function	Fixed	[”softmax”]

Table 5.2: Experiment parameter table for output dense layers.

Parameter	Type	Options
Merge method	Random choice	[”Concatenate”, ”Add”]
Filters	Fixed	[”min”: 32, ”max”: 256, ”step”: 32, ”stddev”: 32]
Kernel size	Random choice	[1, 2, 3]
Stride	Fixed	[1]
Padding	Random choice	[”valid”, ”same”]
Activation function	Random choice	[”relu”, ”elu”, ”linear”]
Kernel initializer	Fixed	[”glorot uniform”]
Bias initializer	Fixed	[”zeros”]
Max pooling flag	Fixed	[0.5]
Max pooling size	Fixed	[2]
Dropout flag	Fixed	[0.5]
Dropout size	Fixed	[”min”: 0.1, ”max”: 0.7, ”step”: 0.1, ”stddev”: 0.2]

Table 5.3: Experiment parameter table for convolutional layers.

CoDeepNEAT and LamaCoDeepNEAT on MNIST

The best network which CoDeepNEAT experiments returned has an error rate of 1.18%. After the final training for 100 epochs, the network has reached 99.01% accuracy (i.e. 0.99% error rate). This network has been found in the 38$^{\text{th}}$ generation. Another interesting network property is the FLOPs value 41,512,062 and the number of all (trainable and non-trainable) parameters 381,898.

With LamaCoDeepNEAT, we have obtained a network with an error rate of 1.72% and an error rate of 1.56% after training for 100 epochs (i.e. 98.44% accuracy). The best network has been created in the 30$^{\text{th}}$ generation. The FLOPs value is 2,195,230 and the number of all parameters is 441,290.

Since our experiments run on different computers with different CPUs, etc., it is not possible to prove the decrease in training time caused by the weight inheritance by a simple comparison of the training times of the networks. Also,

the best networks have a different size (the number of parameters), which affect the training time too, so we need to choose a different comparative method. In this article [75], authors compared an evolutionary algorithm evolving neural networks without a crossover operator (mutation operator only) with its Lamarckian version. They studied and compared test accuracies of the best networks at some points of evolution. They came to the conclusion, that the fitness convergence speed was improved by implementing weight inheritance, which sometimes made it possible to reduce the number of generations.

A plot of the fitness evaluation of the experiment with the best result can be found in Figure 5.2 for both methods. Blue points represent all assembled networks (10 networks per generation) and their reached fitness value in a generation. Red point is the best network with the lowest fitness value in the current generation. To sum up, we have found a network using Lamarckian CoDeepNEAT with weight inheritance to the MNIST dataset, which is comparable to the CoDeepNEAT solution. The process of experiments is similar but the LamaCDN experiment converged to the best solution a bit faster than the CDN experiment.

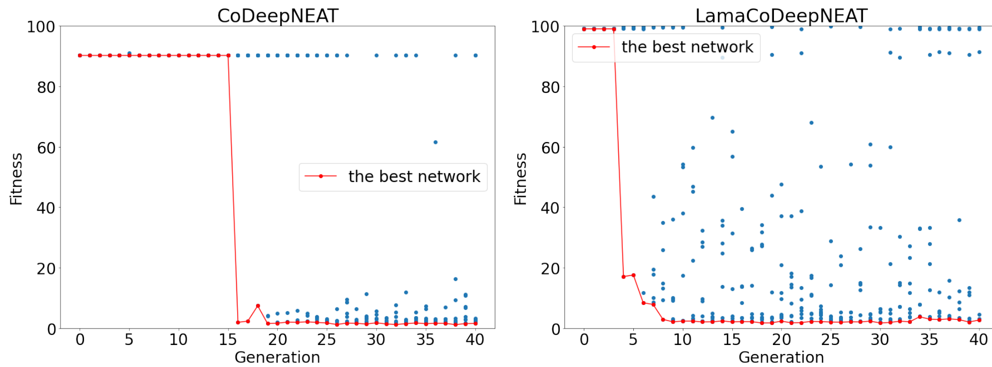


Figure 5.2: A network fitness evaluation during 40 generations in CoDeepNEAT and LamaCoDeepNEAT algorithms on the MNIST dataset.

See Figure 5.3 for further comparison of the average course of the experiment. Red points mean the best assembled networks with the lowest fitness value of each experiment. The red line runs through the mean of the points from the same generation, therefore it represents the average fitness function development in each method. We used for the visualization the experiments which run for all 40 generations (15 experiments for CDN and 12 experiments for LamaCDN in total).

The average course of the experiment is very similar for CoDeepNEAT and LamaCoDeepNEAT. They converge comparably fast and the light differences in achieved final fitness value may be reduced with the increase in the number of experiments. This conclusion supports the idea, that the MNIST dataset is too easy to solve to show the improvement caused by the weight inheritance implementation. The already mentioned article [75] contains a similar observation. Authors state, that the MNIST dataset is easy to solve and can be learned quickly by small networks, which leads to marginal improvements from weight inheritance.

We can calculate the average execution time of experiments, which ran on the same cluster and did not reach the wall time. See Table 5.4 to compare the average

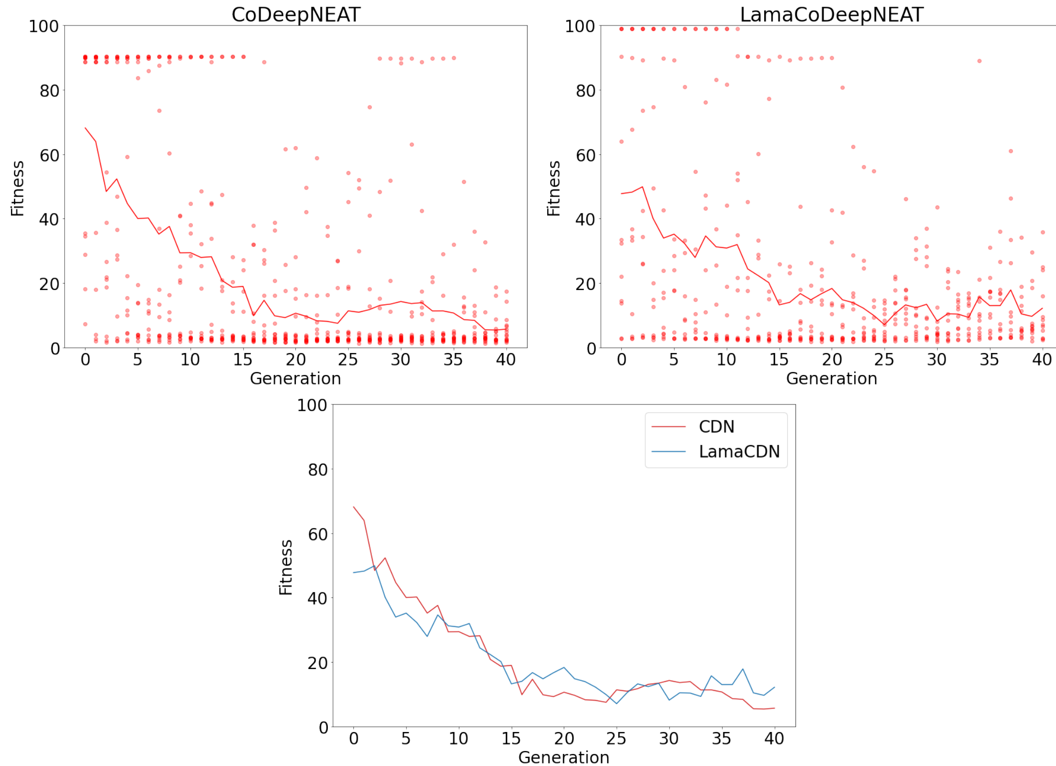


Figure 5.3: An average fitness function value of the best networks in every generation in CoDeepNEAT and LamaCoDeepNEAT on the MNIST dataset.

experiment execution time on some clusters[2]. LamaCDN experiments were less time-consuming on almost every cluster. In some cases, the differences are tens of thousands of seconds. So the conclusion is that LamaCDN performs MNIST experiments returning networks with similar performance as CDN networks, but in a shorter time.

Table 5.4 also shows the average execution time for Multi-objective experiments. The "X" mark means that MOLamaCDN experiments did not run on the particular cluster, so the data for calculation does not exist. The MOLamaCDN experiments are discussed in the next section.

	CDN		LamaCDN		MOLamaCDN	
Cluster	No. of runs	t [s]	No. of runs	t [s]	No. of runs	t [s]
aman	4	108,872	8	98,783	0	X
elan	1	74,051	1	88,115	3	13,471
gita	3	122,651	2	28,296	0	X
zelda	1	125,137	2	52,424	2	9,105
zenon	2	108,970	2	97,665	5	15,468

Table 5.4: Average execution time table for CoDeepNEAT, LamaCoDeepNEAT, MOCoDeepNEAT on the MNIST dataset.

[2]Parameters of the clusters are listed in Attachement A.1.

Multi-objective LamaCoDeepNEAT on MNIST

The best result obtained from MOLamaCDN experiments is the neural network with 1.55% error rate, and 1.23% error rate (e.x. 98.77% accuracy) after final training for 100 epochs. The best network has been created in the 35th generation. The FLOPs value is 270,430 and the number of all parameters is 54,410. This means that the best result of MOLamaCDN is as good as the results of CDN and LamaCDN, but is less demanding on computer resources. The least demanding network found by this experiment reached also promising results: the FLOPs are 112,926 and the error rate is 9.03%. This pair of networks is quite similar compared to the result pairs of other experiments. Usually, the network with the lowest FLOPs reached a high error rate, which was expected.

A visualization of the experiment's interim results is in Figure 5.4. A blue line shows the lowest fitness value from all assembled networks in each generation. A green line connects the lowest reached FLOPs values of all assembled networks in the current generation. So the figure does not show concrete fitness and FLOPs values for every network, because it shows the progress of the fitness and FLOPs evolution during the experiment that returned the best network.

The fitness value did not converge in the same way as in other experiments. The first-generation populations (randomly) created a very good network with circa 8% error rate, which caused a very fast and promising convergence. This probably also led to the similarity of both resulting networks, which is mentioned in the previous paragraph, even though each of them is aimed at a different objective. The neural architecture search focused on a subspace of less demanding and good-performing networks since the beginning.

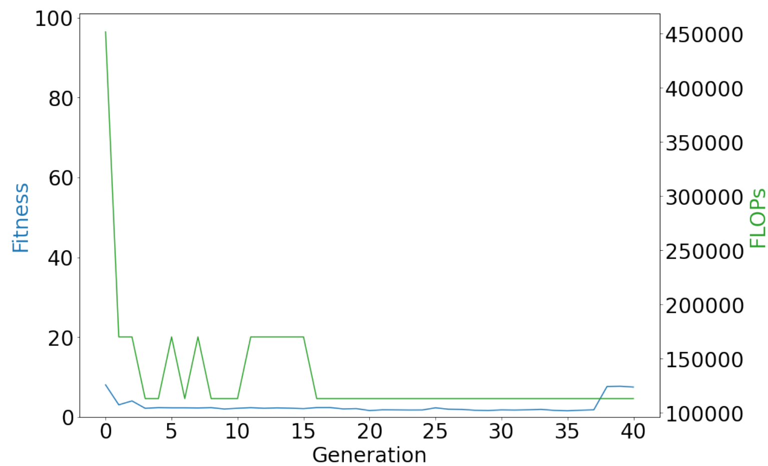


Figure 5.4: A fitness values of the best networks and FLOPs values of the least demanding networks in every generation in the Multi-objective LamaCoDeepNEAT experiment on the MNIST dataset.

A visualization of the development of the first non-dominated front during generation is shown in Figure 5.5 (for a closer description of non-dominated fronts see Section 1.6.2). In both images, a point represents a network from the best front and its fitness and FLOPs values. Colours of the points distinguish the generation, in which the network was created and assigned to the first front. The networks from the same generation are also connected with a dotted line. The

larger subfigure shows an overall view of all networks from the first fronts. The smaller subfigure is focused on the more interesting area, where the differences in reached objective values are more distinguishable.

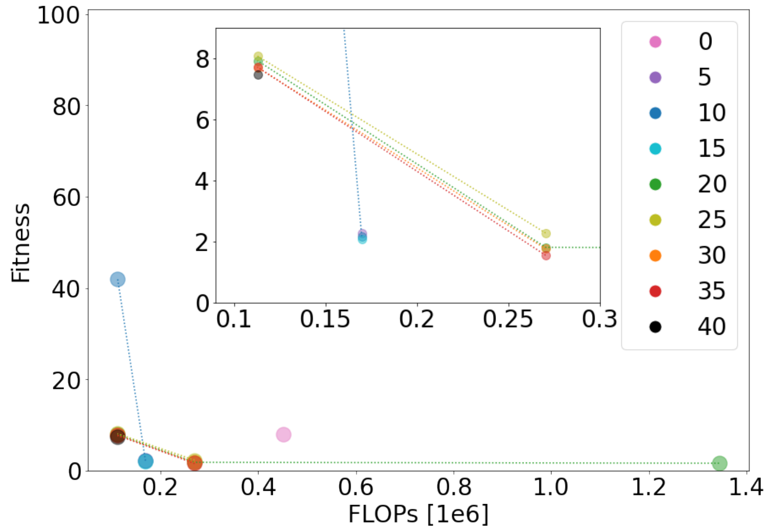


Figure 5.5: A visualization of the first non-dominated front of assembled networks in every fifth generation in Multi-objective LamaCoDeepNEAT experiment returning the best network on the MNIST dataset.

The networks became better in both objectives than their predecessors in almost every showed generation, which is a desired property of multi-objective optimization experiments. It seems that our proposed implementation of multi-objectiveness preserved its characteristic properties. Even though, it was applied to two coevolving populations (one of them divided into species) and assembled networks used to evaluate their objectives, not only on one population of individuals as usual. The experiment was evolving a more diverse population after the 15th generation, based on an observation, that the first fronts are larger after this particular generation.

An average course of MOLamaCDN experiments on the MNIST dataset can be seen in Figures 5.6 and 5.7. In Figure 5.6, a blue line connects the calculated average fitness value in each generation. Blue dots represent the fitness value of all generated networks. Similarly, the average FLOPs value is represented by a green line and the green dots represent all reached values by all networks in Figure 5.7. Both average objectives converge to optimal solutions comparably. The convergence of fitness values corresponds to LamaCDN and CDN experiments. On the other hand, it does not reach such good results as single-objective methods, but that is surely caused by the multi-objectiveness of opposing objectives.

Another interesting observation can be seen in Figure 5.7. The average FLOPs value is slightly increasing after 25th generation. This may be caused by chosen optimization objectives, which are conflicting - it is usually needed to increase the FLOPs of the network to decrease its error rate, and vice versa. But this idea needs to be supported by more experiments running for more generations.

Table 5.4 shows the average execution times of MOLamaCDN experiments and experiments of other methods on particular clusters. The MOLamaCDN executions were significantly faster, which is probably caused by multi-objective

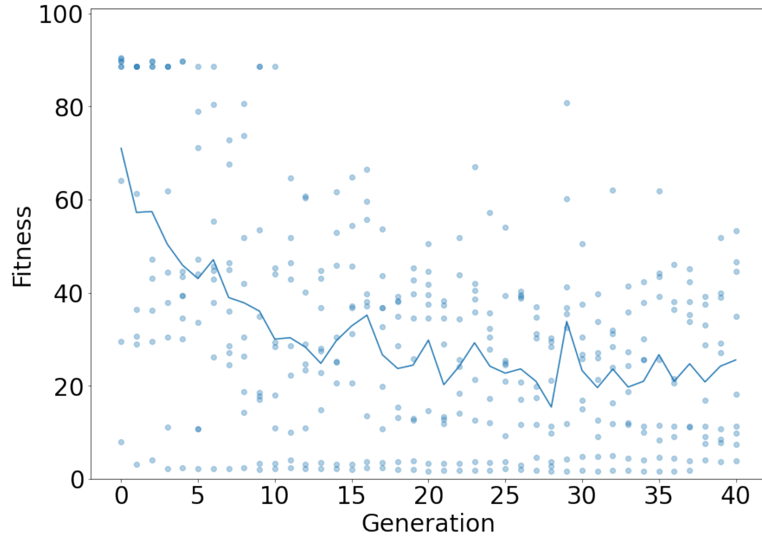


Figure 5.6: An average fitness function value of the best networks in every generation in Multi-objective LamaCoDeepNEAT on the MNIST dataset.

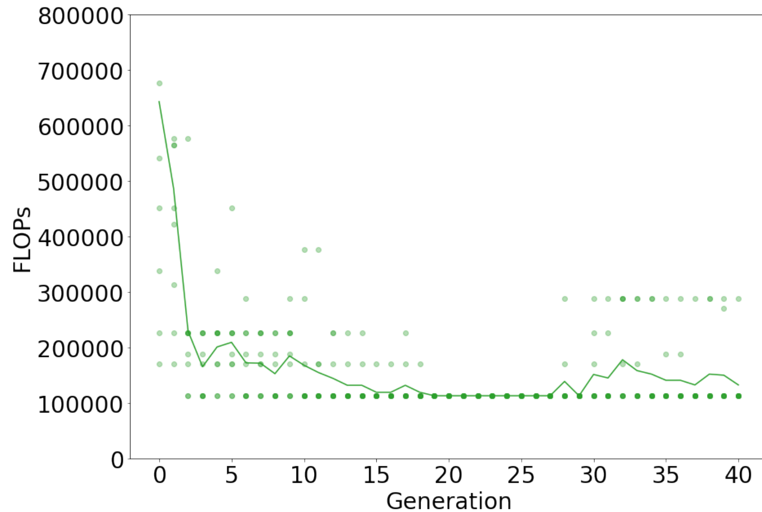


Figure 5.7: An average FLOPs value of the least demanding networks in every generation in Multi-objective LamaCoDeepNEAT on the MNIST dataset.

optimization and Lamarckian principles - the weight inheritance and evolving less computer-demanding neural networks due to FLOPs optimization. For example, the average execution time on the zenon cluster is 15,467.80 seconds for MOLamaCDN, 97,665.00 seconds for LamaCDN, and 108,970.00 seconds for the original CDN. The difference between MOLamaCDN and LamaCDN is more than 2.5 hours and the difference between MOLamaCDN and CDN is almost 26 hours.

5.4 Experiments with CIFAR-10

This section describes the results of experiments using the CIFAR-10 dataset. We introduce experiment parameters, then we present results and best models of each method.

We ran three experiments with different parameter settings for each method. The reason is the high consumption of computing resources needed for creating a CIFAR-10-based image recognition solution. We have proposed three versions of the experiment: a *short* experiment, a *short* experiment with *higher mutation probability*, and a *long* experiment. All variants are closely described in the following sections.

5.4.1 CIFAR-10 - Short Version

Parameters of Short Version

Bohrer et al. [64] considered the experiment parameters used in the original proposal [50] and estimated the time required to be difficult to achieve for common users. They have decided to run the CIFAR-10 experiments with settings similar to MNIST experiment parameters.

We have tried the same approach. Every evolution runs for 40 generations with populations of 30 modules, and 10 blueprints, which assembled 10 networks in each iteration. The networks are trained on the whole CIFAR-10 dataset (50,000 train images, 10,000 test images) for 4 epochs and then evaluated. The module mutation probability is 0.8, and the module crossover probability is 0.2. The blueprint mutation probability is 0.3, and the blueprint crossover probability is 0.1. Other parameters can be obtained from Tables 5.1, 5.2, 5.3, which were presented in the previous MNIST Section 5.3.

CoDeepNEAT and LamaCoDeepNEAT on CIFAR10 - Short Version

The best result of CoDeepNEAT short experiments with CIFAR10 is a neural network, which reached during the evolution an error rate of 39.7%, after training for 100 epochs it is 38.9%. The best network has been created in the 32nd generation. The FLOPs value is 20,021,278 and the number of all parameters is 238,730.

The LamaCoDeepNEAT best network has an error rate of 31.33% and after final training error rate of 31.13%. The network was created in 19th generation. The FLOPs value is 25,411,934 and the number of all parameters is 161,322. But experiments found another interesting network, which reached a higher error rate during the evolution, but after training it reached 70.61% accuracy, which is a 29.39% final error rate. And one experiment found a very promising network with an error rate of only 27.17% error rate, but unfortunately, the experiment ran out of time during the 26th generation.

Figure 5.8 compares the course of both experiments returning the best networks. It seems that both experiments have similar courses, but LamaCDN still returned a better result. More details about experiments can be obtained from Figure 5.9, which shows an average course of the LamaCDN and CDN experiments (we used for average computation 10 CDN experiments and 8 LamaCDN experiments).

The progress of the average error rate corresponds to the progress of experiments with similar settings described in [64], where the average line did not drastically increase/decrease either. Authors present as their best result a network with 77% accuracy (trained for a longer time), which is not significantly

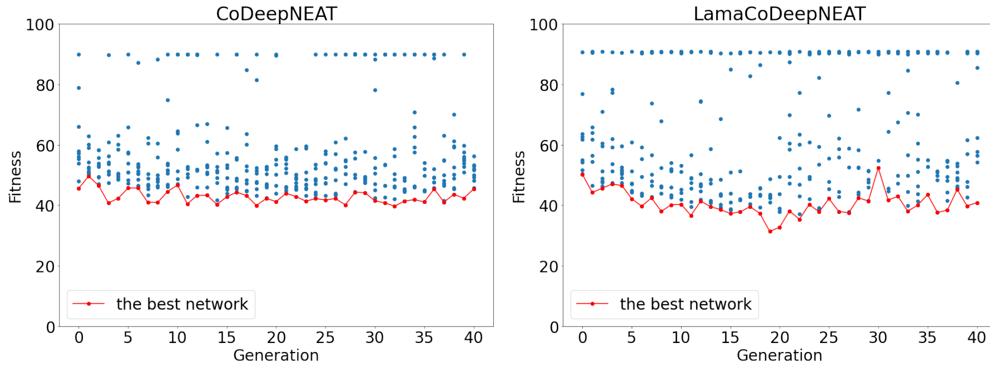


Figure 5.8: A network fitness evaluation during 40 generations in CoDeepNEAT and LamaCoDeepNEAT algorithms on the CIFAR10 dataset - short version of the experiment.

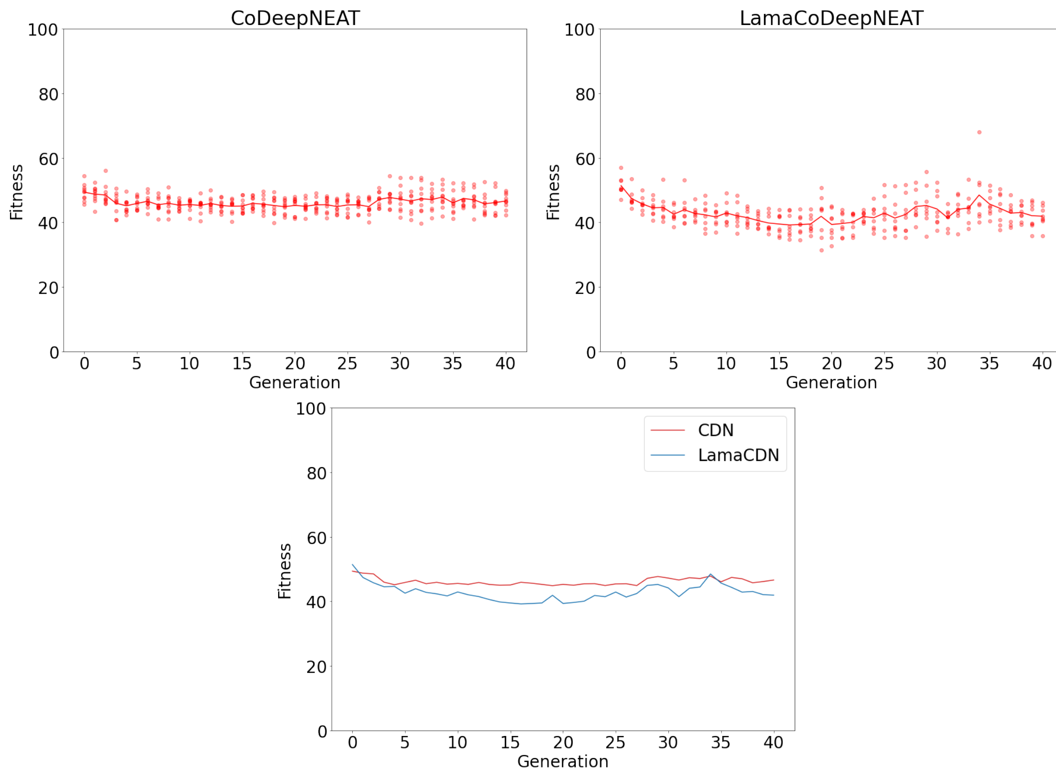


Figure 5.9: An average fitness function value of the best networks in every generation in CoDeepNEAT and LamaCoDeepNEAT on the CIFAR10 dataset - short version of the experiment.

different from our results. But their network is also more robust than ours (see Figure 20 in [64]), while our network is basically assembled from two convolutional modules with max pooling layers of size 2, the first has 128 filters and the kernel of size 3x3, the second module has 96 filters and the kernel size is 3x3. This simple architecture is not able to reach higher accuracy without increasing its complexity so it seems that we need to boost the evolution to support the network growth or let the experiments run with more computationally demanding parameters. These observations lead us to design a new set of experiments, which are discussed later.

If we focus on the average execution time of experiments on the CIFAR-10 dataset, we find out that the LamaCoDeepNEAT performed a faster search. The LamaCDN experiment lasted 82,184.33 seconds on average and CDN lasted 118,184.86 seconds (experiments ran on the kirke cluster). The difference is circa 10 hours.

Multi-objective LamaCoDeepNEAT on CIFAR-10 - Short Version

The best network obtained from short MOLamaCDN experiments has a 55.54% error rate. After the final training, it is 54.7%. The network was obtained in 27$^{\text{th}}$ generation, the FLOPs value is 213,002 and the total number of parameters is 82,058.

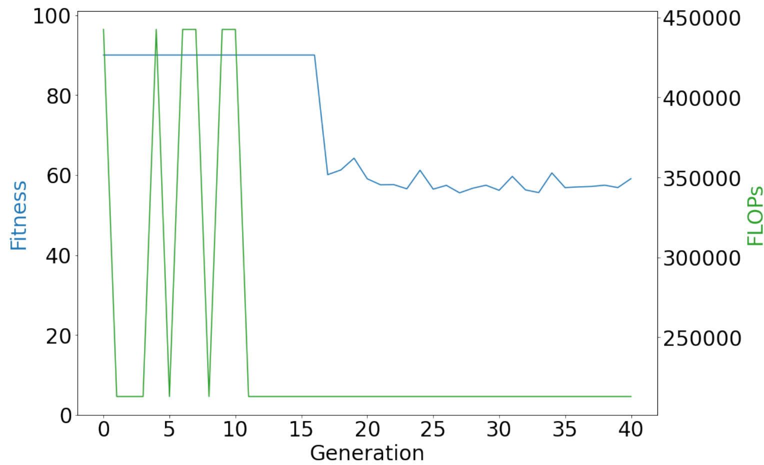
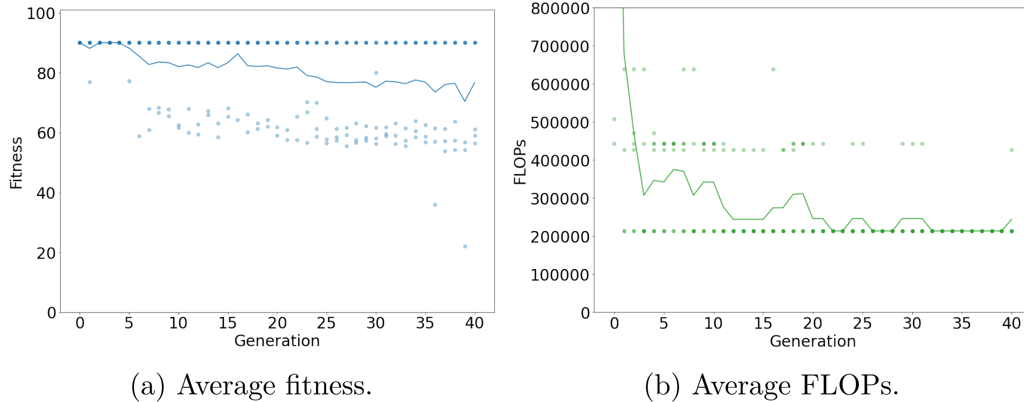


Figure 5.10: A fitness values of the best networks and FLOPs values of the least demanding networks in every generation in the short Multi-objective LamaCoDeep-NEAT experiment on the CIFAR-10 dataset.

This low accuracy is caused by the size and complexity of found neural architecture - the experiment returned a network assembled from only one convolutional module with max pooling and dropout layers. The resulting networks are very similar for the rest of the short MOLamaCDN experiments on CIFAR-10, see Figure 5.11, where the average course of experiments are displayed. On the other hand, found networks were very small, so it seems that minimization of FLOPs dominated. The reason causing this behavior may be that initialized networks were too small to be successful on the CIFAR-10 dataset, and FLOPs minimization on small networks was less challenging than accuracy maximization. A longer training during evaluation and a higher probability of increasing network architecture size during mutation may be helpful.

5.4.2 CIFAR-10 - Short Version, Higher Mutation Probability

We have decided to do another set of experiments with the CIFAR-10 dataset after obtaining the results from the short experiments. As we mentioned, all methods produced small networks, which were not able to perform well on that complex dataset. The hyperparameters of these experiments are different only in



(a) Average fitness. (b) Average FLOPs.

Figure 5.11: An average fitness function value of the best networks (left) and an average FLOPs value of the least demanding networks (right) in every generation in short Multi-objective LamaCoDeepNEAT on the CIFAR-10 dataset.

blueprint mutation probability. This change is supposed to boost the growth of the networks during evolution and help our methods to find larger (and therefore hopefully more accurate) resulting architectures. The specific changes in parameters are listed in Table 5.5.

Parameter	Old Value	New Value
Mutation probability	0.3	0.5
Add connection probability	0.2	0.35
Add node probability	0.2	0.35
Remove connection probability	0.05	0
Remove node probability	0.05	0
Mutate node species probability	0.3	0.1
Mutate optimizer probability	0.1	0.1
Crossover probability	0.1	0.1

Table 5.5: A comparison of blueprint evolution parameter values used in the short CIFAR-10 experiments and in the short CIFAR-10 experiments with higher mutation probability.

CoDeepNEAT and LamaCoDeepNEAT on CIFAR10 - Short Version, Higher Mutation Probability

The network with the lowest error rate during CoDeepNEAT evolution - 43.21%, has reached FLOPs value 82,420,382, the total number of parameters is 487,626 and its origin generation is the 38th generation. The final error rate obtained after training for 100 epochs is 36.06%.

The LamaCoDeepNEAT experiments have found the network with a 37.49% error rate and 27.31% error rate after final training. The network FLOPs value

is 74,726,942 and the number of parameters is 434,634. The network was found in the last generation.

See Figure 5.12 for fitness evaluation of CDN and LamaCDN experiments, which have found the best networks. Figure 5.13 shows an average course of both experiments. We can see that the LamaCDN experiments reached better error rates on average, similar to MNIST and short CIFAR-10 experiments.

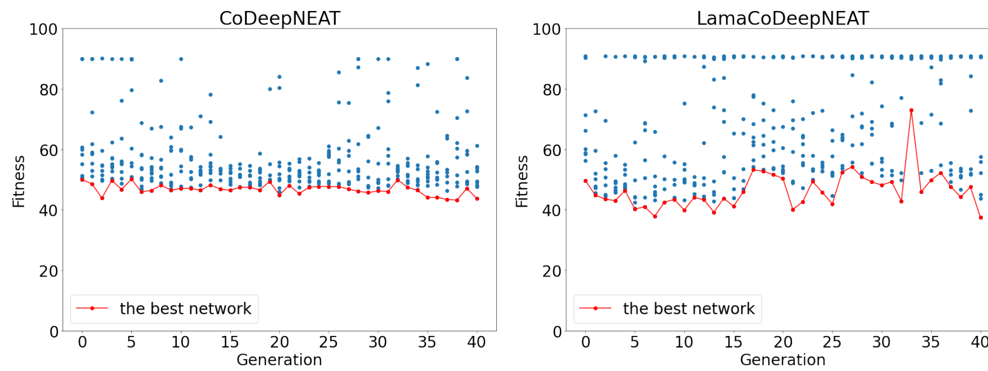


Figure 5.12: A network fitness evaluation during 40 generations in CoDeepNEAT and LamaCoDeepNEAT algorithms on the CIFAR10 dataset - short version of the experiment with bigger mutation probability.

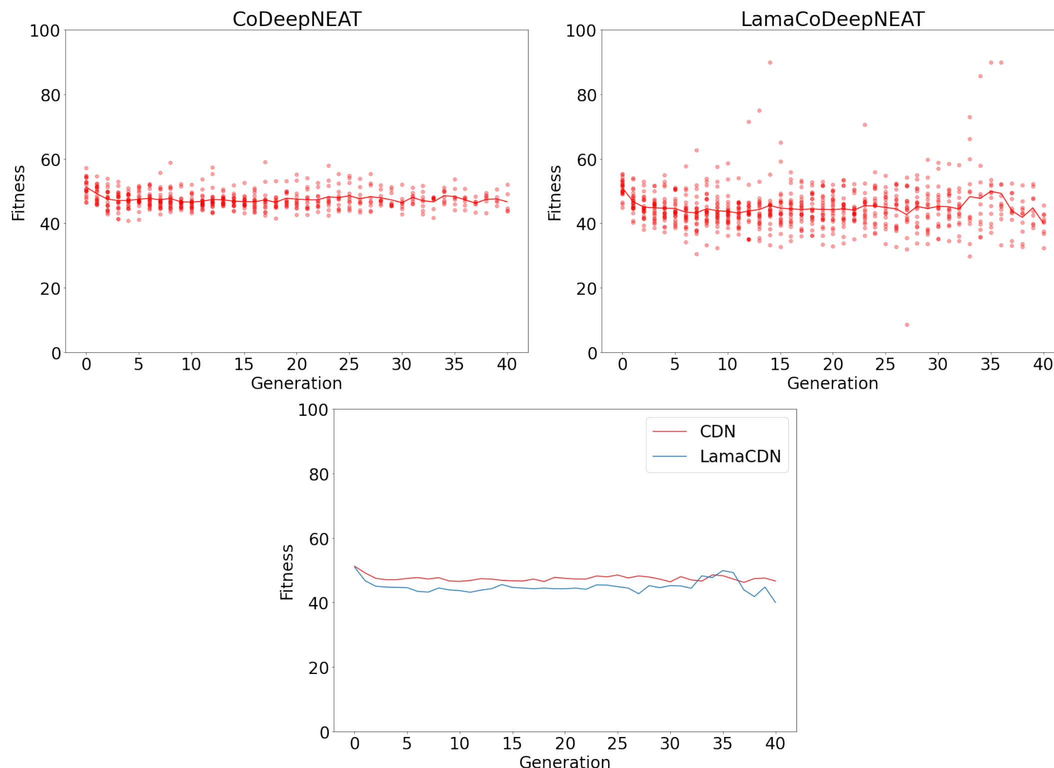


Figure 5.13: An average fitness function value of the best networks in every generation in CoDeepNEAT and LamaCoDeepNEAT on the CIFAR10 dataset - a short version of the experiment with bigger mutation probability.

It seems that increasing the mutation probability helped to find larger, therefore, better-performing networks, as we expected. A comparison of an average

fitness evolution of all four short CIFAR-10 experiments (CDN and LamaCDN with lower probability, CDN and LamaCDN with higher probability) can be seen in Figure 5.14. We can observe, that our LamaCDN experiments with lower probability reached the best fitness in around half time of evolution, meanwhile, LamaCDN with higher probability found the best networks in the last generations on average, which looks like a promising way to reach even better networks. Table 5.6 shows the average FLOPs value and number of total parameters reached by the best networks in every experiment. The average number of FLOPs increased in experiments with higher probability, on the other hand, the number of total parameters actually decreased, but the difference is not as big as it is in the FLOPs values.

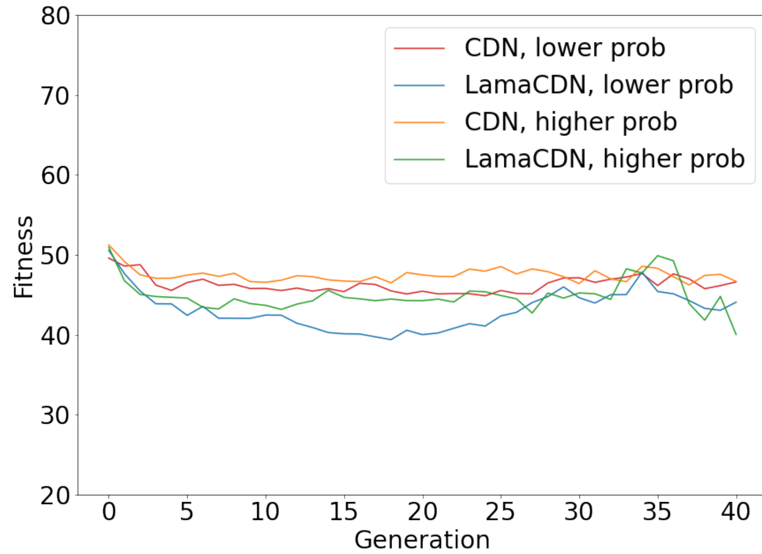


Figure 5.14: An average fitness function value of the best networks in every generation in CDN and LamaCDN on the CIFAR-10 dataset, short version, and in CDN and LamaCDN on the CIFAR-10 dataset, short version with higher mutation probability.

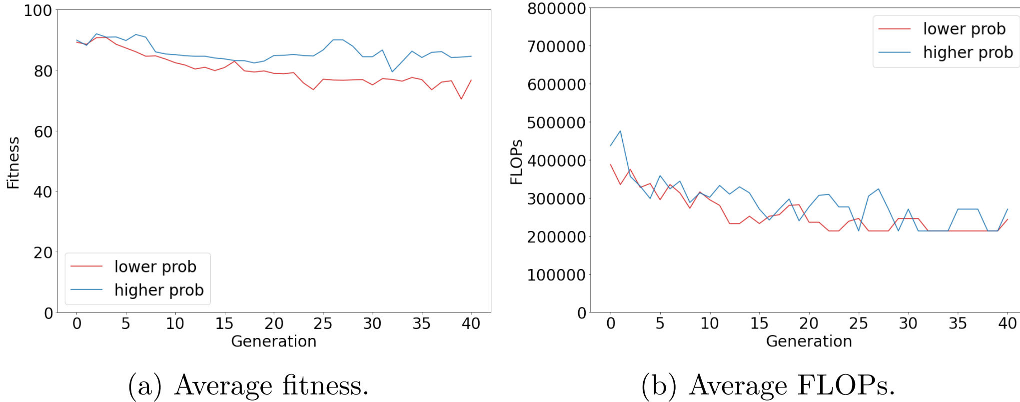
Method	Average FLOPs	Average total params
CDN, lower prob	55,285,676	1,304,161
LamaCDN, lower prob	43,751,295	363,929
CDN, higher prob	84,718,562	1,195,815
LamaCDN, higher prob	47,937,053	361,921

Table 5.6: A table of average FLOPs and the number of total network parameters reached by the best network for short experiments ran on the CIFAR-10 dataset.

Multi-objective LamaCoDeepNEAT on CIFAR10 - Short Version, Higher Mutation Probability

The higher mutation probability did not improve MOLamaCDN as it did with LamaCDN and CDN. This can be seen in Figure 5.15, which shows an average fitness value during evolution in all experiments in Figure 5.15a and an average

FLOPs value in 5.15b. A lot of experiments with higher probability did not finish the evolution before the time ran out (wall time was set to 48 hours), so it seems that their evolution becomes more computationally demanding. This would also explain the observation from the figure with average FLOPs 5.15b, that the experiments did not reach as low FLOPs values as MOLamaCDN experiments with lower probability. Extending execution wall time for experiments may bring more interesting results and more accurate networks.



(a) Average fitness.

(b) Average FLOPs.

Figure 5.15: An average fitness function value of the best networks (left) and an average FLOPs value of the least demanding networks (right) in every generation in short Multi-objective LamaCoDeepNEAT on the CIFAR-10 dataset with lower mutation probability and with higher mutation probability.

5.4.3 CIFAR-10 - Long Version

Parameters of Long Version

The parameters used for long experiments are inspired by parameters of the original CDN CIFAR-10 experiment [50].

Every experiment runs for 72 generations. A module population contains 45 modules, a blueprint population contains 25 blueprints and together they assemble 100 networks. A blueprint is used as a base for 4 networks in every iteration. The training uses all images from the database and it takes 8 epochs.

Other parameters are the same as for the short version, so we repeat them just briefly. A maximal number of species is set to 5 too. The module mutation probability is 0.8, and the module crossover probability is 0.2. The blueprint mutation probability is 0.3, and the blueprint crossover probability is 0.1. For a closer description of network components, look in Tables 5.1, 5.2, 5.3.

Another important parameter of the experiment is wall time, which is unique for the long version. Since we do not rely on evolution iterating for all 72 generations in a reasonable time, we stop the experiment after reaching a given wall time. Every long experiment had wall time set to 168 hours.

CoDeepNEAT and LamaCoDeepNEAT on CIFAR-10 - Long Version

The long CoDeepNEAT experiments on CIFAR-10 found the neural network with a 33.8% error rate gained during evaluation. This network is bigger than the

network found during the short version of the experiment - the FLOPs value is 127,627,630 and the number of trainable and non-trainable parameters is 674,314. The whole experiment lasted for 16 generations. The best network obtained from the long LamaCDN experiment has an evaluation error rate of 21.35%, 111,765,534 FLOPs, and 700,042 parameters. This experiment ran for 15 generations.

See Figure 5.16 for a comparison of fitness evolution during the long experiment of both algorithms.

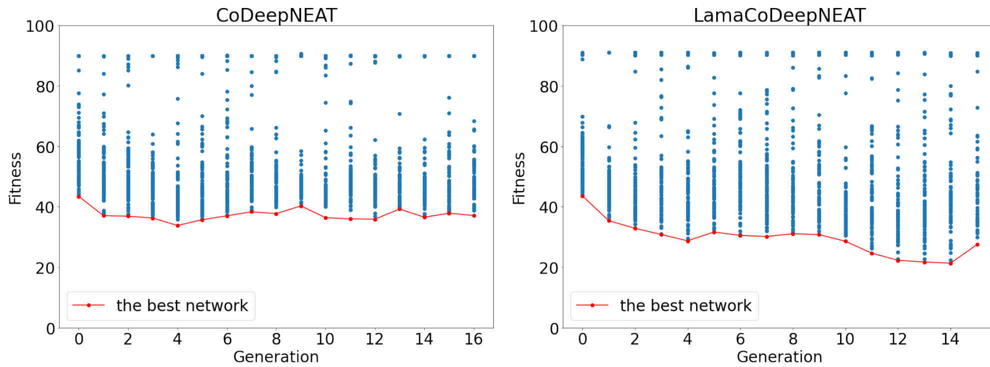


Figure 5.16: A network fitness in CoDeepNEAT and LamaCoDeepNEAT algorithms on the CIFAR10 dataset - the long version of the experiment.

An average course of both experiments is shown in Figure 5.17 (CDN presents 19 experiments, LamaCDN presents 23 experiments). The first observable difference between these two plots is the maximal reached length of evolution. Meanwhile, the longest CDN experiment ran for 19 generations, the longest LamaCDN experiment run for 41 generations. All of the long experiments run on similar clusters kirke, nympha, and halmir. The average number of generations of CDN experiments is 10.95, but the average number for the LamaCDN experiments is again higher - 17.91 generations, so it seems that the Lamarckian version is really faster than the original CDN without weight inheritance. Another difference is in the diversity of assembled networks - the CDN experiments obtained more similar networks than the LamaCDN experiments.

Multi-objective LamaCoDeepNEAT on CIFAR-10 - Long Version

The long MOLamaCDN experiments returned a network with a 45.36% error rate gained after training for 8 epochs during evaluation, 941,630 FLOPs, and 144,842 parameters, which makes it bigger and more promising than the best network of the short MOLamaCDN experiments on CIFAR-10. This experiment's smallest found network has 213,022 FLOPs and a 61.86% error rate. In total, it ran only for 11 generations in 168 hours long execution time. See Figure 5.18 for evolution of the lowest fitness and FLOPs values in this experiment.

Development of the first non-dominated front during evolution is shown in Figure 5.19. The MOLamaCDN evolved networks, which were getting better in both measured objectives during the experiment.

Two long experiments ran for all 72 generations, but the resulting best networks are not very accurate - they are too small and not very complex to reach good results. In this case, optimization of FLOPs objective dominated (as it

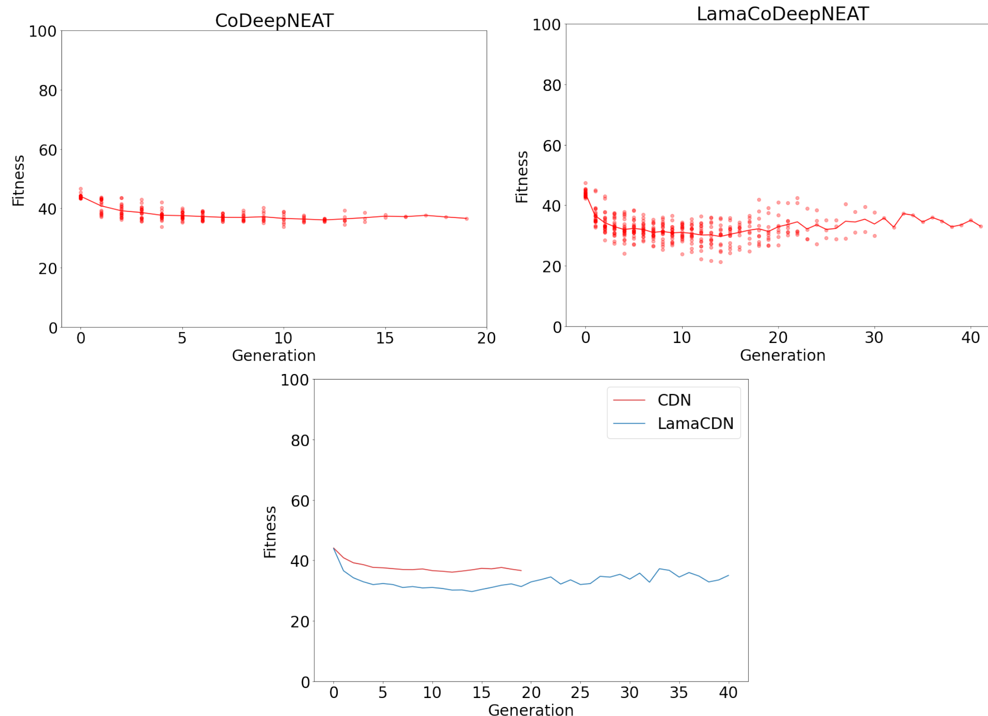


Figure 5.17: An average fitness function value of the best networks in every generation in CoDeepNEAT and LamaCoDeepNEAT on the CIFAR10 dataset - long version of the experiment.

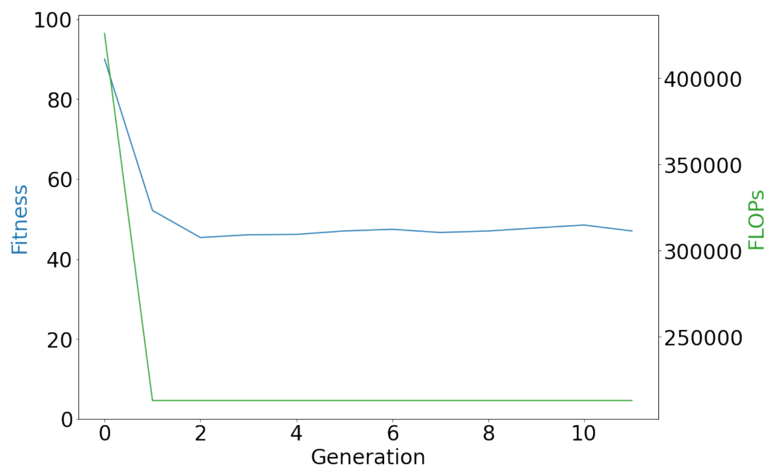


Figure 5.18: A fitness values of the best networks and FLOPs values of the least demanding networks in every generation in the long Multi-objective LamaCoDeep-NEAT experiment on the CIFAR-10 dataset.

happened in short experiments). The rest of the experiments ran for 7.54 generations on average and their average course of fitness and FLOPs value evolution is shown in Table 5.20.

The long versions of the CIFAR-10 experiments have shown more promising results for all three methods even though they did not finish evolution in comparison with the short version of experiments. During their short period of time, they have found more accurate networks, which is probably caused by their bigger size. The LamaCDN is faster than CDN on CIFAR-10 too, and multi-objective

optimization is able to optimize networks in both objectives in the CIFAR-10 dataset. The problem for future development is still the high time consumption.

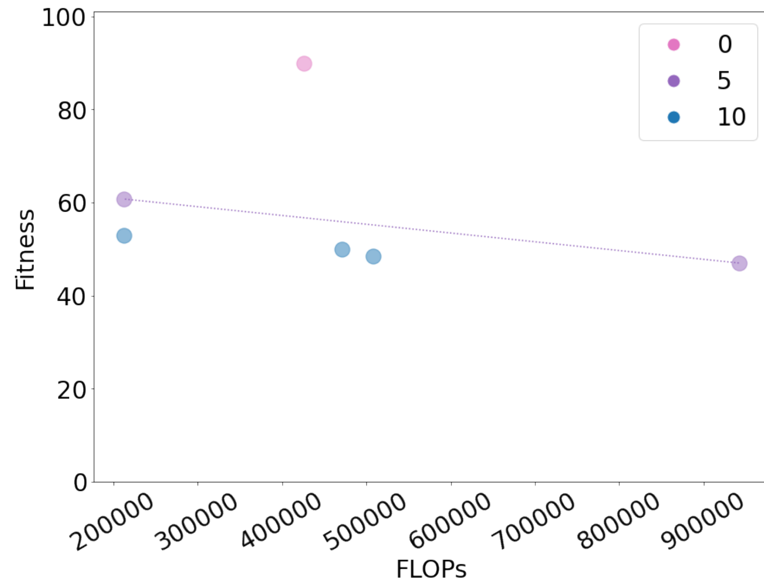
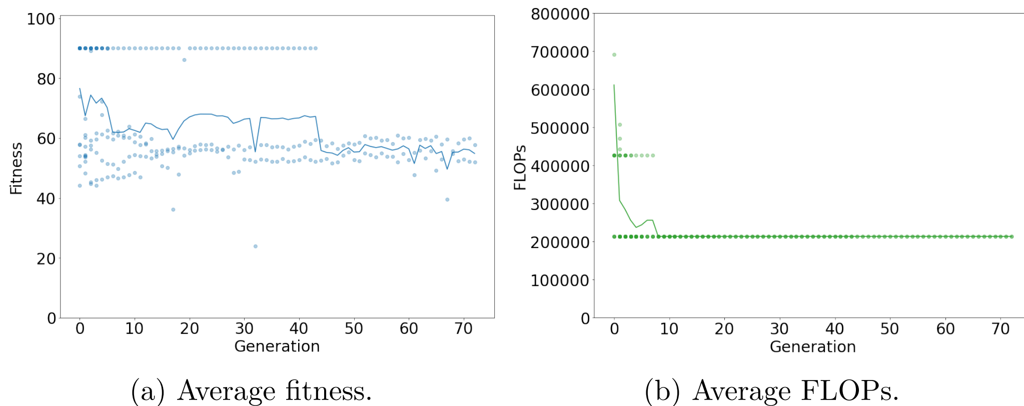


Figure 5.19: A visualization of the first non-dominated front of assembled networks in every fifth generation in Multi-objective LamaCoDeepNEAT on the CIFAR-10 dataset, long version of experiments.



(a) Average fitness.

(b) Average FLOPs.

Figure 5.20: An average fitness function value of the best networks (left) and an average FLOPs value of the least demanding networks (right) in every generation in short Multi-objective LamaCoDeepNEAT on the CIFAR-10 dataset.

Conclusion

In this work, we have proposed, implemented and tested two versions of CoDeepNEAT, which is a well-known neural architecture search algorithm based on coevolutionary algorithms and results in finding neural networks suitable for solving given problems. Both new versions were evaluated on several experiments with the MNIST and CIFAR-10 datasets.

The first version of CDN is called Lamarckian CoDeepNEAT and it enriches the original CDN implementation with neural network weight inheritance implementation, which is inspired by Lamarckian theory of evolution. LamaCoDeepNEAT does not waste hard trained network weights, rather it passes them onto similar successor networks used in the next generation of evolution. The experiments have shown that the method with this weight enhancement is able to find networks with a smaller error rate (or at least as good as the original CDN) in a much shorter time. In the case of the MNIST dataset, the biggest difference in average execution time on the same computation cluster is circa 26 hours between LamaCDN and CDN - a 77% faster execution time. In the case of the short experiments on the CIFAR-10 dataset, the difference is circa 10 hours - a 30% faster execution time. The long CIFAR-10 experiments, which were designed to work with huge populations and to evolve as many generations as they can until they reach the experiment wall time, showed similar results. The CDN experiments ran for 10.95 generations on average, meanwhile, the LamaCDN experiments evolved for 17.91 generations on average in the same amount of time.

The second proposed method is CoDeepNEAT adapted for multi-objective optimization (as opposed to original CDN, which focuses only on single-objective optimization), called Multi-objective LamaCDN since it also uses previously mentioned weight inheritance and LamaCDN as the base algorithm. So instead of searching for a neural network with the best accuracy, we have designed a neural architecture search algorithm, which focuses also on the network's computational requirements (i.e. network floating point operations). The multi-objective implementation is based on the NSGA-II algorithm used not only for blueprint and module selection but also for selection of assembled networks used for weight inheritance and mating. The experiments showed, that our approach preserves desired multi-objective optimization properties, minimizes both defined objectives and evolves networks better than their predecessors. The MOLamaCDN experiments have found a network with a comparably low error rate as the original version for the MNIST dataset, but this network has a much smaller FLOPs value and the total number of parameters. The smallest network from this experiment is even smaller, but the error rate is worse of course. The experiments with CIFAR-10 dataset have found networks, which are usually not as well performing as networks found by LamaCDN and CDN. On the other hand, their average sizes and FLOPs decreased during evolution and the minimization worked as we wanted. We have observed that the MOLamaCDN method is able to find non-demanding and well-performing networks for datasets solvable by small networks such as the MNIST dataset. The networks found for more complex datasets, like the CIFAR-10 dataset, are usually rather less computation-demanding than more accurate. This may be caused by the fact, that it is harder to achieve a good

error rate than a low FLOPs value when our first initial populations of randomly generated individuals create simple and small networks.

The presented methods may be enriched with the implementation of some method, which will boost the network growth in size, so they can achieve better performance in both LamaCDN and MOLamaCDN (e.g. different mutation probability, initialization with more complex individuals). The next possibility for improvement is to try different base methods - a different multi-objective optimization algorithm for example.

In more general, better results may be obtained by doing more and more experiments with different datasets and hyperparameters since significantly more computational resources are likely to become available in the near future. Increasing the number of training epochs will lead to getting better results but also higher computational consumption. In our work, we aimed only for evolving convolutional networks, but further study may focus on using both proposed methods on evolving LSTM networks suitable for e.g. language modeling, as it was done with CDN in the original work [50].

Bibliography

[1] Xinjie Yu and Mitsuo Gen. *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010.

[2] Charles Darwin. *On the origin of species, 1859*. Routledge, 2004.

[3] AE Eiben, JE Smith, AE Eiben, and JE Smith. Representation, mutation, and recombination. *Introduction to Evolutionary Computing*, pages 49–78, 2015.

[4] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80:8091–8126, 2021.

[5] Anant J Umbarkar and Pranali D Sheth. Crossover operators in genetic algorithms: a review. *ICTACT journal on soft computing*, 6(1), 2015.

[6] Thomas Bäck, David B Fogel, Darrell Whitley, and Peter J Angeline. Mutation operators. *Evolutionary computation*, 1:237–255, 2000.

[7] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1:3–52, 2002.

[8] Michael D Vose. *The simple genetic algorithm: foundations and theory*. MIT press, 1999.

[9] Holland Jh. Adaptation in natural and artificial systems. *Ann Arbor*, 1975.

[10] John N Thompson. Concepts of coevolution. *Trends in Ecology & Evolution*, 4(6):179–183, 1989.

[11] Chern Han Yong and Risto Miikkulainen. Cooperative coevolution of multi-agent systems. Technical report, Citeseer, 2001.

[12] Mitchell A Potter and Kenneth A De Jong. A cooperative coevolutionary approach to function optimization. In *Parallel Problem Solving from Nature—PPSN III: International Conference on Evolutionary Computation The Third Conference on Parallel Problem Solving from Nature Jerusalem, Israel, October 9–14, 1994 Proceedings 3*, pages 249–257. Springer, 1994.

[13] Bi Li, Tu-Sheng Lin, Liang Liao, and Ce Fan. Genetic algorithm based on multipopulation competitive coevolution. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 225–228. IEEE, 2008.

[14] David E Goldberg, Jon Richardson, et al. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, volume 4149. Hillsdale, NJ: Lawrence Erlbaum, 1987.

[15] Bruno Sareni and Laurent Krahenbuhl. Fitness sharing and niching methods revisited. *IEEE transactions on Evolutionary Computation*, 2(3):97–106, 1998.

[16] David A Van Veldhuizen and Gary B Lamont. Multiobjective evolutionary algorithm research: A history and analysis. Technical report, Citeseer, 1998.

[17] Aimin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and evolutionary computation*, 1(1):32–49, 2011.

[18] Patrick Ngatchou, Anahita Zarei, and A El-Sharkawi. Pareto multi objective optimization. In *Proceedings of the 13th international conference on, intelligent systems application to power systems*, pages 84–91. IEEE, 2005.

[19] Nidamarthi Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary computation*, 2(3):221–248, 1994.

[20] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[21] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. *IEEE transactions on evolutionary computation*, 18(4):577–601, 2013.

[22] Himanshu Jain and Kalyanmoy Deb. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach. *IEEE Transactions on evolutionary computation*, 18(4):602–622, 2013.

[23] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[24] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. pages 127–147, 1943.

[25] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[26] Hassan Ramchoun, Youssef Ghanou, Mohamed Ettaouil, and Mohammed Amine Janati Idrissi. Multilayer perceptron: Architecture optimization and training. 2016.

[27] Murat H Sazli. A brief review of feed-forward neural networks. *Communications Faculty of Sciences University of Ankara Series A2-A3 Physical Sciences and Engineering*, 50(01), 2006.

[28] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[29] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 2021.

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

[31] Victore Powell. Image kernels explained visually. `https://setosa.io/ev/image-kernels/`. [Online], accessed: 2023-04-04.

[32] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review. *arXiv preprint arXiv:2009.07485*, 2020.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[34] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[35] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023.

[36] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[37] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[38] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, pages 7105–7114. PMLR, 2019.

[39] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[40] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks. *arXiv preprint arXiv:1801.01078*, 2017.

[41] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[42] Yesmina Jaafra, Jean Luc Laurent, Aline Deruyver, and Mohamed Saber Naceur. Reinforcement learning for neural architecture search: A review. *Image and Vision Computing*, 89:57–66, 2019.

[43] Yuqiao Liu, Yanan Sun, Bing Xue, Mengjie Zhang, Gary G. Yen, and Kay Chen Tan. A survey on evolutionary neural architecture search. *IEEE Transactions on Neural Networks and Learning Systems*, 34(2):550–570, 2023.

[44] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary intelligence*, 1:47–62, 2008.

[45] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. *Advances in neural information processing systems*, 31, 2018.

[46] Renato Negrinho and Geoff Gordon. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017.

[47] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International conference on machine learning*, pages 550–559. PMLR, 2018.

[48] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR, 2018.

[49] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv preprint arXiv:1804.09081*, 2018.

[50] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. *arXiv e-prints*, pages arXiv–1703, 2017.

[51] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[52] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[53] Jean-Baptiste de Monet de Lamarck. *Philosophie zoologique, ou Exposition des considérations relatives à l'histoire naturelle des animaux...*, volume 1. Dentu, 1809.

[54] Lucas Gabriel Coimbra Evangelista and Rafael Giusti. Short-term effects of weight initialization functions in Deep NeuroEvolution. *Evo* 2021*, page 21, 2021.

[55] Jason Liang, Elliot Meyerson, Babak Hodjat, Dan Fink, Karl Mutch, and Risto Miikkulainen. Evolutionary neural automl for deep learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 401–409, 2019.

[56] Jeremy Cohen. How to Optimize a Deep Learning Model for faster Inference? `https://www.thinkautonomous.ai/blog/deep-learning-optimization/`. [Online], accessed: 2023-04-05.

[57] Francois Chollet et al. Keras: Deep Learning library for TensorFlow and Theano. `https://github.com/keras-team/keras/blob/c2e36f369b411ad1d0a40ac096fe35f73b9dffd3/keras/metrics.py`.

[58] Paul Pauls. The documentation of the Tensorflow-Neuroevolution framework. `https://tfne.readthedocs.io/en/latest/index.html`.

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

[60] Yann LeCun. MNIST handwritten digits database. `http://yann.lecun.com/exdb/mnist/`. [Online], accessed: 2023-04-03.

[61] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[62] Alex Krizhevsky. CIFAR-10 and CIFAR-100 datasets. `https://www.cs.toronto.edu/~kriz/cifar.html`. [Online], accessed: 2023-04-03.

[63] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, pages 7105–7114. PMLR, 2019.

[64] Jonas da Silveira Bohrer, Bruno Iochins Grisci, and Marcio Dorn. Neuroevolution of neural network architectures using codeepneat and keras. *arXiv preprint arXiv:2002.04634*, 2020.

[65] Jonas da Silveira Bohrer. Keras-CoDeepNEAT. `https://github.com/sbcblab/Keras-CoDeepNEAT`.

[66] Paul Pauls. The Tensorflow-Neuroevolution Framework. `https://github.com/PaulPauls/Tensorflow-Neuroevolution`.

[67] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[68] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek

Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[69] Sanghyeon An, Minjun Lee, Sanglee Park, Heerin Yang, and Jungmin So. An ensemble of simple convolutional neural network models for mnist digit recognition. *arXiv preprint arXiv:2008.10400*, 2020.

[70] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[71] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

[72] Minseok Seo, Hakjin Lee, Doyi Kim, and Junghoon Seo. Implicit stacked autoregressive model for video prediction. *arXiv preprint arXiv:2303.07849*, 2023.

[73] Josef Steppan. Sample images from MNIST dataset. [Online], accessed: 2023-04-03].

[74] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[75] Jonas Prellberg and Oliver Kramer. Lamarckian evolution of convolutional neural networks. In *Parallel Problem Solving from Nature–PPSN XV: 15th International Conference, Coimbra, Portugal, September 8–12, 2018, Proceedings, Part II 15*, pages 424–435. Springer, 2018.

List of Figures

1.1 Roulette wheel selection of five individuals. 8
1.2 Two-point crossover with two children. 9
1.3 Uniform crossover with one children and probability 0.5. 9
1.4 Bit-flip mutation of an individual. 10
1.5 The visualization of Pareto front. 13

2.1 The scheme of a perceptron. 16
2.2 The visualization of a feedforward neural network with two hidden layers. 17
2.3 An example of image convolution with sharpening kernel of author's cat photo. Produced with a website app available on [31]. . 18
2.4 An example of pixel matrix convolution with a kernel of size 2 × 2, valid padding and stride 1. 19
2.5 Max pooling with neighbourhood of size 2 × 2. 19

3.1 The abstract illustration of neural architecture search. 21
3.2 The visualization of the directed acyclic graph search space. The red connections define a found child model, which has input in node 1, and nodes 3 and 6 are output nodes. 24
3.3 A genotype to phenotype mapping example. Source: [51]. 26
3.4 The two types of structural mutation in NEAT. Source: [51]. . . . 27
3.5 NEAT crossover of two networks. Source: [51]. 28
3.6 CoDeepNEAT assembly of a neural network from a blueprint and modules. 29

4.1 The visualization of weight inheritance during the assembling of a neural network from the blueprint B and the modules M, N. The dimensionality of modules weight matrices is shown as a number of input channels × kernel size × kernel size × number of output filters. The weight matrix of module N is shrunken in the first dimension in order to make it compatible with module M. 32
4.2 The visualization of the cloning of blueprint. The original blueprint B creates two assembled networks M, N and two clones C, D. After training, each network passes its calculated objective values $o1$ and $o2$ to the relevant blueprint copy and copies become part of the population. 35
4.3 The effect of cloning on the population of individuals. Picture (1) shows a new population of five individuals (red, orange, green, blue, and purple). Picture (2) shows the cloning phase, where the red, green, and blue individuals are copied. The original individuals are then deleted in (3). From this population are then selected the best individuals, in the picture (4) those are the orange individual and the red and blue clones, the rest is removed. The final stage of the population (5) is ready for mating. 38

5.1 Sample images from the MNIST dataset, from [73]. 40

5.2 A network fitness evaluation during 40 generations in CoDeepNEAT and LamaCoDeepNEAT algorithms on the MNIST dataset. 43
5.3 An average fitness function value of the best networks in every generation in CoDeepNEAT and LamaCoDeepNEAT on the MNIST dataset. 44
5.4 A fitness values of the best networks and FLOPs values of the least demanding networks in every generation in the Multi-objective LamaCoDeepNEAT experiment on the MNIST dataset. 45
5.5 A visualization of the first non-dominated front of assembled networks in every fifth generation in Multi-objective LamaCoDeepNEAT experiment returning the best network on the MNIST dataset. 46
5.6 An average fitness function value of the best networks in every generation in Multi-objective LamaCoDeepNEAT on the MNIST dataset. 47
5.7 An average FLOPs value of the least demanding networks in every generation in Multi-objective LamaCoDeepNEAT on the MNIST dataset. 47
5.8 A network fitness evaluation during 40 generations in CoDeepNEAT and LamaCoDeepNEAT algorithms on the CIFAR10 dataset - short version of the experiment. 49
5.9 An average fitness function value of the best networks in every generation in CoDeepNEAT and LamaCoDeepNEAT on the CIFAR10 dataset - short version of the experiment. 49
5.10 A fitness values of the best networks and FLOPs values of the least demanding networks in every generation in the short Multi-objective LamaCoDeep-NEAT experiment on the CIFAR-10 dataset. 50
5.11 An average fitness function value of the best networks (left) and an average FLOPs value of the least demanding networks (right) in every generation in short Multi-objective LamaCoDeepNEAT on the CIFAR-10 dataset. 51
5.12 A network fitness evaluation during 40 generations in CoDeepNEAT and LamaCoDeepNEAT algorithms on the CIFAR10 dataset - short version of the experiment with bigger mutation probability. 52
5.13 An average fitness function value of the best networks in every generation in CoDeepNEAT and LamaCoDeepNEAT on the CIFAR10 dataset - a short version of the experiment with bigger mutation probability. 52
5.14 An average fitness function value of the best networks in every generation in CDN and LamaCDN on the CIFAR-10 dataset, short version, and in CDN and LamaCDN on the CIFAR-10 dataset, short version with higher mutation probability. 53
5.15 An average fitness function value of the best networks (left) and an average FLOPs value of the least demanding networks (right) in every generation in short Multi-objective LamaCoDeepNEAT on the CIFAR-10 dataset with lower mutation probability and with higher mutation probability. 54
5.16 A network fitness in CoDeepNEAT and LamaCoDeepNEAT algorithms on the CIFAR10 dataset - the long version of the experiment. 55

5.17 An average fitness function value of the best networks in every generation in CoDeepNEAT and LamaCoDeepNEAT on the CIFAR10 dataset - long version of the experiment. 56
5.18 A fitness values of the best networks and FLOPs values of the least demanding networks in every generation in the long Multi-objective LamaCoDeep-NEAT experiment on the CIFAR-10 dataset. 56
5.19 A visualization of the first non-dominated front of assembled networks in every fifth generation in Multi-objective LamaCoDeep-NEAT on the CIFAR-10 dataset, long version of experiments. . . 57
5.20 An average fitness function value of the best networks (left) and an average FLOPs value of the least demanding networks (right) in every generation in short Multi-objective LamaCoDeepNEAT on the CIFAR-10 dataset. 57

List of Tables

5.1 Experiment hyperparameter table. 42
5.2 Experiment parameter table for output dense layers. 42
5.3 Experiment parameter table for convolutional layers. 42
5.4 Average execution time table for CoDeepNEAT, LamaCoDeepNEAT, MOCoDeepNEAT on the MNIST dataset. 44
5.5 A comparison of blueprint evolution parameter values used in the short CIFAR-10 experiments and in the short CIFAR-10 experiments with higher mutation probability. 51
5.6 A table of average FLOPs and the number of total network parameters reached by the best network for short experiments ran on the CIFAR-10 dataset. 53

A.1 CPU properties of Metacentrum clusters used for experiments. . . 73

A. Attachments

A.1 Hardware

Cluster	CPU
aman	4x 14-core Intel Xeon E7-4830 v4 (2.00GHz)
elan	2x Intel Xeon Processor (2x 16) 2.2 GHz
gita	non-public data
halmir	64x AMD EPYC 7543
kirke	64x AMD EPYC 7532
nympha	32x Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
zelda	4x Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz
zenon	2x AMD EPYC 7351 (2x 16 Core) 2.40 GHz

Table A.1: CPU properties of Metacentrum clusters used for experiments.

A.2 Digital Attachements

The implementation of this work is also available at a GitHub repository on `https://github.com/pivodovr/MasterThesis`.

The repository directory structure:

LamaCoDeepNEAT

— examples - CDN, LamaCDN experiment scripts

— tests - CDN, LamaCDN implementation test scripts

— tfne - CDN and LamaCDN implementation

MOLamaCoDeepNEAT

— examples - MOLamaCDN experiment scripts

— tests - MOLamaCDN implementation test scripts

— tfne - MOLamaCDN implementation