ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Šinkovec** Jméno: **Petr** Osobní číslo: **420176**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Hranové kódování pro hluboké neuronové sítě

Název diplomové práce anglicky:

Edge Encoding for Deep Neural Networks

Pokyny pro vypracování:

Learning artificial neural networks (ANNs) is mostly understood as a continuous optimization problem of selecting appropriate parameter (weight) values. The complementary discrete optimization task of selecting an appropriate architecture of the network is, on the other hand, mostly solved by human experts. Although there exist several approaches to optimize the ANN architecture none of them scales well to network sizes present in state-of-the-art applications. The objectives of this work are:
1) Study methods for ANN architecture optimization. Focus on indirect encoding approaches and Edge Encoding (EE) in particular.
2) Design and implement an algorithm based on EE allowing optimization of deep neural network architectures (e.g., Convolutional Neural Networks).
3) Evaluate your algorithm experimentally. Focus mostly on scalability and ability to encode modular and hierarchical architectures.

Seznam doporučené literatury:

[1] Drchal, J. (2013). Base Algorithms for Hypercube-based Encoding of Artificial Neural Networks. Czech Technical University.
[2] Luke, S., & Spector, L. (1996). Evolving graphs and networks with edge encoding: Preliminary report. In Late breaking papers at the genetic programming 1996 conference (pp. 117-124). Stanford University, Stanford Bookstore CA, USA.
[3] Gruau, F. (1994). Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm. Ecole Normale Supérieure de Lyon, France.
[4] Miikkulainen, R., Liang, J., Meyerson et al. (2017). Evolving Deep Neural Networks. Retrieved from http://arxiv.org/abs/1703.00548
[5] Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Le, Q., & Kurakin, A. (2016). Large-Scale Evolution of Image Classifiers. Retrieved from http://arxiv.org/abs/1703.01041

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jan Drchal, Ph.D., centrum umělé inteligence FEL

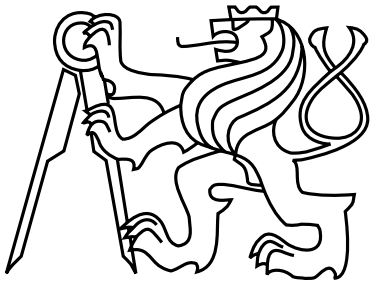
Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **08.02.2018** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce: **30.09.2019**

Ing. Jan Drchal, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Diploma Thesis

Edge Encoding for Deep Neural Networks

Petr Šinkovec

Supervisor: Ing. Jan Drchal, Ph.D.

Study Programme: Open Informatics, Master

Field of Study: Artificial Intelligence

May 7, 2018

Aknowledgements

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

Prague, date ....

Abstract

Acquiring systems for automatic structure optimization of machine learning (ML) models is desirable from two main reasons. Firstly, it can save ML designers man hours from more or less straightforward work and secondly, it can detect very effective and compact structures that would be very unlikely to be discovered by human. Notable success in the field of supervised ML classifiers achieved the artificial neural network (ANN), especially the deep ANN (DNN), which represents a model based system with particular internal architecture, that determines its performance. For the ANN topology optimization there exist several methods based on various principles where some of them are based on evolutionary algorithms (EA).

Thesis' objective is to investigate a new approach for DNN architecture optimization by the use of genetic programming (GP) algorithm over a population of computer programs that generate DNN. Individual in the context of used EA is a computer program whose instructions work with graph structure representation via the method called edge encoding. The belief is, that such type of encoding with a proper deployment in the context of ANN, can generate models with hierarchical and modular architectures that are especially valuable in most of the complex ML tasks.

Abstrakt

Získání systémů pro automatickou strukturální optimalizaci modelů strojového učení (ML) je žádoucí ze dvou hlavních důvodů. Za prvé může ušetřit ML designérům víceméně přímočarou práci a za druhé, dokáže odhalit velmi efektivní a kompaktní struktury, které by člověkem pravděpodobně nebyly objeveny. Pozoruhodný úspěch v oblasti ML klasifikátorů dosáhla umělá neuronová síť (ANN), zejména hluboká ANN (DNN), která představuje model–based systém s příslušnou vnitřní architekturou, která určuje výkonnost modelu. Pro optimalizaci topologie ANN existuje několik metod založených na různých principech, kde mnohé z nich vychází z použití evolučních algoritmů (EA).

Cílem diplomové práce je zkoumat nový přístup k optimalizaci architektury DNN pomocí algoritmu genetického programování (GP) v populaci počítačových programů, které generují DNN. Jedinec je v kontextu použitého EA počítačový program, jehož instrukce pracují s reprezentací grafové struktury pomocí metody zvané hranové kódování. Domněnka této práce je, že tento typ kódování se správným nasazením v kontextu ANN může generovat modely s hierarchickými a modulárními architekturami, které jsou obzvláště výhodné v mnohých náročných ML úlohách.

Keywords

machine learning, artificial neural network, deep neural network, convolutional neural networks, indirect encoding, edge encoding, evolutionary algorithm, genetic programming, grammatical evolution, modular architectures

Klíčová slova

strojové učení, umělá neuronová síť, hluboká neuronová síť, konvoluční neuronové sítě, nepřímé kódování, hranové kódování, evoluční algoritmus, genetické programování, gramatická evoluce, modulární architektury

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Structure optimization of machine learning models has the goal to minimize the computational effort of such a models to reach the desired performance. Structure optimization can be considered as a highly non linear optimization problem for which there is usually nothing better to use, than a simple exhaustive grid search. Genetic algorithms offer usually much better choice than an expensive grid search with the little dark side of the need to implement or at least use some extra procedures that itself require more hyperparameters and a careful implementation. Evolutionary algorithms encompasses a large number of variants how the evolution itself can be carried out – how to address the problem of selection, mutation, niching, size of a population, number of epochs, fitness function etc. The thesis point of interest is to investigate one variant of evolutionary algorithm – namely the grammatic evolution, which is being deployed over a practical problem – synthesis of artificial neural network (ANN) with the assistance of graph edge encoding.

Many machine learning model based classifiers was found to be effective in the case, when they contained some structural features or modules that at least partially deal with an input problem in more analytical way by typically spliting the problem to smaller subtasks which are more easy to solve and combine partial results together. There is apparently no universal mechanism of how to design such modular architectures and therefore it is fully up to the ML designers knowledge/experience or on a deployment of an evolutionary algorithm.

Nowadays (2018) there is still a considerable demand for visual ML classifiers where for the widest range of visual classification problems the Deep Convolutional Neural Network (CNN) is considered to be the best. When mentioning the benefits of modularity, we can encounter something like functional modules with specific topology structure in defacto all of the best deep CNN models. For example the models GoogLeNet (2014) [1] or Google Inception-v4 ResNet (2016) [2], [3] contained so called inception blocks which are the local architectural neural layer blocks composed of 5×5, 3×3, 1×1 sized convolutional filter layers, also combined with maximum pooling layers. When the inception blocks were combined with the rest of the models, it made those networks win the image classification contest, making them the most powerful models at their time.

There is naturally a conjecture, that such beneficial structures can spontaneously emerge during the process of some evolutionary algorithm working with sufficiently effective representations of searched architectures (in this case an ANN). It is very likely, that these representations would need some clever mechanism for encoding the neural network and also

for automatic exploiting of a knowledge about the fitness of one architecture when trying to come up with a better architecture. For the means of a clever encoding of the neural networks we have consulted several approaches from which the topic of graphical edge encoding approach and its related subproblems were assigned to the author. There is also a similar work that under the same context studied the cell encoding approach for DNNs (Matěj Doložal's diploma thesis [4]). Our thesis should make a similar assessment of studied phenomenon with additional deployment of several more techniques and analysis of more interesting experiments.

The project is a part of the series of research papers that are focused on automated module acquisition for DNN. Therefore the author of this thesis wants to make this work valuable for the needs of possible future participants in this research area by making the thesis' results, conclusions and methods descriptions as effective as possible at the expense of perhaps a less extensive theoretical background part of the thesis, which is however considered as a very redundant for those kind of people, who might read this work.

The problem assignment was very broad and directly encouraged to design such procedures, that facilitates the deployment of an edge encoding for neural networks. Naturally, in the beginning of a work on the project, there was carried out a literature search over similar topics by both; the author and the supervisor of this thesis. After that literature assessment it was clear, that there are several particular procedures, which had to be designed as a kind of novel approaches by the author himself. Afterward there naturally emerged a necessity of evaluate the used methods, which was also not straightforward.

Structure optimization of ML models is associated with so called *model–based* ML models. Especially for the ANNs, which are one of the most typical representative of those models, the problem of their structure optimization lies with the specification of the exact number of neurons in each layer, position of each layer in a network, type of a neuron activation function, neuron connections itself, as well as another particular properties of ANN layers; i.e. numbers and sizes of convolutional filters in CNN networks.

In the context of DNN, the structure optimization task was usually carried out by hand, according to the best empirically and generally known architectures for a particular problem. The major disadvantage of DNN computer aided architecture design frameworks rests simply in a huge computational demands that are associated with DNN training; therefore those automatic structure optimizers mostly rely on predefined sets of ANN configurations, whose more or less cleverly selected combinations are used for sampling the candidates for better architectures. Amongst those methods there are often used some means of ANN encoding, that facilitates the process of manipulation with ANN, that is necessary for the optimizer to perform. Such encodings are fundamental in this thesis as well; namely the indirect edge encoding 2.2.

1.1 Brief problem specification

This is a list of the most important starting points, which were essential for the objective of this work.

1. Design a context free **grammar**, that describes the programming language P for evolving **ANNs** – especially deep convolutional ANNs.
2. Design an **interpreter** of a language P that works over the graphs and uses an **edge encoding**.
3. Focus on the feature, that such language P must contain instructions that facilitates the notion of an **indirect encoding**.
4. Deploy **some evolutionary algorithm** to investigate the properties of a grammar.
5. Evaluate whole algorithm **experimentally** with a focus on the ability of evolving modular and hierarchical architectures of produced ANN.

1.2 Detailed problem specification

After several analysis and consultations with the supervisor, the final more detailed plan to solve the assignment of this thesis was specified as:

1. Design a context free grammar and its variants, that can serve as Backus Naur form of some programming language P, which in the result will be able to generate ANNs. Language P must work over a graph edge encoding.
2. The designed language P must contain the instruments for achieving modular and hierarchical architectures of resulting phenotypes – besides the basic edge encoding instructions, come up with some **recurrent** operators and **loops**. Moreover make the language and its interpreter able to work with **IF–ELSE** expressions and **variables** in the traditional programing sense.
3. As an evolutionary algorithm for evolving examined architectures, use the **grammatical evolution**.
4. As a genetic programming baseline algorithm use if possible the simplest approaches for population selection and fitness sharing – try to avoid further complications of already complicated algorithm ecosystem.
5. Make an assumption, that ANNs can be either a convolutional neural network (CNN) or a simple dense sequential layered network.
6. Networks themselves do not have to be sequential (in the sense of stacking one layer on top of the other layer). One layer can have multiple receiving layers – by considering the layers of the network as a nodes and the connections between layers as edges, then the network graph is in general an **acyclic graph**. (The network graph must be acyclic to avoid a recurrence and the natural emergence of recurrent neural network RNN that was not investigated in this work at all).

7. Investigate the properties of used methods on **CIFAR–10** dataset and **XOR–N** problem. Measure the **test errors** of found models on these datasets and also their **complexity**.

8. Use some method for **modularity measure** and combine it to make a quality assessment of an individual.

9. Find the best experimental setup for training CIFAR–10 models and make a useful **comparison with state-of-the-art approaches** – use third party results with their proper citations.

10. Report whether the used method was beneficial in the tasks, where the usage of specialized modules is typical – especially XOR–N problem, but also whether there emerged some interesting structures in CNNs on a problem of visual classification.

1.3 Related Work

Most of the experiments were evaluated on the CIFAR–10 dataset, which have been also used in several state-of-the-art (SOTA) related works. We are providing a comparison of the automatic and also the hand designed methods for ANN architecture discovery, by reporting their brief description alongside with achieved test errors and links to their particular papers.

1.3.1 Hand designed SOTA models CIFAR–10 results

Here is a brief overview of the best ANN models, that have been manually designed for maximizing CIFAR–10 dataset test accuracy. It is important to note the special treatment of dataset – especially a multiple levels of data augmentation have been used and also the smaller scale essence of those experiments that were dedicated primarily for the test error minimization:

1. 96.53%: *Fractional Max-Pooling* [5]. Uses data augmentation during training.

2. 95.59% *Striving for Simplicity: The All Convolutional Net* [6]. It reaches 92% without data augmentation, 92.75% with small data augmentation, 95.59% when using aggressive data augmentation and larger network.

3. 94.16% *All you need is a good init* (CTU paper by Jiří Matas) [7]. Only mirroring and random shifts used, no extreme data augmentation. Uses thin deep residual network with maxout activations.

4. 93.95% *Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree* [8].

5. 93.72% *Spatially-sparse convolutional neural networks* [9].

6. 94% is an estimated approximate test accuracy of the human on CIFAR–10 dataset. [10]

1.3.2 Other DNN structure optimization approaches

This overview shows several approaches for deriving suitable CNN architectures. All shown examples demonstrates the result of the experimental run on CIFAR–10 dataset. The data come from the most recent SOTA methods, that were aimed to acquire best architecture for CNN, LSTM, RNN, DNN models. However none of these methods were targeted to generate modular structure or a usage of indirect encoding techniques; instead of that they worked with a direct representation of ANN components.

1. Direct encoding approaches based on **cartesian genetic programming** for CNN architecture design described in a recent report [11](2017) reached on the reference CIFAR–10 dataset remarkable state-of-the-art results; 94% test accuracy.

   It is important to note, that this approach used direct CNN encoding and predefined numbers of its components; e.g. hand tailored sets of filter sizes to choose from. Number of evaluated CNN instances in this method per one experiment was circa 2000 in oppose to the same number in this work which was per one experiment run approximately 300 (because of limited resources).

2. Another very recent approach [12](2017) uses **reinforcement learning** to learn recurrent neural network **controller**, which searches for the best CNN architecture (in the space of hand tailored sets of CNN parameter values).

   It is not an evolutionary approach, but it still searches for the right CNN architecture. Stated approach enabled the best architectures to have 96.5% test accuracy which surpassed the best human designed CIFAR–10 classifier.

3. **CoDeepNEAT** approach [13](2017) is aimed for optimizing the DNN architectures. With a notable success it was deployed in the area of Long Short Term Memory networks (LSTM) as well as CNN.

   The comparable experiments proved the ability to discover compact CNN architectures with 92.3% test accuracy on CIFAR–10 dataset. This was achieved after the run of modified NEAT [14], [15], [16] algorithm with the population of 100 CNNs in 300 generations.

4. **Large-Scale Evolution** [17](2017) reaches 94.6%. The method is based on a direct CNN encoding with 11 special types of individual mutation and the population starting with only 1 individual. The source paper is especially interesting as it contains several practical tips for fitness evaluation speed up.

5. **Our experimental method** based on grammatical evolution and edge encoding for CNN synthesis achieved 76.8% test accuracy on CIFAR–10 dataset. It used only 30 individuals and 30 generations due to a lack of computational resources and the need to execute many preliminary experiments, which very reduced our overall resources for pursuing the system performance at achieving test accuracy.

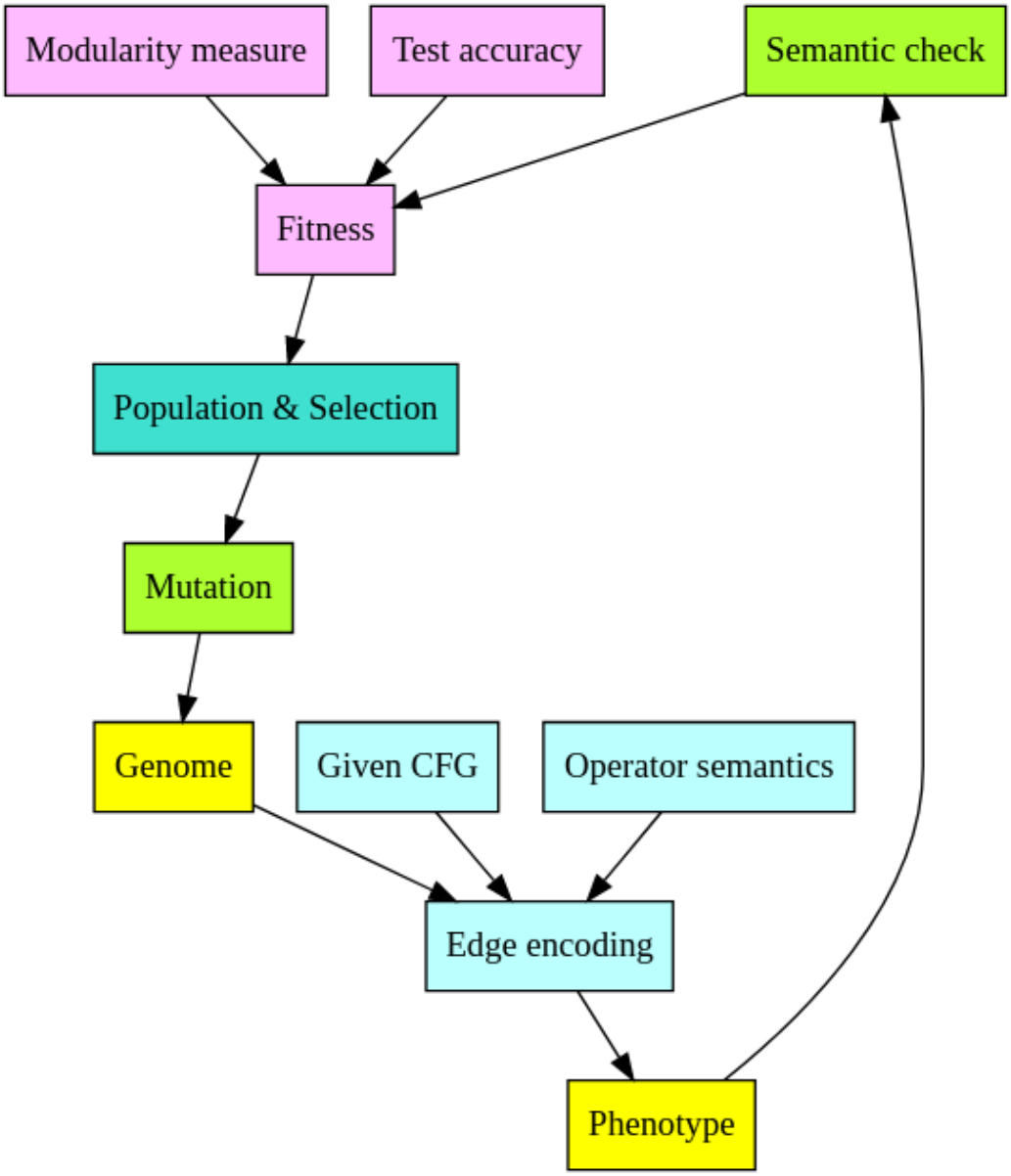


Figure 1.1: This schema should make a graphic overview of the whole workflow. There are many similarly useful ways of how to plot such a workflow graph. Colors of the nodes denote the similarity of their respective concepts. For example the process of mutation – i.e. introducing a new quality to an individual is closely linked with a semantic check. The fitness function consists of both the modularity measure and the test accuracy. The genome is just an encoding for the phenotype. Edge encoding needs the instructions with their semantics and it also needs a program working with these instructions – and a program needs a language which is defined by some grammar.

Chapter 2

Methods description

This chapter has a dual purpose. Firstly it makes a brief description of a each important phenomena with links to supplementary literature. Secondly it describes more closely the principles of details of those key concepts which were actually used and implemented in the experimental phase of the project. However, descriptions in this chapter never go to such latent details and problems, which emerges during the practical implementation of these procedures.

It is important to mention, that this chapter does not serve as a purely theoretical part of this thesis (which might be expected in this work), because the number of occurring topics is considerable and the usage of those topics is usually limited to a single occurrence in the whole algorithm ecosystem. From a large amount of details in the used methods, the author primarily wants to tell only the most useful and the most relevant facts.

2.1 Grammatical evolution

Grammatical evolution ([18], [19], chapter 2.3.2 in [14]) is a process of evolving a population of genomes which represents a sequences of instructions for a given context free grammar (CFG), for the purpose of finding such genomes, that has possibly the best properties at the given problem. Genomes themselves are the sequences of codons, that control what will the context free grammar generate. There is some starting non terminal symbol, which is being rewritten according to rules of a given context free grammar to another symbol – either a terminal or a nonterminal. The grammar rules have to be applied on nonterminal symbols in some way, that produces a final string consisting of only the terminal symbols.

In the context of this work, the resulting string produced by the grammar is a program that can be executed on top of an edge encoding (described in a section with encodings). To be more exact, the resulting string is rather considered as a program tree, which is much more suitable for a practical work with an expression then a simple string.

The process of grammatical evolution somewhat resembles a biological scheme of the genetic processes during an organism multiplication and development. Some of the original biological nomenclature with an equivalent of those concepts in the context of this work are enumerated here:

1. Genome (or chromosome) is a sequence of codons. It is a main unit for carrying a genetic information.
2. Codon in a living organism is a triplet of particular nucleotides.
3. Nucleotides can be considered as a basic units for carrying a genetic information (Adenine, Cytosine, Guanine, Thymine).
4. Phenotype is a final observable form of an organism. It may be for example a body of an animal.

In the context of this work, the above concepts represents the following:

1. Codon is a simple integer which represents a number of CFG rule, which will be applied for the rewrite of the current string.
2. Genome (or chromosome) is a list of integers. Alongside with the given CFG, the genome gives an encoding for a particular individual.
3. Nucleotides can be considered as equal to codons (but with respect to usual EA nomenclature we sticked to codons only and thus we will not talk about nucleotides any more).
4. Phenotype is simply an ANN produced by a program from a given genome.

2.1.1 Mechanism of grammatical evolution

For a reminder, here is a brief definition of grammar. It is a tuple $G = (N, \Sigma, S, P)$ where:

1. N is a finite set of nonterminal symbols;
2. Σ is a finite non empty set of terminal symbols where $N \cap \Sigma = \emptyset$;
3. $S \in N$ is a starting symbol;
4. P is a finite set of rules $\alpha \to \beta$, where α and β are words over $N \cup \Sigma$ such that, α contains at least one non terminal.

Context free grammar (CFG) (also Type-2 grammars according to the famous Chomsky hierarchy) is a grammar, where each rewriting rule is in the form $A \to \gamma$, where $\gamma \in (N \cap \Sigma)*$ and A is non terminal.

Given a concrete CFG G and a genome g we can take the codons of g and consider them as instructions for rewriting rules selections of a given G to produce some word of the language described by G. Process starts with declaring some starting non terminal symbol. We describe the mechanism on a following simple example.

1. Starting symbol:
   expr
2. Grammar G:

   $\langle expr \rangle ::= \langle END \rangle$
   $\quad | \quad \langle DOUBLE \rangle \langle expr \rangle \langle expr \rangle$
   $\quad | \quad \langle SPLIT\_CONV \rangle \langle int \rangle \langle stride \rangle \langle padding \rangle \langle expr \rangle \langle expr \rangle$

   $\langle int \rangle ::= 4 \mid 8 \mid 16 \mid 24 \mid 32$

   $\langle stride \rangle ::= 2$

   $\langle padding \rangle ::=$ valid
3. Genome g:
   $[1, 2, 3, 0, 0, 2, 1, 0, 0]$

Codon in a genome serves as an index of a rewriting rule to be applied on a current string; for example the rule <int>→24 has index 3 and the rule <expr>→END has index 0. Rewriting goes in the depth first manner. If the codon integer value is greater then the number of applicable rules, then it is used the remainder after division by the number of rules. If some non terminal can be rewritten by exactly 1 rule, then that rule is used immediately without consumption of the input codon; this can be seen in the following example at the points 3. → 4. and 7. → 8. The sequence of rewriting operations is following:

1. *expr*, genome: $[\mathbf{1}, 2, 3, 0, 0, 2, 1, 0, 0]$
2. DOUBLE *expr expr*, genome: $[\mathbf{2}, 3, 0, 0, 2, 4, 0, 0]$
3. DOUBLE SPLIT_DENSE *int stride padding expr expr expr*, genome: $[\mathbf{3}, 0, 0, 2, 4, 0, 0]$
4. DOUBLE SPLIT_DENSE 24 2 valid *expr expr expr*, genome: $[\mathbf{0}, 0, 2, 4, 0, 0]$
5. DOUBLE SPLIT_DENSE 24 2 valid END *expr expr*, genome: $[\mathbf{0}, 2, 4, 0, 0]$
6. DOUBLE SPLIT_DENSE 24 2 valid END END *expr*, genome: $[\mathbf{2}, 4, 0, 0]$
7. DOUBLE SPLIT_DENSE 24 2 valid END END SPLIT_DENSE *int stride padding expr expr*, genome: $[\mathbf{4}, 0, 0]$
8. DOUBLE SPLIT_DENSE 24 2 valid END END SPLIT_DENSE 32 2 valid *expr expr*, genome: $[\mathbf{0}, 0]$
9. DOUBLE SPLIT_DENSE 24 2 valid END END SPLIT_DENSE 32 2 valid END *expr*, genome: $[\mathbf{0}]$
10. DOUBLE SPLIT_DENSE 24 2 valid END END SPLIT_DENSE 32 2 valid END END

Resulting string corresponds to the individual program:

```
DOUBLE(SPLIT_DENSE(24, 2, valid, END, END),
       SPLIT_DENSE(32, 2, valid, END, END))
```

2.2 Phenotype encodings

Various types of phenotypes (e.g. ANN) can be straightforwardly represented as graphs or whatever encoding structure for which exists an unambiguous transformation from the space of defined encoding to the space of phenotypes.

There are two basic types of encodings:

1. **Direct encoding.** It uses a *direct* mapping from genotype to phenotype, which means that this mapping provides a complete and unambiguous information about the phenotype reconstruction. As the ANNs can be considered as graphs, they can be easily encoded with the connectivity matrix M, where the weighted connection between neuron i and neuron j is encoded by a matrix element $M_{i,j}$ (for RNNs with self loops there are also non zero elements on and under the diagonal of the matrix M). From this typical example of direct ANN encoding it is clear, that for the purpose of encoding an ANN, the code must have some polynomial size $p(n)$ of elements with respect to the number of neurons n. When dealing with bigger ANN instances, the optimization becomes very problematic as the encoding did not reduce the size of the search space.

2. **Indirect encoding.** This type of encoding enables to reduce the search space of codes with respect to the size of their produced respective phenotypes; i.e. the length of the (indirect) code is no longer necessarily proportional to some polynomial of the number of (e.g. ANN) phenotype elements – it can be significantly shorter, which is the purpose of indirect encoding techniques. This advantage goes on the expense of the need to design reasonable indirect encoding mappings, which are typically in the form of parameterized programming instructions; i.e. loops, and recursion.

As it was told, the definition of phenotype (especially graph) encoding can be arbitrary as long as the encoding will be unambiguous. There are two types of interesting and effective graphs encodings. These are cellular and edge encodings.

2.2.1 Cellular encoding

Cellular encoding [20] describes how can be an arbitrary graph generated from a single starting cell (or node). The most important rules for controlling, how the cells are gradually added and connected is based on utilization of the SEQ and PAR operators. Recursive structures can be made by REC operator. Suitable description of cellular encoding is also in the related thesis [4]. Details of cellular encoding are not relevant for this work and therefore they are omitted.

2.2.2 Edge encoding

It is an analogous technique [18], [20], [21] to the cellular encoding which works with edges instead of nodes. At the same time it is a core type of encoding which is investigated in this thesis. Edge encoding can describe general graphs (recurrent or acyclic) by gradual application of edge encoding operators on the single starting edge. Two operators that are sufficient to generate all acyclic graphs are SPLIT and DOUBLE. There are two extra

operators, that facilitates the synthesis of recurrent graphs, they are the LOOP and the REVERSE operator.

These are the very basic and general types of edge encoding operators. For the purpose of this thesis there were designed several more customized operators that are described in other section:

1. Operator SPLIT($expr_1$, $expr_2$). It is applied on a given input edge $e = (node_A, node_B)$ and produces new node $node_C$ and two new edges $e_1 = (node_A, node_C)$ and $e_2 = (node_C, node_B)$. After that the original edge e is deleted and input expression $expr_1$ is applied on e_1 and expression $expr_2$ is applied on e_2.
2. Operator DOUBLE($expr_1$, $expr_2$). It is applied on a given input edge $e = (node_A, node_B)$ and produces new edge $e' = (node_A, node_B)$. After that it applies $expr_1$ on e and $expr_2$ on e'.
3. Operator LOOP($expr$). It is applied on a given input edge $e = (node_A, node_B)$ and produces new edge $e' = (node_B, node_B)$ and after that it applies $expr$ on e'.
4. Operator REVERSE($expr$). It is applied on a given input edge $e = (node_A, node_B)$ and produces new edge $e' = (node_B, node_A)$, then it removes edge e and after that it applies $expr$ on e'.

2.3 Custom grammar & Operator design

There are many ways of how to define a reasonable grammar for the purpose of this thesis' objectives i.e. edge encoding and synthesizing deep convolutional ANNs. The investigated grammars were consisting from this list of rules:

⟨*expr*⟩ ::= ⟨*END*⟩
| ⟨*DOUBLE*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*FOR*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*REPEAT*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*IF_ELSE*⟩ ⟨*bool*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*VAR_ASSIGN*⟩ ⟨*int_var*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*SPLIT_CONV*⟩ ⟨*filter_size*⟩ ⟨*n_filters*⟩ ⟨*stride*⟩ ⟨*padding*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*SPLIT_POOL*⟩ ⟨*filter_size*⟩ ⟨*stride*⟩ ⟨*padding*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*SPLIT_DENSE*⟩ ⟨*int*⟩ ⟨*expr*⟩ ⟨*expr*⟩

⟨*int*⟩ ::= ⟨*int_digit*⟩
| ⟨*int_var*⟩
| ⟨*int_op*⟩ ⟨*int*⟩ ⟨*int*⟩

⟨*bool*⟩ ::= ⟨*bool_op*⟩ ⟨*int*⟩ ⟨*int*⟩

⟨*bool_op*⟩ ::= ⟨*gt*⟩ | ⟨*ge*⟩ | ⟨*lt*⟩ | ⟨*le*⟩

⟨*int_digit*⟩ ::= 0 | 1 | 2 | 4 | 8 | 16

⟨*int_op*⟩ ::= ADD | SUB | MUL

⟨*filter_size*⟩ ::= 1×1 | 3×3 | 5×5

⟨*n_filters*⟩ ::= int

⟨*stride*⟩ ::= 1
| 2

⟨*padding*⟩ ::= same
| valid

⟨*int_var*⟩ ::= X1 | X2 | X3 | X4

The rules <bool>, <bool_op>, <stride>, <n_filters>, <filter_size>, <int_op>, <int_digit>, <padding> are straightforward because they produces terminal expressions. Non terminal rules are for <int> and <expr> which is the most interesting rule. There was added a support for the usage of up to 4 different variables which behave in the common programming manner (variable scope, write to, read from, define variable).

The crucial rule <expr> consists of possible 9 rewriting options, where each one produces a special edge encoding operator which is applied on the input edge $e = (node_A, node_B)$. Several operators contains properties which are very specific for the domain of convolutional neural networks: these concepts are padding, stride, convolutional filter size, number of neurons etc. The explanations of these concepts is also sufficiently described in related thesis [4] and for the sake of compactness and especially the fact, that they can be treated as black box parameters, their description is omitted.

Details of the operators are following:

1. Operator END(). It is a simple termination of an expression; i.e. the execution on input edge e do not continue.
2. Operator IF_ELSE(*boolean_value*, $expr_1$, $expr_2$). Based on the truth value of *boolean_value* either *expr_1* or *expr_2* is applied on input edge e.
3. Operator VAR_ASSIGN(*variable*, *int*, *expr*). The value *int* is assigned to *variable*. After that the execution continues by applying *expr* on input edge e.
4. Operator SPLIT_CONV($filter\_size$, $n\_filters$, *stride*, *padding*, $expr_1$, $expr_2$). Parameter values $filter\_size$, $n\_filters$, *stride* and *padding* are used for the setup of a new convolutional neural layer l. There is created a node *node_C* and the layer l is inserted into the *node_C*. There are created 2 new edges $e_1 = (node_A, node_C)$ and $e_2 = (node_C, node_B)$ and the original edge e is deleted. After that the $expr_1$ is evaluated on e_1 and $expr_2$ is evaluated on e_2.
5. Operator SPLIT_POOL($filter\_size$, *stride*, *padding*, $expr_1$, $expr_2$). It is analogous to the SPLIT_CONV operator with the difference, that the particular used layer is not a convolutional layer, but it is the pooling layer. Consequently there is not any reason for having a parameter $n\_filters$ which is naturally omitted.
6. Operator SPLIT_DENSE(*int*, $expr_1$, $expr_2$). Again, it is analogous to the previous operator. Instead of pooling or convolutional layer it is used a simple fully connected layer of *int* neurons.
7. Operator FOR(*int*, *expr*). It is described in its own section 2.4.1.
8. Operator REPEAT(*int*, *expr*). It is described in its own section 2.4.2.

2.4 Modularity operators

The key property of our algorithm to be able to produce modular and scalable ANN architectures, is facilitated by two special indirect edge encoding operators: FOR and REPEAT. They use an indirect edge encoding for the introduction of new phenotype structures via some repetitive event. As far as the author of this thesis is aware at the time of writing this report and with respect to discussions with his supervisor, these approaches can be considered as a novel approaches. Author provides hereby a pseudo code, text description and a set of illustrative images, that will certainly make the comprehension of these approaches faster and easier.

2.4.1 Operator REPEAT

As every operator in the context of edge encoding, it is applied on a single start edge *input_edge*. First argument of REPEAT operator is a positive integer n which denotes the number of repeats. Second parameter is an expression *input_expr* which will be processed n times.

Operator starts with defining a set $E = \{input\_edge\}$; i.e. it contains only the original edge. After that it is applied the original *input_expr* on each $edge \in E$ (in the first iteration there is only 1 edge) and the new emerged edges e_{new} (that were created from *edge* and *input_expr*) are given into a set E'. During the iteration over the $edge \in E$ there is gradually constructed the E_{union} from all sets E' in the loop. The set E is in the end replaced by the set E_{union} and the whole process is repeated again, until the number of iterations reaches n. This operator makes clearly an exponential growth of the phenotype, which has to be carefully taken into account.

Algorithm Operator REPEAT. Function **create_graph_phenotype** is the edge encoding interpreter function. It takes as the parameters: 1) Input edge *input_edge*. 2) Expression which is applied to that edge *input_expr*. 3) Variables set in the form of key value mapping *variables*.

```
E = {input_edge}
for i = 1..n do
    E_union = {}
    for edge ∈ E do
        E' := {edge}
        create_graph_phenotype(edge, input_expr, variables)
        // E' contains all edges, that emerged after the create_graph_phenotype call
        E_union := E_union ∪ E'
    end for
    E := E_union
end for
```

For an example that should make the principle of REPEAT operator clear, imagine a simple program REPEAT(3, DOUBLE(SPLIT(END, END), SPLIT(END, END)). The program DOUBLE(SPLIT(END, END), SPLIT(END, END)) is applied 3 times on an input edge. Its execution leads to the following steps:

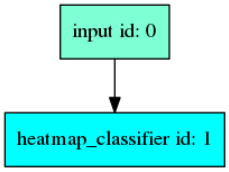


Figure 2.1: REPEAT example. Original input edge.

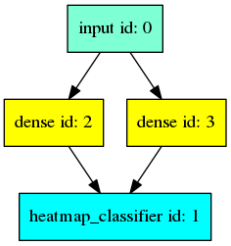


Figure 2.2: REPEAT example. After the first repeat.

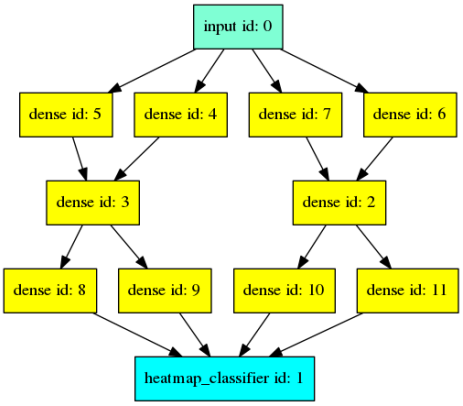


Figure 2.3: REPEAT example. After the second repeat.

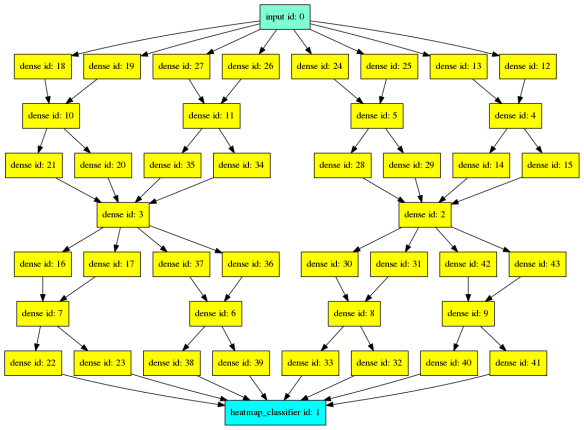


Figure 2.4: REPEAT example. After the third repeat.

2.4.2 Operator FOR

It is the second operator that supports the notion of indirect encoding. Unlike the REPEAT operator, the FOR operator is not fully recursive, because it is designed to emulate a **sequential processes**. Both operators work on the graph edge encoding – i.e. they take as an input one particular edge and the expression, which will be executed on that edge. Mentioned sequential process nature of the FOR operator means, that it takes an input edge, then it splits that edge by creating a **new void block** and naturally two new edges (and deleting the original edge). After that, the input expression is executed on the first newly created edge. Based on the number of iterations, the whole process is repeated from the second new edge. Moreover, unlike in the case of REPEAT operator or any other operator, there is possible to see the scope of the used variables throughout the whole FOR loop expression. It means that the variable scope from the first iteration is transferred after its (possible) modification to the second iteration and so forth – just as if the scope of variables lived in the *sequential* nature of the program execution, which contradicts the natural *functional* nature of the program execution.

Basically there are two types of program execution – first type of execution which is typical for *functional* programming is characterized by the fact, that there is nothing like

a *sequential* execution – each operation is just a function call on a set of its arguments, there is nothing like global variables and the scope of variables is strictly limited only to current called function and no function can have a side effect. Second type of execution is perhaps more familiar sequential or imperative execution, where the program expressions can be given in the sequence, there exists global variables and functions with side effects. And the FOR operator is an effort to bring an imperative notion of programming in to the designed edge encoding programming language.

Algorithm Operator FOR. Function **create_graph_phenotype** is the edge encoding interpreter function. It takes as the parameters: 1) Input edge. 2) Expression which is applied to that edge. 3) Variables set in the form of key value mapping.

```
node_up_orig = input_edge.node_from
node_down_orig = input_edge.node_to

for i = 1..n do
    if i > 1 then
        node_a = GraphNode('concatenate_layer')
    else
        node_a = GraphNode('help_node')
    end if
    node_b = GraphNode('help_node')
    node_list += [node_a, node_b]
    help_edge = Edge(node_a, node_b)
    node_a.output_edges.add(help_edge)
    node_b.input_edges.add(help_edge)

    create_graph_phenotype(help_edge, input_expression, variables)
end for

connect_nodes(node_up_orig, node_list, node_down_orig)

node_up_orig.output_edges.remove(input_edge)
node_down_orig.input_edges.remove(input_edge)
```

Function **connect_nodes**$(node\_up\_orig, node\_list, node\_down\_orig)$ takes the input nodes and connects them to a chain. The $node\_list$ contains the list of pairs $[node_{i,a}, node_{i,b}]$, where the connection is achieved by placing $node_{i+1,a}$ on the place of $node_{i,b}$. Node $node_{i,b}$ is deleted afterwards. Similarly it is dealt with the input node $node\_up\_orig$ which replaces $node_{1,a}$ and $node\_down\_orig$ which replaces $node_{n,b}$.

Here is an example sequential program that is possible to be generated and executed in our programming language for the generation of CNNs. Note its development tree 2.5.

```
X1 := 2
for 3 do
    X1 := X1 · 2
    DOUBLE(SPLIT_DENSE(X1, END, END), SPLIT_DENSE(X1, END, END))
end for
```

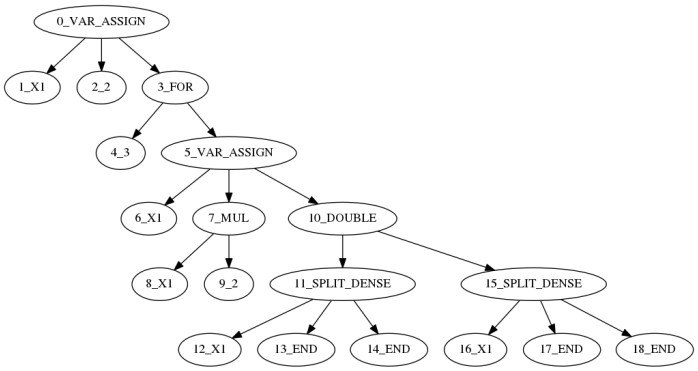


Figure 2.5: Development tree corresponding to the example program.

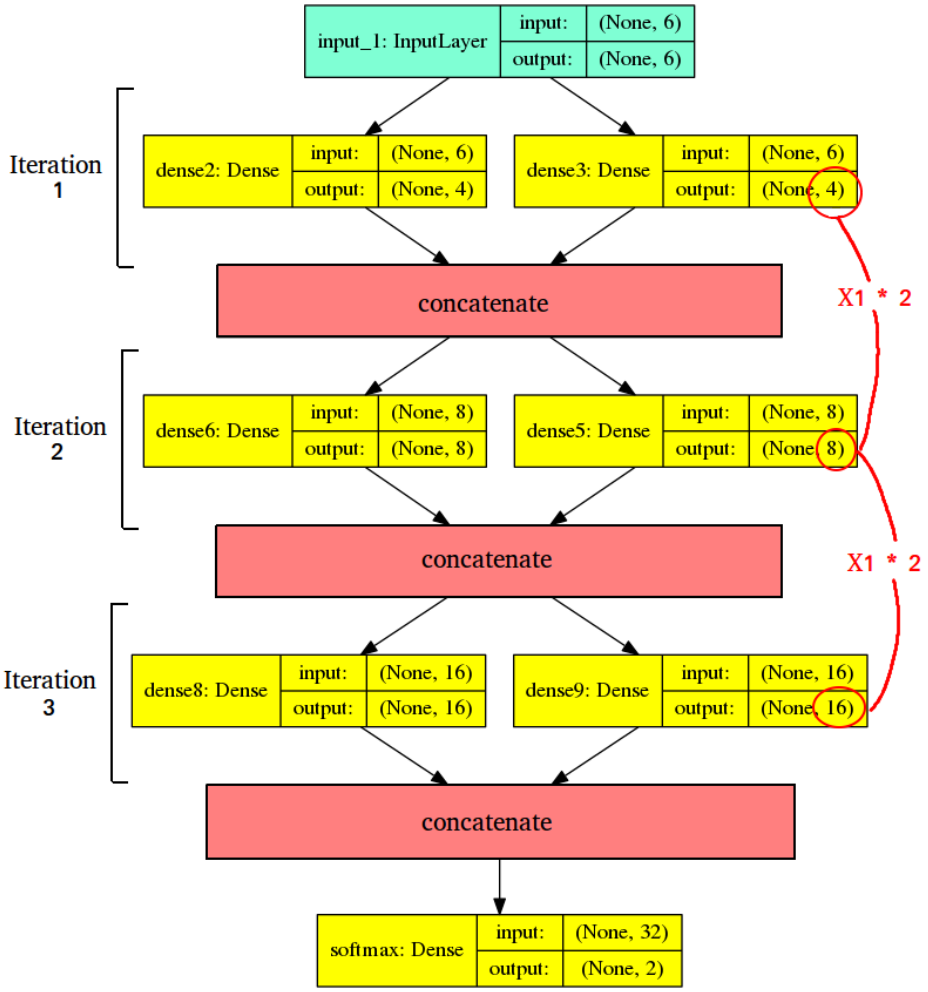


Figure 2.6: Resulting network model phenotype corresponding to the example program. Notice that the number of neurons in layers is dependent on the number of neurons in the previous layer according to the program.

2.5 Used evolutionary algorithm

From a wide variety of existing evolutionary algorithm approaches [14], [20], [22], [23] it was selected a simplified variant of a genetic programming (GP). This variant was selected in order to not complicate already complicated algorithm schema, although it is very likely to be suboptimal in its current form.

Baseline algorithm works with a population of individuals – each individual is a genome that encodes a program which form is being optimized. Number of individuals in the population denotes the parameter *population_size*. Genetic algorithm works in a predefined number of generations or epochs – described by the parameter *gp_training_epochs*. Each generation is processed by firstly evaluating the fitness of all individual in current population. After that there is made a selection of those individuals who will continue to exist in a population to another generation. Based on selection mechanism there is also determined what number and which individuals will be modified (mutated) or bred to enrich the gene pool of current population.

2.5.1 Selections

Selection process determines the set of individuals that will participate in the next generation. Here will be described the simple selection mechanism that was used throughout the whole project. Firstly, there is computed a fitness value for each individual in the population. According to that, the list of individuals is sorted from the best to the worst. Number of individuals is *population_size*. After population sort there are through $elitelist\_proportion\_size \in [0, 1]$ and $individuals\_reproduction\_ratio \in [0, 1]$ specified two parameters n_elite and $n\_reproduction$, where:

$$n\_elite = \lceil population\_size \cdot elitelist\_proportion\_size \rceil$$

$$n\_reproduction = \lceil population\_size \cdot individuals\_reproduction\_ratio \rceil$$

First n_elite individuals of sorted population are inserted to new generation. Then it is necessary to generate $n\_to\_generate = population\_size - n\_elite$ new individuals (either by mutation or crossover). That $n\_to\_generate$ individuals are gradually created by cyclic taking the first $n\_reproduction$ best individuals and mutating them. Cyclic taking is because there is typically $n\_to\_generate > n\_reproduction$ and therefore some individuals must be mutated multiple times.

2.5.2 Population initialization

The process of evolutionary architecture search has to start with some initial population. Each individual must start in some state and there is some space for customization of search. Either the initial individual is generated randomly with possible limitation for such its initial phenotype, or the initial individual can have exactly the same form as any other individual in population by setting its phenotype to some given starting form.

The latter initialization method was used. In the context of this work, the initial individual corresponds to END expression; i.e. the genome [0]. The advantage of this initialization is in the fact, that the program of an individual is evolved gradually from the most trivial expression. It does not evolve from some already absolutely randomly generated source code, which does not have any justification throughout the evolution process.

It cannot be said whether the initialization by random programs would be definitely worse, because we did not try this method due to a limited time / computational resources which was spend for more interesting experiments.

2.6 Mutations

In a context of evolutionary algorithm, the mutation is used for introducing a new feature or at least a change of a given individual. This thesis investigated 5 types of mutations, which are applied during the process of grammatical evolution on a given genome.

1. **Integer flip mutation**. Applied regardless of original genome.
2. **Subtree mutation.** Applied with regard to original genome.
3. **Add twin mutation.** Applied with regard to original genome.
4. **Crossover mutation.** Applied with regard to two parent genomes.
5. **Mutation mixture.** Combination of the previous 4 types.

Each mutation method guarantees to find a syntactically correct program in a first try, but none of them guarantee, that it will find a semantically correct program in a first try. Semantic correctness is achieved by repeated application of used mutation procedure to the input expression for a given amount of time (e.g. several seconds). If the time for a particular mutation run is not sufficient, then the result of a mutation is in practice usually substituted by some trivial individual; e.g. $new\_codon = [0]$.

2.6.1 Integer flip mutation

It is the most straightforward type of mutation applied directly on a genome (list of integers). An input genome with n codons is mutated by randomly changing an integer value of its each codons with probability $1/n$. This type of mutation produces always a syntactically correct program tree (development tree) – because the program itself is synthesized by direct application of codons as instructions for a given grammar and according to the principle of grammatical evolution, no integer value can harm the procedure, as there are used only module remainders of the instruction codon value.

2.6.2 Subtree mutation

Subtree mutation is a more systematic method for mutating the chromosome, because it already does not work completely uninformed. Firstly the input genome is transformed to its development tree (program tree) form. After that, there is uniformly randomly selected one node *selected_node* of a development tree. In the form of development tree, there is already known the type of a *selected_node* (e.g. numeric or boolean expression) – therefore it can be substituted by a new node *new_node* with the same type, but most likely different form. The parameter *new_random_subtree_depth* controls the maximum depth of such *new_node*. Node *new_node* is generated with random uniform distribution with only one constraint, that the first expression shares the same type as a *selected_node*. After the substitution of a subtree expression, the development tree is transformed back to its corresponding genome form, which is yielded as a result of a mutation of the input genome.

2.6.3 Add twin mutation

This mutation type is even more informed method of mutation. It was designed by the author of this thesis to mitigate some issues with unsatisfactory performance of previous mutation methods. This special type of mutation proved to be very successful in the experiments, as it significantly sped up the process of population improvement. The specialization of this method is based on the fact, that it mutates only the <expr> type of program instructions, such that they have exactly two <expr> expressions as their arguments; e.g. DOUBLE(<expr>, <expr>) or SPLIT(<expr>, <expr>) (but not e.g. REPEAT(<int>, <expr>) because it has only 1 <expr> argument).

Method works by firstly selecting such suitable expression, that has 2 <expr> parameters A and B. Suppose, that A is a terminal END node and B is a non END <expr> expression. After the selection of such node, the value of B is copied and inserted at the place of node A. This treatment increases the modularity of resulting phenotypes, as there apparently occurs a copy of a genetic material (which could be evolutionary justified) and inserting it to the void genome place.

As in the subtree mutation, the input genome is transformed to its development tree form. Then it is tested, whether a suitable node for add twin mutation even exists. If it does not exists, then it is applied only a subtree mutation.

2.6.4 Crossover

It is the only mutation type, for which there must be two genomes (parent genomes) present in order to carry out a single mutation. Parent genomes are transformed to development trees A and B. Uniformly randomly there are selected nodes $selected\_A$ from A and $selected\_B$ from B. Node $selected\_A$ is then substituted by a copy of node $selected\_B$. Tree A is then transformed back to its corresponding genome form and yielded as a result of mutation.

2.6.5 Mutation mixture

Mutation mixture can be considered as another type of mutation, with a difference, that its procedure is stochastically selected. It is selected according to a probability distribution of application a particular mutation type at a given trial. Random vector that defines the particular probability is selected after a meta optimization process (in practice, these probabilities are specified according to researcher's experience or a limited grid search with a reasonable values).

Biological idea behind the random application of different types of mutations can be explained as a diversity of the world environment, that affects an individual with different effects, that lead his natural spontaneous genome modifications in different ways.

2.7 Fitness function

Evolutionary algorithm needs to have defined a function that specify a score of an individual at a given problem setup. Usually the score (fitness) is being maximized. Its value directly determine, whether an individual will survive to the next generation, or at least selected for mutation, or if the individual will be completely erased from the population.

Fitness function should be the function, that best describes, what makes an individual good at given task. In the problems similar to the ones in this thesis (statistical supervised machine learning), the best simple fitness function is a test accuracy (i.e. the accuracy of a classifier on unseen data).

Sometimes it is necessary to consider more features of an individual and combine them to the single fitness value – e.g. complexity of an individual, architecture features, defacto anything that can be measured. In a case of more complicated fitness functions, there is also necessary to normalize their values across the current population to make the influence of their components unbiased.

Maybe the simplest and also the used fitness function was based on a linear combination of a test accuracy and a modularity measure Q (section 2.8).

$$fitness = \alpha \cdot test\_accuracy + (1 - \alpha) \cdot Q$$

Mentioned normalization of a fitness function components throughout the population was not necessary, because both components have defined their values strictly into the interval [0..1]. Parameter α in the linear combination specify the importance of particular component on a whole fitness function. Its value is specified by experimentator based on his experience or more likely trial and error in the context of hyper parameter optimization.

2.8 Modularity measure

Modularity measure is applied on a particular individual's phenotype (i.e. ANN / graph). It gives the notion of a network (graph) modularity – respectively of how much is the graph structured into recurrent topological structures, fractal structures, repeated subparts, especially something called *scalled–free* property. Alternatively it says how much the graph differs from the random one. The "how much" measure is in the statistic sense, which yields the modularity values to the interval $[0, 1]$.

The task of measuring the modularity is originally the task of measuring the *scale–free* property of a given graph [24]. This property is measured with the Bhattacharyya coefficient B as the overlap between the probabilistic distribution P (the power law distribution) and the probabilistic distribution q (the distribution of the evolved network). Coefficient B is defined as: $B(P, q) = \sum_k \sqrt{P(k)q(k)}$, where $P(k) \propto k^{-\gamma}$, $q(k)$ is the proportion of the evolved network nodes having degree k and γ is an exponent parameter which was selected according to experiments in [24]. This explanation is mentioned rather to emphasize, that the modularity measure is mostly about the measuring the statistical difference between the expected distribution of the nodes groups and the observed distribution. As all of the methods for a modularity measure, that were successful enough to get into the generally reputable group of methods for this kind of problem are based in this principle, we were facing to selection of the methods from the set of similar approaches. One thoroughly described and mainly computationally tractable method was selected; it is described in the following section.

2.8.1 Graph community structure detecting algorithm

Testing the ability to evolve modular networks is formulated identically to the *scale–free* test, with the difference, that we are maximizing the modularity measure Q. For this purpose it is used the so called Graph Community Structure Detecting Algorithm (Newman, 2003 [25], [24]). The method is based on iterative polynomial greedy division of the graph to the parts (communities), while maximizing a current modularity measure Q of an actual division. Procedure falls in the general category of agglomerative hierarchical clustering methods.

Here is a short description of the algorithm. Let e_{ij} be the fraction of edges in the network that connect vertices in group i to those in group j, and let $a_i = \sum_j e_{ij}$. Then:

$$Q = \sum_i (e_{ii} - a_i^2)$$

is the fraction of edges that fall within communities, minus the expected value of the same quantity if edges fall at random without regard for the community structure. If a particular division gives no more within-community edges than would be expected by random chance we will get Q = 0. Values other than 0 indicate deviations from randomness, and in practice values greater than about 0.3 appear to indicate significant community structure.

Chapter 3

Technical details

Chapter contains several very useful descriptions of such issues, that had to be dealt within the context of this project – especially a neural network phenotype of an individual and all problems, that emerged during the work with such complicated data structures that had a considerable amount of constraints.

Based on this and the previous chapter, the reader should be able to easily reimplement the code for all experiments.

3.1 Dealing with phenotype bloat

There is a frequent problem in evolutionary algorithms when synthesizing a new individual either by mutation or by crossover generally. Individual produced by e.g. a genome might produce as complicated phenotype that it could be computationaly intractable. It also often does not even depend on the genome length, because for instance a multiple usage of REPEAT operation can produce extremely large phenotype structure by using only a few instructions. The problem of bloat – gradual unbounded increasing of the size of population phenotypes has in this particular project the following 5 instances:

1. **REPEAT operator restriction.**
2. **Tensor size limit.**
3. **Number of blocks.**
4. **Edge bloat.**
5. **Genome length.**

3.1.1 REPEAT operator restriction

As the application of a single REPEAT operator produces exponencial number of edges in phenotype with respect to the number of repeats, the multiple application of REPEAT operator in itself would lead to the superexponencial number of produced edges. For example, if there is a REPEAT(N, REPEAT(N, REPEAT(N, EXPR))), and the execution of EXPR on a given edge produces only 2 edges, than after that multiple REPEAT expression execution there is $2^{2^{2^N}}$ new edges, which is apparently intractable.

Therefore it was enforced that at most 1 REPEAT occurence can be used on every program tree branch. The check for enforcing this rule can be done in a logarithmic time to a number of program

expressions. As the complexity of only a single REPEAT is exponencial on its own, the parameter $max\_repeats = 3$ was introduced to limit maximum number of REPEATs to only 3.

3.1.2 Tensor size limit

ANNs produced by a program might be semanthically correct, but the sizes of tensors representing layers of a network can be too big to fit in a memory of computer or GPU. Therefore it was introduced a parameter $tensor\_limit = 5 \times 10^8$. The idea behind the $tensor\_limit$ is that its value represents the total sum of every used tensor sizes. As the single tensor value is represented as float32, which is 4 bytes, and for every tensor value is for the need of backpropagation algorithm necessary to use 1 more 4 byte number, then we are talking about neccessity of having at least 8 bytes for 1 tensor value. The value of 5×10^8 was not selected randomly. It corresponds to the assumption, that generated networks will not consume more than 4 GB of GPU memory in total. This amount of memory comfortably possessed every single GPU card that I have been using for the purpose of this project.

3.1.3 Number of blocks

To avoid overcomplication of resulting network phenotype structure, I introduced another parameter $max\_block\_count = 40$. It limits the number of used blocks (or layers) in the phenotype. The idea is to encourage some effectiveness of new block usage and reduce the occurence of such individuals that have a good fitness only because they used a high number of chaotically connected blocks – i.e. the same result could be most likely achievable by better network topology.

3.1.4 Edge bloat

Last bloat controlling parameter is $edge\_bloat\_limit = 25$. It is used in a context of the FOR operator and tells the maximal number of new edges that are produced from a single edge. There is no prohibition of nested FOR operators as it was at the REPEAT operator. The only limit is the number of new edges produced from the original one. For example FOR(4, FOR(6, SPLIT(EXPR, EXPR))) (where EXPR produces exactly 1 new edge) is a valid expression, because it produces only 24 new edges which is in the limit.

3.1.5 Genome length

This is a straightforward bloat limit based on enforcing the genome to have at most $max\_genome\_length = 150$ codons. This is suitable to avoid having too long programs, which can easily emerge during the mutation – especially crossover.

3.2 Wrapping

Wrapping in the context of evolutionary algorithms is the designation for a method which is used for determination of the resulting phenotype from the input genome in a case when the input genome runs out of genetic material. For example let us have a situation, where the current string is equal to "expr" and let us have a genome with only 1 codon – this single codon encodes for example the rule which rewrites "expr" -> "SPLIT(expr, expr)". As that rewrite is carried out, we are left with unclosed expression, because "expr" is not a terminal symbol. Problem of wrapping deals with the question of how to make the resulting expression closed.

Popular approach is to cyclically use the given genome until there is a closed expression. This can lead to problems as some genomes (e.g. the one mentioned here) can never yield a closed expression. There is not any common notation of how to deal with this issue and therefore it must be done with respect to the particular task.

Used approach uses a cyclic wrapping with the customization, that if there is an expression "expr", then it is rewrited into an "END" expression. And if the expression is an "int" then it is rewritten to "int_digit". Another recursive expressions such as "expr" and "int" are not present – therefore this treatment is sufficient.

3.3 Integer flip mutation modulo bias

In the previous chapter it was described the mechanism of grammatical evolution, where the codons of an input genome are used as indices for a particular rule selection. When the integer flip mutation is applied over the genome, it is necessary to generate all possibilities with equal probability. As it was mentioned, when applying the index of a rule to select a particular one, it is used a remainder after division by the number of current rewriting rules possibilities. New random value, that facilitates the mutation of the codon must be therefore randomly generated to the interval $[0, lcm([s_1, s_2, .., s_n]) - 1]$ where s_i is the number of rewriting rules of the symbol i and lcm is the least common multiplier of the list of numbers.

Chapter 4

Experiments and Results

Experiments took place on the CESNET Metacentrum, the grid computer environment IT4 Innovations in Ostrava, and my immodest computer with CPU Intel i5-6600 3.30 GHz, 32 GB RAM and GPU GeForce GTX 1070 with 8 GB memory. The complications with the computational resources came in the very early phase of experimenting, because the project on IT4 grid was canceled and I was left with only the CESNET Metacentrum, which was not as suitable for GPU experiments as the IT4 grid was. Later I was redirected to try a Google Colab project, which offers for free a huge amount of computational resources – especially Nvidia Tesla K80 GPU cards. The main issue with working on Google Colab was, that the work there was meant as a Python notebook project in the tab of the Internet browser, which had to be always open without any possibility to run tasks in a batch mode or in the background with closed browser session. Moreover there were frequently issues with authentication of the Google account which was necessary to have, and these issues had to be dealt manually which made running of my experiments very tedious and in the result I sticked mainly to the Metacentrum which was quite sufficient for my needs.

As the fitness computation per individual had very high computational requirements, it was necessary to make several implementation adjustments, that saved the results of an individual's program and his setup context to some large lookup table, which was queried each time the new individual was about to be evaluated. Implementation was carried out in Python 3.* language. As a very useful tool was used the Keras library with the Tensorflow backend. The whole written code for both the experiments and their analysis had approximately 2000 lines of code.

4.1 Model hyperparameter optimization

In order to achieve possibly the best results from the baseline evolutionary algorithm, there was executed a limited search over a set of suitable parameter configurations. Found configuration was used for XOR–N and CIFAR–10 experiments as well. The baseline parameter setup which was used for hyperparameter optimization was the following:

$\langle s \rangle ::= \langle END \rangle$
| $\langle DOUBLE \rangle \langle s \rangle \langle s \rangle$
| $\langle REPEAT \rangle \langle int \rangle \langle s \rangle$
| $\langle SPLIT\_DENSE \rangle \langle int \rangle \langle s \rangle \langle s \rangle$

$\langle int\_digit \rangle ::= 0 \mid 1 \mid 2 \mid 4 \mid 8 \mid 16$

$\langle int \rangle ::= \langle int\_digit \rangle$

```
dataset: XOR
loss: categorical_crossentropy
optimizer: adam
edge_bloat_limit: 25
input_dimension: 4
grammar_lcm: 12
new_random_subtree_depth: 2
```

```
gp_training_epochs: 50
population_size: 50
xor_dataset_size: 50000

elitelist_proportion_size: 0.15
individuals_reproduction_ratio: 0.5
xor_dataset_dimension: 4

mutation_type: all_mutation_mixture
max_genome_length: 150
ann_training_epochs: 50
n_repeats: 3
```

4.1.1 ANN training parameters setup

Training of ANNs is controlled by several thoroughly studied concepts that directly influence the way of how the ANN weights are adjusted to learn desired features. These concepts are believed to be familiar to the reader, therefore it will be discussed only the directly relevant notes and enumerated only the particular methods which were used. The ANN training setup was preserved in all experiments, although there could be likely made some domain specific parameter settings improvements. After considering the limited computational resources and the observations of common discussion we used this settings:

1. **Loss function**: categorical crossentropy. Forcing the CNN to have as many output neurons as was the number of K classes was achieved by using an additional 1×1 convolution layer with K filters, which was connected to the global max pooling layer that provided the final value of CNN loss function.
2. **Gradient optimizing method**: Adam optimizer method [26]. Learning rate = 0.002 without decay. $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1.0 \times 10^{-8}$.
3. **Network initialization**: Glorot uniform method.
4. **Dataset augmentation**: no special method used; only the original described datasets. In the final experiment for evaluating the best individual it was used an augmentation described in 4.4.1.
5. **Cross validation & Number of repeats**: Due to the limited computational resources it was used only the original validation dataset to provide test accuracy; i.e. no cross validation was used. In both the XOR and CIFAR experiments, the whole dataset was split into the train set (83 %) and test set (17 %). The number of CNN training repeats was set only to 1, which can make the estimate of model test accuracy biased. This issue was confronted rather by running the whole experiments multiple times, then to precisely estimate the individuals test accuracy.
6. **Stopping function**: ANN training was stopped with *patience* = 2; i.e. when the test error did not improved on a third learning epoch, the learning was terminated.
7. **Learning epochs**: On all experiments it was set to 50.
8. **Batch size**: 256 (it also determined the amount of the GPU memory to allocate for the Keras CNN model). In the final experiment for evaluating the best individual the batch size was 512.

4.1.2 Mutation probabilities selection

After some experimenting the focus was aimed into these two configurations:

```
elitelist_proportion_size: 0.2 vs 0.15,
individuals_reproduction_ratio: 0.4 vs 0.5
```

Finally, the settings:

```
elitelist_proportion_size: 0.15,
individuals_reproduction_ratio: 0.5
```

was selected. It means, that 50% of the population participates in the selection phase (with respect to previous 40%) and only 15% of the best individuals are copied to the next generation (with respect to the previous 20%). This setup broadens the variety of population with the slight drawback of possible slower convergence. However the broader search over the program state space is more desirable than its faster convergence, therefore the second option was selected.

4.1.3 Mutation method selection

Mutation operator investigation was carried out over the settings with better population selection probabilities. It showed out, that these methods of mutation were better with respect to one another:

$$int\_flip \prec subtree \prec add\_twin\_subtree + subtree \preceq crossover + subtree$$

$$crossover + subtree \prec add\_twin\_subtree + crossover + subtree$$

The mutation method int_flip was omitted and was not used in the final experiments at all. It does make sense, because the int_flip mutation is completely uninformed and a single change of a codon in chromosome may result in completely different individual, which will have most likely worse performance.

Moreover the mixed mutation procedure is more likely to bring better variety as it combines all mutation approaches together and at the same place it has a better modularity emerging capabilities with only slight precoded procedures as in the add_twin_subtree method. Mutation mixture used 70 % of subtree mutation, 15 % of add_twin mutation and 15 % of crossover.

4.2 XOR dataset experiments

The idea of experimenting over XOR dataset which is on the first glance quite easy to correctly solve, is that the proper solution should contain some modularity. The XOR problem itself can be considered as the problem of an even parity. When the binary input contains an even number of ones, than it is classified as 1, otherwise it is classified as 0. One can decompose the classification of an input word to subtasks, solve those subtasks and compose results of those subtasks to obtain the result to report. For example the subtask which can be identified from the human perspective, can be to compare, whether the first bit of an input is equal to the second bit of the input, and then replace the second bit with the previous answer and repeat it over the whole input word and finally report a bit value of the last compared identity. After all, this is the essence of logical circuits designing, which have apparently some functional modules (gates) structure.

Our speculation was that the grammatical evolution over an edge encoding on XOR dataset, may be able to find programs, which produce such individuals, whose structure resemble some non chaotic hierarchical structure.

An important implementation detail might be, that the generated ANN individuals were not composed from CNNs but from classic fully connected ANNs. Although the layered ANN can be emulated by CNN by using 1×1 convolution, for the reasons of project development clarity there was added a support for dense layers and thus for all the classic fully connected ANNs.

XOR datasets themselves were generated in a way that facilitates the deployment of neural network classifier by introducing some random noise to dataset samples. Data sample for XOR–N was an N dimensional vector v, where $v_i \in [0, 1]$ had the semantic such that values $v_i \in [0, 0.5)$ corresponded to 0 and values $v_i \in [0.5, 1]$ corresponded to 1. The real truth value of a XOR–N data sample s was simply determined after denoising the s and checking an even parity in the respective binary valued sample.

Dataset	Training set samples	Test set samples
XOR–4	42 k	8 k
XOR–6	100 k	20 k
XOR–8	165 k	35 k

Table 4.1: Sizes of data samples corresponding to a particular XOR dataset type.

4.2.1 Investigated grammars

Here is an overview of investigated types of grammars, which have been used in the phase of experimenting on top of the XOR problem.

4.2.1.1 Full grammar [A-setup]

There is apparently no doubt, that a full grammar with the sufficient time and computational resources would give the best results, because the space of programs generated by full grammars contains all programs of the aforesaid grammars; i.e variables, conditional expressions, recursive integer values definition, arithmetic operators and boolean expressions.

$\langle expr \rangle ::= \langle END \rangle$
| $\langle DOUBLE \rangle \langle expr \rangle \langle expr \rangle$
| $\langle REPEAT \rangle \langle int \rangle \langle expr \rangle$
| $\langle FOR \rangle \langle int \rangle \langle expr \rangle$

| ⟨*VAR_ASSIGN*⟩ ⟨*int_var*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*IF_ELSE*⟩ ⟨*bool*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*SPLIT_DENSE*⟩ ⟨*int*⟩ ⟨*expr*⟩ ⟨*expr*⟩

⟨*bool*⟩ ::= ⟨*bool_op*⟩ ⟨*int*⟩ ⟨*int*⟩

⟨*bool_op*⟩ ::= ⟨*gt*⟩ | ⟨*ge*⟩ | ⟨*lt*⟩ | ⟨*le*⟩

⟨*int*⟩ ::= ⟨*int_digit*⟩
| ⟨*int_var*⟩
| ⟨*int_op*⟩ ⟨*int*⟩ ⟨*int*⟩

⟨*int_digit*⟩ ::= 0 | 1 | 2 | 4 | 8 | 16

⟨*int_op*⟩ ::= ADD | SUB | MUL

⟨*int_var*⟩ ::= X1 | X2 | X3 | X4

4.2.1.2 Minimalistic [B-setup]

It is the smallest type of used grammar, that contains only the necessary components for generating DNNs. Operator FOR/REPEAT is necessary to make a support for an indirect encoding functionality of produced programs.

⟨*s*⟩ ::= ⟨*END*⟩
| ⟨*DOUBLE*⟩ ⟨*s*⟩ ⟨*s*⟩
| ⟨*REPEAT*⟩ ⟨*int*⟩ ⟨*s*⟩
| ⟨*FOR*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*SPLIT_DENSE*⟩ ⟨*int*⟩ ⟨*s*⟩ ⟨*s*⟩

⟨*int_digit*⟩ ::= 0 | 1 | 2 | 4 | 8 | 16

⟨*int*⟩ ::= ⟨*int_digit*⟩

4.2.1.3 Minimalistic & recursive num expr [C-setup]

This grammar is also as small as possible, with the difference of including recursive definition of integer values. This modification allows to greatly expand the space of used numeric expressions, which is necessary for declaring larger integer values.

⟨*expr*⟩ ::= ⟨*END*⟩
| ⟨*DOUBLE*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*REPEAT*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*FOR*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*SPLIT_DENSE*⟩ ⟨*int*⟩ ⟨*expr*⟩ ⟨*expr*⟩

⟨*int_digit*⟩ ::= 0 | 1 | 2 | 4 | 8 | 16

⟨*int*⟩ ::= ⟨*int_digit*⟩
| ⟨*int_op*⟩ ⟨*int*⟩ ⟨*int*⟩

⟨*int_op*⟩ ::= ADD | SUB | MUL

4.2.1.4 Minimalistic & VARIABLES & recursive num expr [D-setup]

It is very similar to the previous **C** setup with extra expressions for using the integer variables. Those are namely the 4 variables X1, X2, X2, X2 of which the grammatical evolution process can choose from. Purpose of variables is again for the expansion of the space of numerical expressions.

⟨*expr*⟩ ::= ⟨*END*⟩
| ⟨*DOUBLE*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*REPEAT*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*FOR*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*VAR_ASSIGN*⟩ ⟨*int_var*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*SPLIT_DENSE*⟩ ⟨*int*⟩ ⟨*expr*⟩ ⟨*expr*⟩

⟨*int_digit*⟩ ::= 0 | 1 | 2 | 4 | 8 | 16

⟨*int*⟩ ::= ⟨*int_digit*⟩ | ⟨*int_var*⟩ | ⟨*int_op*⟩ ⟨*int*⟩ ⟨*int*⟩

⟨*int_op*⟩ ::= ADD | SUB | MUL

⟨*int_var*⟩ ::= X1 | X2 | X3 | X4

4.2.1.5 Minimalistic & VARIABLES & IF_ELSE [E-setup]

This grammar did not use compounded numerical expressions, but only the variables. Moreover there is a support for the investigation of conditional expression, which however in the end did not prove to be particularly beneficial.

⟨*expr*⟩ ::= ⟨*END*⟩
| ⟨*DOUBLE*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*REPEAT*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*FOR*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*VAR_ASSIGN*⟩ ⟨*int_var*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*IF_ELSE*⟩ ⟨*bool*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*SPLIT_DENSE*⟩ ⟨*int*⟩ ⟨*expr*⟩ ⟨*expr*⟩

⟨*int_digit*⟩ ::= 0 | 1 | 2 | 4 | 8 | 16

⟨*int*⟩ ::= ⟨*int_digit*⟩ | ⟨*int_var*⟩

⟨*int_var*⟩ ::= X1 | X2 | X3 | X4

4.2.2 Results of the XOR experiments

The question is, what grammar is the most effective by the terms of required computational time for finding the best program which will generate the best network. The comparison is therefore based on investigating the quality of a solution reached after the same number of epochs and the same size of a population.

Parameter settings of each experiment was exactly the one from section 4.1 (Model hyperparameter optimization) only with the replacement of used grammar. Experiments themselves were 10 times repeated, because each experiment run gave different results. As a reported measure for the quality of an experiment setup, was selected only the test accuracy (despite the fact, that during the training it was used fitness function which considered also a modularity measure).

XOR experiments were executed on 4, 6, 8 vector sizes. But for the resulting comparison, only the XOR–6 problem was investigated. Experiment dataset sizes are in the table 4.1.

For a better clarity, the experiments setup were labeled as:

1. **A-f** (4.2.1.1) Full grammar with FOR operator.
2. **A-r** (4.2.1.1) Full grammar with REPEAT operator.
3. **B-f** (4.2.1.2) Minimalistic with FOR operator.
4. **B-r** (4.2.1.2) Minimalistic with REPEAT operator.
5. **C-f** (4.2.1.3) Minimalistic & recursive num expr with FOR operator.
6. **C-r** (4.2.1.3) Minimalistic & recursive num expr with REPEAT operator.
7. **D-f** (4.2.1.4) Minimalistic & VARIABLES & recursive num expr with FOR operator.
8. **D-r** (4.2.1.4) Minimalistic & VARIABLES & recursive num expr with REPEAT operator.
9. **E-f** (4.2.1.5) Minimalistic & VARIABLES & IF_ELSE with FOR operator.
10. **E-r** (4.2.1.5) Minimalistic & VARIABLES & IF_ELSE with REPEAT operator.

As it will be shown on the following boxplots, the best experiment setup is the setup **C–f** which corresponds to the *minimalistic grammar with recursive numeric expressions and FOR operator*.

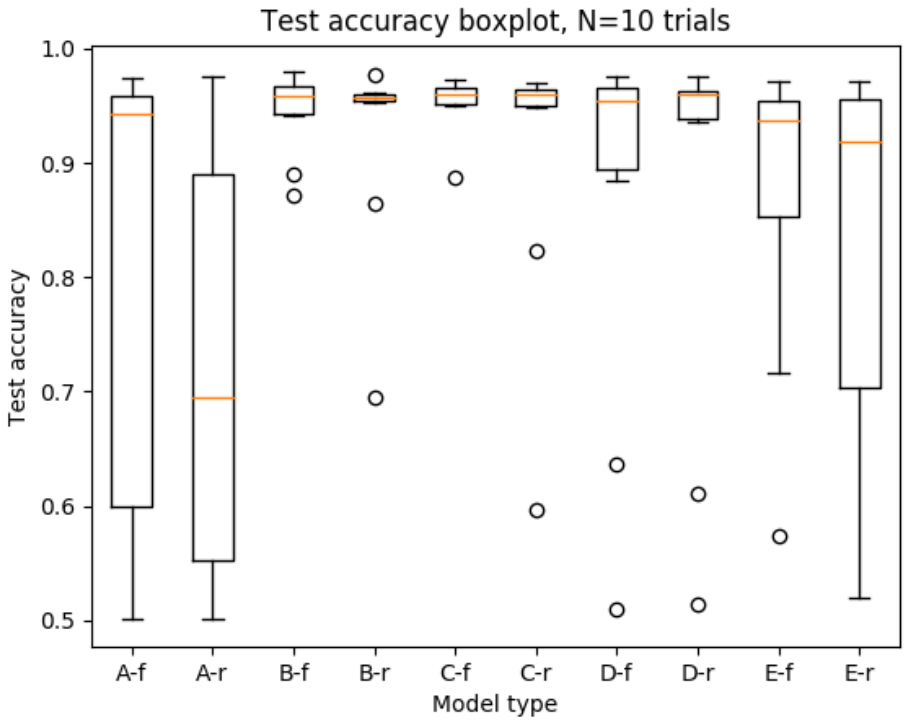


Figure 4.1: Boxplot overview of $N = 10$ best **test accuracy** achieved per model. All experiment trials had the same population size and the generations count. Achieved test errors are similar regardless the usage of FOR or REPEAT operator (except at the experiment **A** where the FOR operator was significantly more successful). Apparently the most extensive setup **A** was too difficult to optimize in a reasonable time and thus reached the worst performance (especially with REPEAT operator). Setups **B** and **C** had very small variance of the best test accuracy, whereas setups **D** and **E** which were using variables performed significantly worse. Evidently the usage of variables does not bring any advantage and only decreased the performance of GP optimizer by increasing the size of the space of programs.

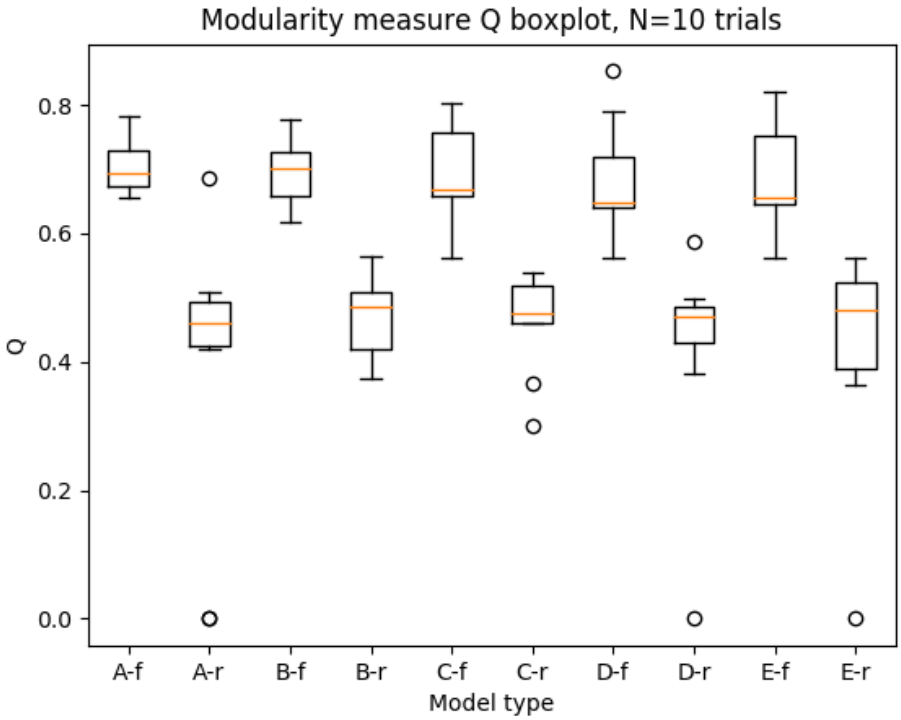


Figure 4.2: Boxplot overview of $N = 10$ best Q measure achieved per model. All experiment trials had the same population size and the generations count. Apparently the FOR operator created more modular architectures then the REPEAT operator. Interestingly the Q value does not apparently depend on the type of used setup. It is probably because the crucial influence on the Q value has the modularity operators themselves, which were used the same across all of the setups.

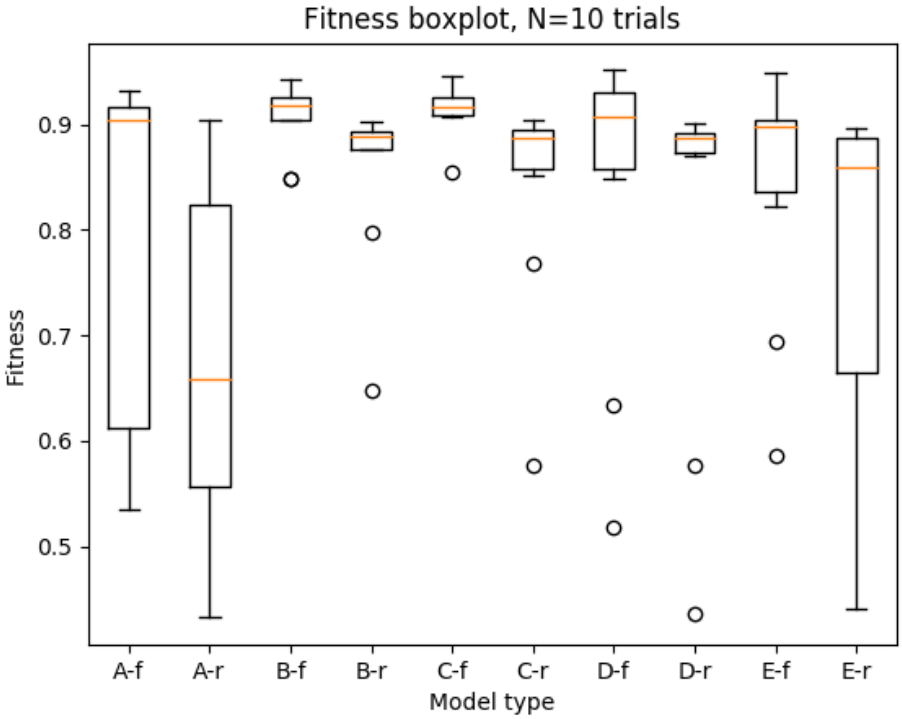


Figure 4.3: Boxplot overview of $N = 10$ best **fitness** achieved per model. All experiment trials had the same population size and the generations count. Value of the fitness function was a linear combination of test error value (weighted 0.85) and Q modularity value (weighted by 0.15). Fitnesses were typically better for FOR operator, but it is important to notice that this was achieved by the better Q value (viz boxplot 4.2) and almost the same test error value (viz boxplot 4.1). The main reason for this boxplot was to investigate, whether there were some benefits for prioritizing the amount of modularity values for the individuals, but when the modularity values proved to be almost the same across all setups (4.2), it makes this boxplot quite redundant.

4.2.2.1 Example XOR experiment case study

Following 3 graphs show some properties of the model **C**–**f** evaluation. Each individual during the **C**–**f** experiment reached some value of test error, modularity measure Q, $test\_error$ and $comlexity$; all of these records were recorded in the database alongside with the number of epoch when the individual was found for the first time.

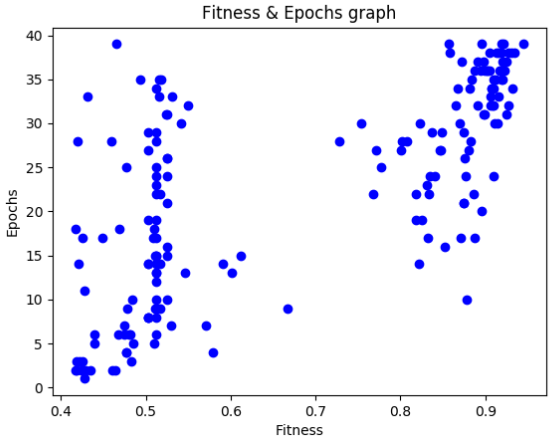


Figure 4.4: Population individuals $fitness$ & $epochs$ graph in **C**–**f** experiment.

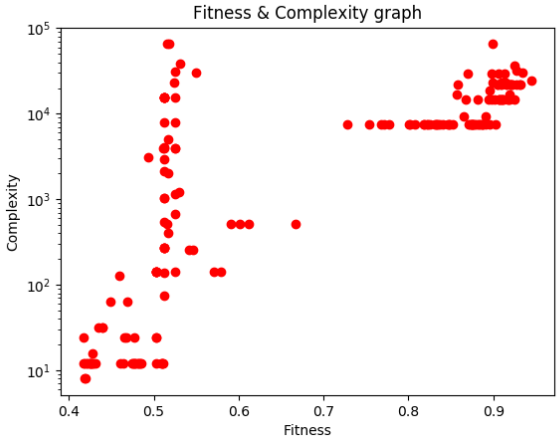


Figure 4.5: $fitness$ & $complexity$ graph in **C**–**f** experiment. Rendered dots follows approximately the XOR–6 learning curve. Vertical line at the $fitness \approx 0.5$ are the results of very inefficient architectures.

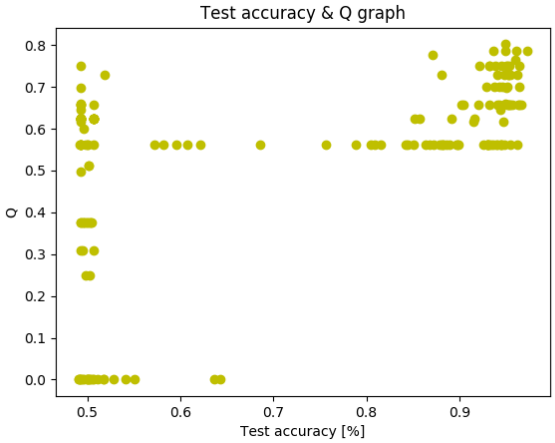


Figure 4.6: $test\_accuracy$ & Q graph in **C**–**f** experiment. Horizontal line of plotted yellow points might mean, that many distinct individuals had exactly the same Q value, which is not unusual, as the Q value depends on graph topology, not on all details of individuals source code.

4.2.2.2 XOR experiment example individual

This individual proved to be the one with the highest test accuracy. Its chromosome was:

```
[6, 2, 10, 3, 7, 8, 0, 9, 0, 10, 9, 4, 9, 2, 0, 8,
 3, 10, 3, 0, 4, 9, 2, 0, 8, 0, 9, 2, 0, 8, 4, 4, 0]
```

Its development tree was somewhat redundant, because it contained several expressions that did not have any influence on the resulting phenotype – for example a for loop that loops only an END expression (which does nothing).

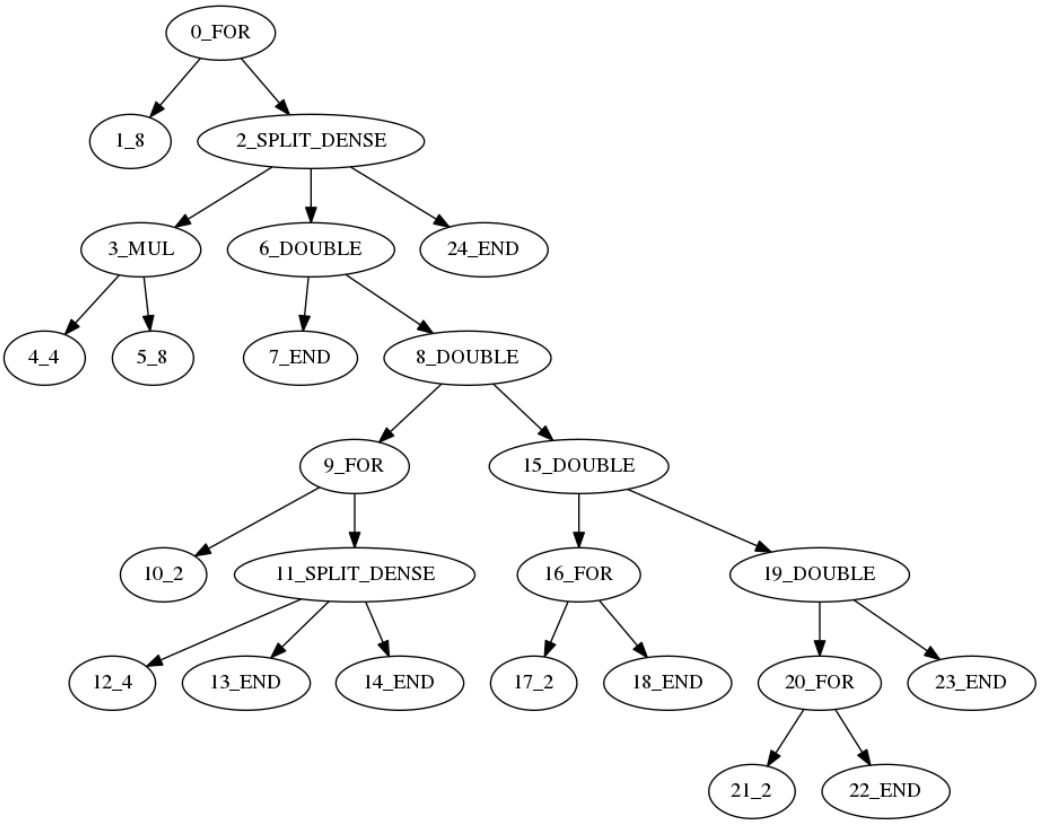


Figure 4.7: Development tree. It is a source code of the best individual of the **C–f** experiment. It contains 8 times repeated main FOR loop expression and the values of neurons were increased by multiplication which generated greater integer value, then would be the maximal value of the grammar without defined recursive integer expressions.

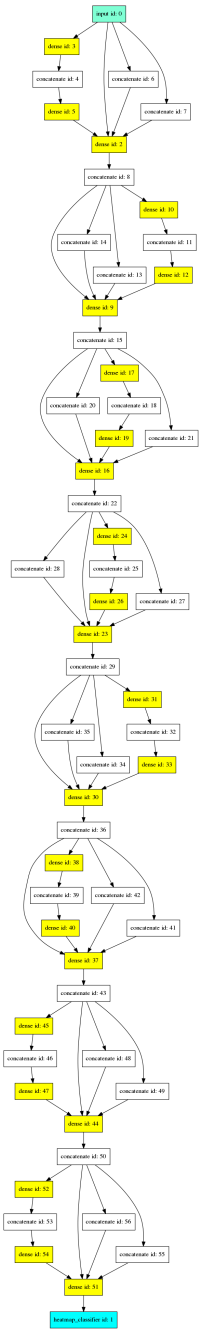


Figure 4.8: A network phenotype graph topology of the best **C–f** individual. This graph is created by applications of the development tree instructions an a single starting edge.

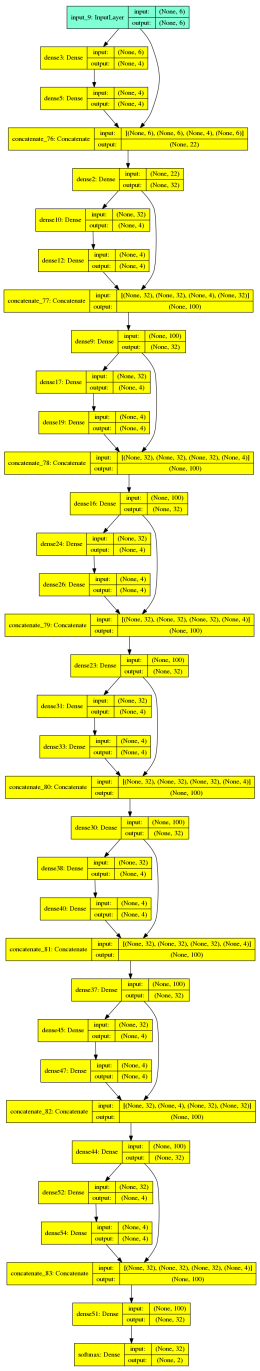


Figure 4.9: ANN produced from the same graph phenotype. The FOR operator had a tendency to create more traditional deep neural networks architectures in contrast with REPEAT operator that produced typically broader and visually more interesting architectures (although from the perspective of test error the FOR operator was better).

4.3 CIFAR dataset experiments

The decision to carry out a significant part of experiments on the CIFAR–10 dataset [27] came from its wide familiarity amongst the visual ML labeled datasets; i.e. it serves as a benchmark for studying the performance of classifiers. At the same time, there is a reason to belief, that the deploying of highly modular architecture design of ANN classifier can help the network to identify (from the human perspective) very distinct $K = 10$ classes in visual data. In terms of scalability, the right solution should be able to quickly adapt itself from the CIFAR–10 to the CIFAR–100 classification problem, with the usage of the same network architectural features.

The CIFAR–10 dataset contains 50000 of 32×32 RGB images. Those are represented as a $50000 \times 32 \times 32 \times 3$ tensors with bytes values. Before the training the values of tensors are normalized into to the float valued interval $[-1, 1]$. There is $K = 10$ target classes that corresponds to real life objects such as: cats, dogs, trucks, ships, etc.

Methodology for the experiments was very similar to one practiced in the XOR–N experiments; several variants of grammars were proposed and their performance mutually compared on provided boxplot diagrams. For the illustration, there is also provided an example CNN individual.

4.3.1 Investigated grammars

Due to the problems with large computational demands, we limited our investigation to 3 basic types of grammars (with FOR and REPEAT variants), that were believed to be the most interesting.

4.3.1.1 Full grammar [A-setup]

It is the most general type of grammar with all mentioned expressions for the usage of variables, compounded numeric expression, boolean and conditional expressions and pooling/convolutional CNN layers.

⟨*expr*⟩ ::= ⟨*END*⟩
| ⟨*DOUBLE*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*FOR*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*REPEAT*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*IF_ELSE*⟩ ⟨*bool*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*VAR_ASSIGN*⟩ ⟨*int_var*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*SPLIT_CONV*⟩ ⟨*filter_size*⟩ ⟨*n_filters*⟩ ⟨*stride*⟩ ⟨*padding*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*SPLIT_POOL*⟩ ⟨*filter_size*⟩ ⟨*stride*⟩ ⟨*padding*⟩ ⟨*expr*⟩ ⟨*expr*⟩

⟨*int*⟩ ::= ⟨*int_digit*⟩
| ⟨*int_var*⟩
| ⟨*int_op*⟩ ⟨*int*⟩ ⟨*int*⟩

⟨*bool*⟩ ::= ⟨*bool_op*⟩ ⟨*int*⟩ ⟨*int*⟩

⟨*bool_op*⟩ ::= ⟨*gt*⟩ | ⟨*ge*⟩ | ⟨*lt*⟩ | ⟨*le*⟩

⟨*int_digit*⟩ ::= 0 | 1 | 2 | 4 | 8 | 16

⟨*int_op*⟩ ::= ADD | SUB | MUL

⟨*filter_size*⟩ ::= 1×1 | 3×3 | 5×5

⟨*n_filters*⟩ ::= int

⟨*stride*⟩ ::= 1
| 2

⟨*padding*⟩ ::= same
| valid

⟨*int_var*⟩ ::= X1 | X2 | X3 | X4

4.3.1.2 Minimalistic [B-setup]

It is the smallest type of used grammar, that contains only the necessary components for generating CNNs. Operator FOR/REPEAT is necessary to make a support for an indirect encoding functionality of produced programs.

⟨*expr*⟩ ::= ⟨*END*⟩
| ⟨*DOUBLE*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*FOR*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*REPEAT*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*SPLIT_CONV*⟩ ⟨*filter_size*⟩ ⟨*n_filters*⟩ ⟨*stride*⟩ ⟨*padding*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*SPLIT_POOL*⟩ ⟨*filter_size*⟩ ⟨*stride*⟩ ⟨*padding*⟩ ⟨*expr*⟩ ⟨*expr*⟩

⟨*int*⟩ ::= ⟨*int_digit*⟩

⟨*int_digit*⟩ ::= 0 | 1 | 2 | 4 | 8 | 16

⟨*filter_size*⟩ ::= 1×1 | 3×3 | 5×5

⟨*n_filters*⟩ ::= int

⟨*stride*⟩ ::= 1
| 2

⟨*padding*⟩ ::= same
| valid

4.3.1.3 Minimalistic & recursive num expr [C-setup]

This grammar is also as small as possible, with the difference of including recursive definition of integer values. This modification allows to greatly expand the space of used numeric expressions, which is necessary for declaring larger integer values.

⟨*expr*⟩ ::= ⟨*END*⟩
| ⟨*DOUBLE*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*FOR*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*REPEAT*⟩ ⟨*int*⟩ ⟨*expr*⟩
| ⟨*SPLIT_CONV*⟩ ⟨*filter_size*⟩ ⟨*n_filters*⟩ ⟨*stride*⟩ ⟨*padding*⟩ ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*SPLIT_POOL*⟩ ⟨*filter_size*⟩ ⟨*stride*⟩ ⟨*padding*⟩ ⟨*expr*⟩ ⟨*expr*⟩

⟨*int*⟩ ::= ⟨*int_digit*⟩
| ⟨*int_op*⟩ ⟨*int*⟩ ⟨*int*⟩

⟨*int_digit*⟩ ::= 0 | 1 | 2 | 4 | 8 | 16

⟨*int_op*⟩ ::= ADD | SUB | MUL

⟨*filter_size*⟩ ::= 1×1 | 3×3 | 5×5

⟨*n_filters*⟩ ::= int

⟨*stride*⟩ ::= 1
| 2

⟨*padding*⟩ ::= same
| valid

4.3.2 Results of the CIFAR experiments

For a better clarity, the experiments setup were labeled as:

1. **A-f** (4.3.1.1) Full grammar with FOR operator.
2. **A-r** (4.3.1.1) Full grammar with REPEAT operator.
3. **B-f** (4.3.1.2) Minimalistic with FOR operator.
4. **B-r** (4.3.1.2) Minimalistic with REPEAT operator.
5. **C-f** (4.3.1.3) Minimalistic & recursive num expr with FOR operator.
6. **C-r** (4.3.1.3) Minimalistic & recursive num expr with REPEAT operator.

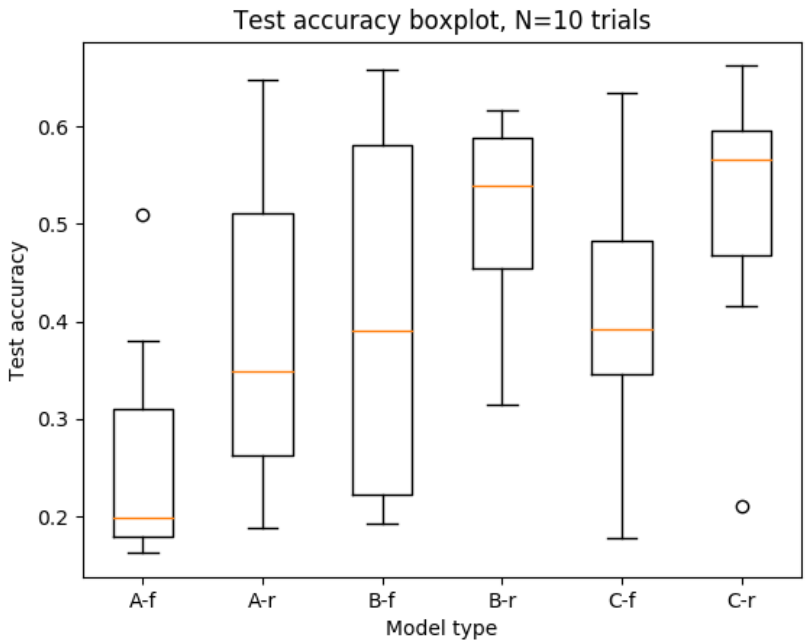


Figure 4.10: Boxplot overview of $N = 10$ best **test accuracy** achieved per model. All experiment trials had the same population size and the generations count. Unlike in the case of XOR problem, in all experiments the average fitness was better when the modularity operator was REPEAT instead of FOR. In terms of the best average test accuracy per experiment, the setup **C-r** was the best (minimalistic model with recursive numerical expressions and REPEAT operator). Apparently the most extensive setup **A** was too difficult to optimize in a reasonable time and thus it reached the worst performance. Setups **B** and **C** differs mainly in the achieved variance of the best test accuracy. The setup **C** that could adjust its integer constant values arbitrarily had the variance slightly lower.

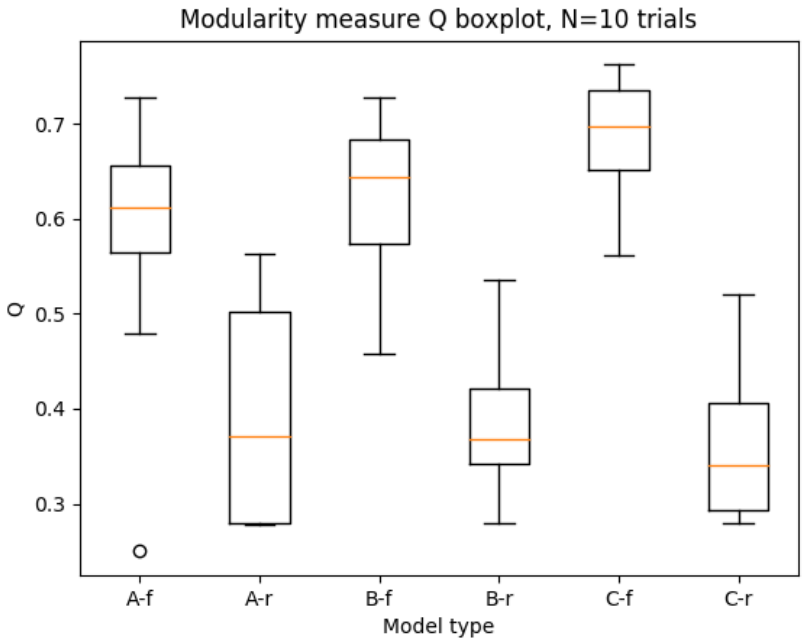


Figure 4.11: Boxplot overview of $N = 10$ best Q **measure** achieved per model. All experiment trials had the same population size and the generations count. As well as in the case of XOR experiments, the FOR operator created more modular architectures then the REPEAT operator. Interesting fact to notice is that the modularity measures for REPEAT best experiments were significantly smaller, although those experiments reached better test accuracies, then their FOR counterparts. It may bring an insight, that the benefits of having highly modular architecture does not guarantee better test performance. It is apparently domain dependent.

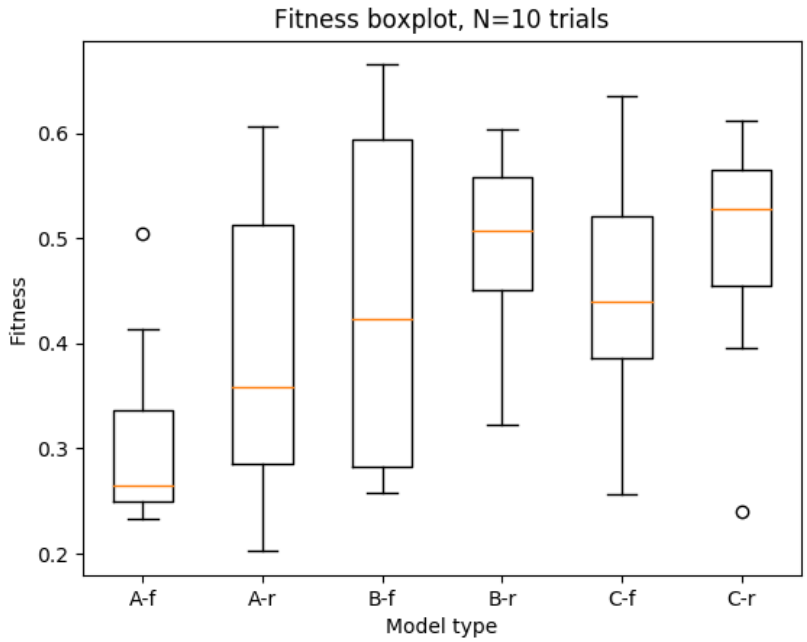


Figure 4.12: Boxplot overview of $N = 10$ best **fitness** achieved per model. All experiment trials had the same population size and the generations count. Value of the fitness function was a linear combination of test error value (weighted 0.85) and Q modularity value (weighted by 0.15). Fitnesses were better for the REPEAT operator, due to its significantly better effect on test error, although the Q values were more beneficial to FOR produced architectures 4.11. The main reason for this boxplot was to investigate, whether there were some benefits for prioritizing the amount of modularity values for the individuals, but the modularity values in case of CIFAR experiments proved to have even negative effects on resulting fitness. Therefore although the FOR experiments produced more modular architecture, their overall fitnesses were worse.

4.3.2.1 Example CIFAR experiment case study

Following 3 graphs show some properties of the model **C–f** evaluation. Each individual during the **C–f** experiment reached some value of test error, modularity measure Q, *test_error* and *comlexity*; all of these records were recorded in the database alongside with the number of epoch when the individual was found for the first time.

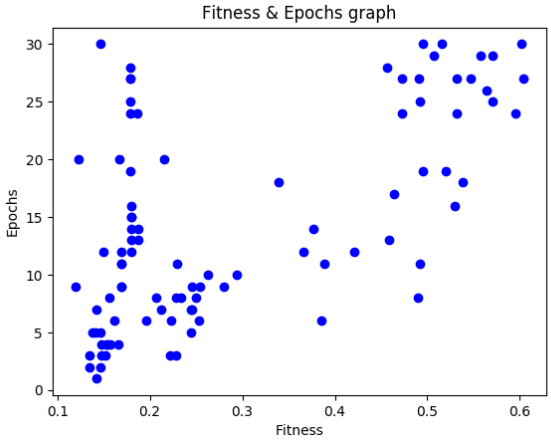


Figure 4.13: Population individuals *fitness* & *epochs* graph in **C–f** experiment.

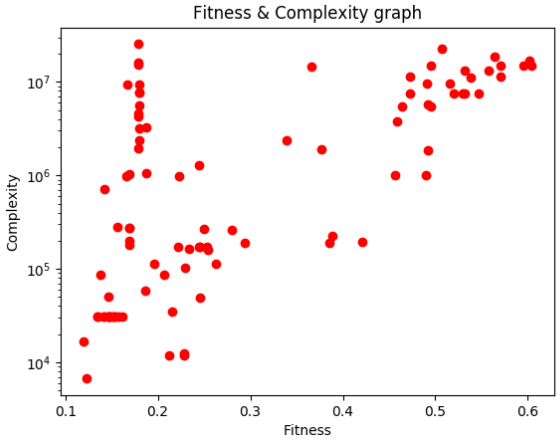


Figure 4.14: *fitness* & *complexity* graph in **C–f** experiment. Rendered dots follows approximately the learning curve of the CNN classifier on the CIFAR–10 dataset.

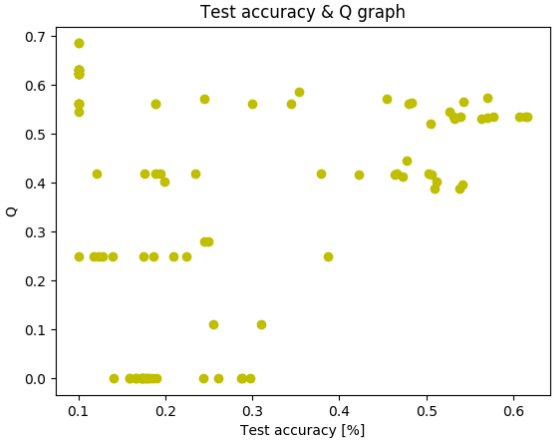


Figure 4.15: *test_accuracy* & Q graph in **C–f** experiment. It turned out, that the models for CIFAR–10 problem had in general significantly lower modularity values then the networks for XOR experiments (see figure 4.6).

4.3.2.2 CIFAR experiment example individual

This is an interesting individual with $Q = 0.4072$ and $test\_accuracy = 0.612$. Notice its rather short chromosome and then see the network it encodes; figure 4.17:

```
[32, 69, 48, 64, 20, 36, 73, 8, 47, 14, 52, 54, 0, 41, 41,
 10, 65, 65, 41, 15, 65]
```

Unlike in the case of FOR operator where the edges which did not introduce any new blocks had no effect on the phenotype, in the case of REPEAT operator every edge truly matters. Notice the DOUBLE operator in the following development tree in the branch, where it is not introduced any new block; it produces new empty edges without any effect. But after considering the effect of REPEAT operator, those empty edges serve as starting points for a new level of recursion which introduces a new structure on top of them.

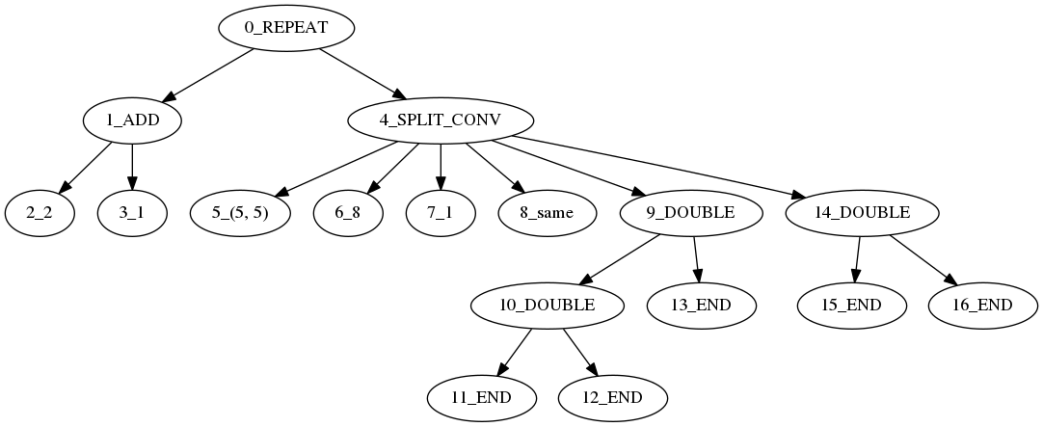


Figure 4.16: Development tree. It is a source code of the example individual of the **C–r** experiment. Although it contains only 1 instruction for introducing the convolutional layer, the resulting phenotype is quite complicated.

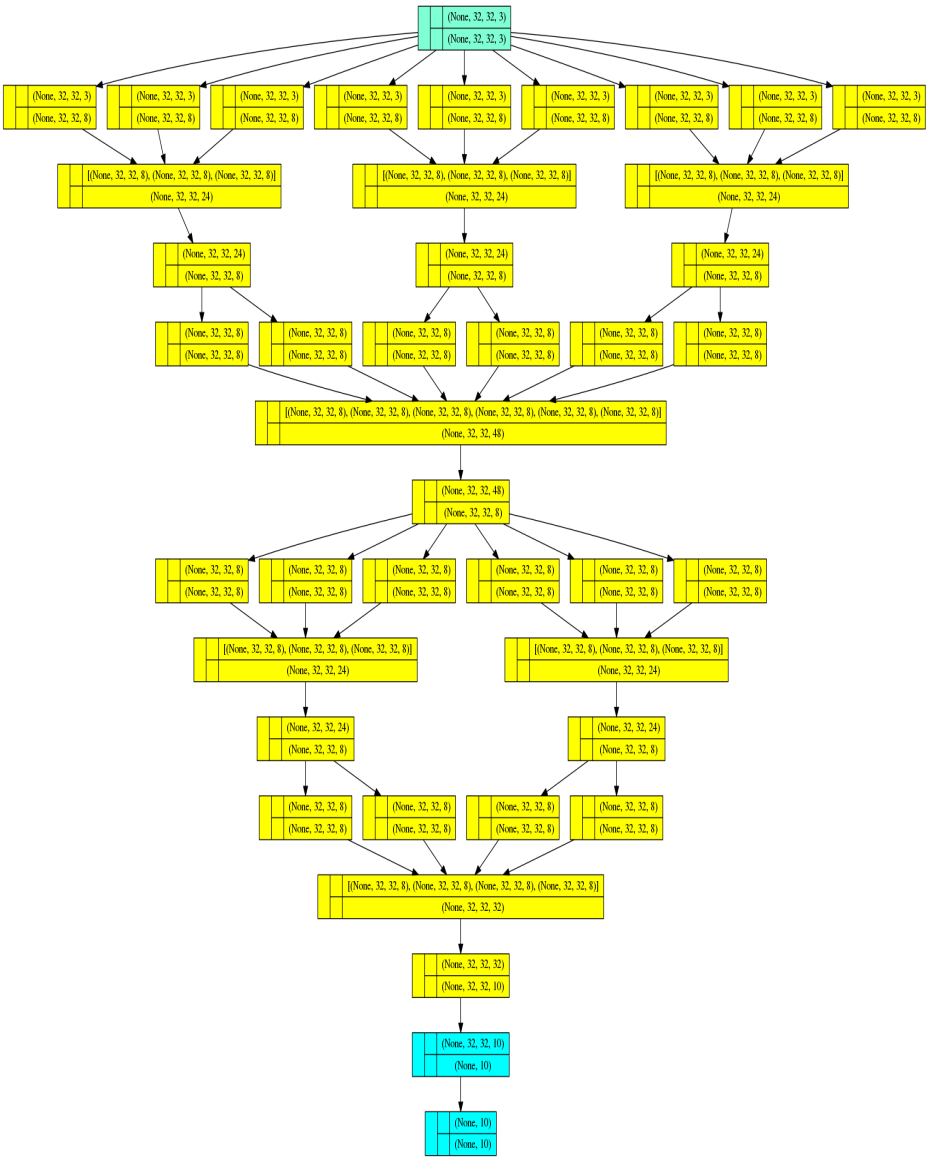


Figure 4.17: CNN produced from the example genome. The REPEAT operator has the ability to produce complex structures from a short program due to its exponential nature. It is somewhat questionable, whether it can lead to those desired modular structures. This network belonged to the best found architectures and reached 61.2% test accuracy on CIFAR–10 dataset. Notice also, that this particular CNN does not use pooling layers; the network consists only from convolutional and concatenate layers (not mentioning output global max pooling layer).

4.4 Discussion

In terms of a modularity measure for the CNN phenotype it cannot be claimed that the more modular architecture is, the better test accuracy will have; it is apparently domain dependent whether more modularity brings more benefits or not. In the case of XOR–N problems there was observed a positive effect of modularity, whereas at the CIFAR–10 experiments it showed out that the best individuals had lower Q values on average. Probably the intrinsic essence of XOR–N problem prioritized the networks, that had a certain structure that had a higher Q value, although the best individuals did not appear to have some specialized functional blocks (4.2.2.2).

The claimed assertion about the general independence of the level of network modularity on the network test accuracy on (general) classification problems was verified experimentally and it is based on the following thought. We took the same problem setup; i.e. XOR/CIFAR with some particular grammar, and on both of this setups we additionally set the different modularity operators – FOR and REPEAT. It was observed that the modularity Q value is in the average higher at the networks produced by FOR operator. Based on the results which were also plotted in boxplot diagrams there was observed that the FOR operator (more modular operator) performed better on XOR problem, but performed worse on CIFAR dataset and therefore we cannot generally rely on the benefits of higher modularity values.

Our final experiment was aimed for investigating the method performance on CIFAR–10 dataset in the similar conditions as the SOTA results; i.e. taking the best individual and train its CNN as well as possible, by using an augmented dataset and a carefully chosen optimizer settings. The test accuracy of our best individual was **79.5 %**.

There were also carried out the experiments whose objective was to detect the best suggested preliminary form of the edge encoding programming language grammar for generating the CNNs (see 4.1 and 4.10). There are several interesting observations:

1. The usage of variables in our experiments did not bring any advantage in increasing the performance. It notably increased the space of searched programs, which made the whole evolution significantly harder, which was showed by worse average performance of the produced individuals.
2. The amount of modularity Q values per individuals did not notably depend on the details of used grammars. Practically the only entities, that influenced the Q values were the modularity operators FOR and REPEAT.
3. Recursive integer values proved to be generally beneficial for the GP optimizer. It was probably because they enabled to use arbitrary integer values – on the expense of greater search space, but the benefits prevailed.
4. The smaller the grammar was, the better performance it gave; i.e. the conditional expressions and variables only decreased the performance.

We did not have much ambitions to reach SOTA performance when dealing with such a novel and unexplored approach, but at the same time we believe we provided a useful starting point for possible further research in this area.

4.4.1 Comparisons & Problem Remarks

Method	Source	CIFAR–10 test accuracy
Manually designed models		
Fractional Max-Pooling	[5]	96.53 %
The All Convolutional ANN	[6]	95.59 %
Mixed Gated, Tree CNN	[8]	93.95 %
Spatially-sparse CNN	[9]	93.72 %
Human performance	[10]	94 %
Automatic methods		
Q-Learning	[28]	93.1 %
Cartesian GP	[11]	94.0 %
Reinforcement Learning	[12]	96.5 %
CoDeepNEAT	[13]	92.3 %
Large-Scale Evolution	[17]	94.6 %
Our method: plain evolution		**64.7** %
Our method: best individual		**79.5** %

Table 4.2: SOTA overview of methods and their performances. Our reported test accuracy value comes from the best found architecture I_{best} of **C-f** setup 4.3.1.3 (I_{best} was an individual found on 27/30 generation from the population of 30 individuals). The value reffered as **Our method: best individual** was achieved by training I_{best} on augmented dataset with unlimited number of ANN training epochs. **Our method: plain evolution** means the experiment run I_{best}, where the number of training epochs per individual was limited on 50 and the dataset was default without augmentation.

Used data augmentation for the training of the best individual I_{best}, per 1 ANN weight update used 96 randomly augmented batches of 512 CIFAR–10 images. Used parameters correspond to the Keras ImageDataGenerator class.

```
batches_per_epoch = 96
batch_size = 512
max_ann_training_epochs = 800
featurewise_center = True
featurewise_std_normalization = True
rotation_range = 10
width_shift_range = 0.2
height_shift_range = 0.2
horizontal_flip = True
```

4.4.2 CNN training related problems

During the final analysis of the reasons why our method achieved such a poor performance, we consider the following remark to be very important. Approximately 60 % of all individuals during the evolution process were not fully trained; i.e. their achieved train set accuracy was typically lower than the test set accuracy (underfit), or there was a steady improvement of both the test and train accuracy during the training which have been although interrupted after 50 epochs. Moreover, the resulting test accuracy is a random variable, therefore the whole training of an ANN should be repeated multiple times, which we obviously could not afford.

We cannot blame only the absence of unlimited number of ANN training epochs, because our reported individual I_{best} was trained after 17/50 epochs (and reached 64.7 % accuracy). When the I_{best} was trained on augmented dataset, it took him 652 epochs to reach 79.5 test accuracy. It means, that some quality individuals can be trained after a lower number of epochs, but as was investigated, that does not apply for the 60 % majority of individuals that were not fully trained.

Another thing to investigate was to compare the computational demands of SOTA CIFAR–10 networks, with the networks produced by our algorithm. For example the number of weights of the I_{best} individual was approximately 7 % of the number of weights of the best [17] individual (that reached 94.6 % test accuracy). Another interesting remark, that compares the amount of computational resources is, that [17] used 250 parallel GPU workers per 280 hours, whereas our experiment that is reffered in the table 4.3 used only 10 CPU workers per 120 hours (because our access to GPU machines on Metacentrum was restricted after our extensive usage in early experimenting stage).

	Params.	**Test Accuracy**	**Computational Resources**
Large-Scale Evolution	5.4 M	94.6 %	250 workers, 270 hrs, CPU + GPU
Our method	0.4 M	79.5 %	10 workers, 120 hrs, CPU

Table 4.3: Comparison of the I_{best} and the best individual from [17] in terms of number of parameters and reached test accuracy on CIFAR–10 dataset.

4.4.3 Assessment of the solution effectiveness

There are naturally several more related works, where the GP approach was investigated. The essential difference is namely in the level of pre-coded knowledge. A common practice in the effort to solve CNN structure optimization problem in the SOTA methods is:

1. Increase number of neural layers in the offspring networks as training progresses.
2. Increase number of convolutional filters with its depth in the network.
3. Select parameter sizes from the predefined sets of available values.
4. Use some means of dataset augmentation for training the classifier.
5. For a test accuracy report, take the best found architecture, use a grid search over the values of gradient optimizer settings and train the best model on it.

None of those hand tailored (but apparently reasonable) tricks were used in our experiments, as our focus was mainly on the design of operators and indirect encoding techniques for achieving modular architectures.

4.5 Suggestions

As the used methods for DNN structure optimization did not achieve the state of the art performance, (in sense of CIFAR-10 dataset test accuracy) here are several concrete points, which author considers worth for further investigation, that should very likely improve the performance:

1. Implement more advanced evolutionary algorithm. Consider a method called genetic programming with explicit fitness sharing (GPEFS) [14].
2. Consider a completely different reproduction strategy; e.g. $(1 + \lambda)$ mentioned in [11].
3. When using the rewriting rules in the phase of random program synthesis, use a random vector as a non uniform probability distribution in order to support the usage of particular rewriting rules. For example the rule for deploying the SPLIT_CONV instruction would by prioritized by sampling it in 30% of all expressions, while the SPLIT_POOL instruction would be sampled by e.g. only 15% times.
4. Use some technique for detecting the equivalence of synthesized programs. In general this task is intractable, but with the limited context of this problem, there should be some suitable solution at the expense of introducing another complication.
5. Create the program source code optimizer that omits the parts of the code with no apparent influence on the resulting phenotype; e.g. FOR loops without any effect, or investigate why it is beneficial to have some redundant genome structures (similar to introns in real genomes).
6. Implement more advanced techniques for mutation. It might be considered for example automatically defined functions (ADF) [23].
7. Find some visual datasets where the benefits of introducing more modular architecture are more apparent then in the case of CIFAR–10 dataset, but still preserve the sufficient complexity of such dataset that would simplify the finding of actual usefulness of tested method.
8. Try to come up with more indirect encoding operators and introduce more features of the designed language.
9. This thesis contained a detailed investigation of CIFAR–10 results. For the purpose of proving a scalability of the algorithm, use the CIFAR–100 dataset. Generated networks for CIFAR–100 problem should share very similar topological features as those for CIFAR–10.
10. For a purpose of another advanced mutation technique or a modularity investigation, consider the genome analysis by searching for the frequent subsequences in the population of genomes. Then use those perspective genome subsequences to construct the new functional blocks for inserting to the new population.
11. Use *weight inheritance* of CNNs layer weights during the population mutation. This method can speed up the fitness evaluation; it is described in [17].
12. Provide more computational resources to be able to train CNNs without the limit of maximal training epochs; see a remark 4.4.2.

4.6 Development & Problems

The thing that slowed the experimentations was that the access to Ostrava IT–4 computational grid was by some circumstances canceled just after the analysis and code development finished, which was somewhat frustrating because IT–4 grid contained large number of GPU clusters where the CIFAR–10 experiments would be processed far more easily then in Metacentrum. Also unfortunately the Metacentrum provided minimum access to suitable GPU cards (because the Tensorflow required at least the technology CUDA–3.0 which was available only on a fraction of Metacentrum GPUs). Therefore it was necessary to carry out CIFAR–10 experiments on CPUs, where the typical running time of 1 experiment was 120 hours. In the result the project consumed more then 1000 CPU days.

Regarding the other more serious problems the author considered as a demotivating work the end phase of experimenting, where it was clear, that without further method improvements, the performance of the solution will stay very far from the results of SOTA methods (as it reached only **79.5 %** test accuracy of the best model, compared with the **94.6 %** test accuracy of the best SOTA method based on EA). Although the assignment was not to use edge encoding for CNNs to compete with SOTA methods, such poor performance might discourage the further investigation of this specific and quite interesting combination of methods for automatic models acquisition, which would make this work a dead end (although the knowledge about dead ends is useful as well, but it is not what the author is used to).

Development of this thesis was based on 3–4 cycles where the author followed this schema:

1. Investigate the literature or web sources about the topics which he was advised to by his supervisor for the further work extension.
2. Suggest and design algorithms for domain specific deployment of studied techniques (as most of the procedures had to be adapted for investigated techniques).
3. Implement and debug algorithms.
4. Decide and implement the new approaches for experimental evaluation of the new extension.
5. Report results alongside with the suggestions of the future extension.
6. Consult results with supervisor and repeat the loop.

Chapter 5

Conclusion

The assignment of this thesis specified the goal to design and implement an algorithm based on edge encoding that could allow the optimization of DNN architectures. I was also instructed to evaluate my algorithm experimentally with the focus mostly on scalability and ability to encode modular and hierarchical architectures. I believe that all of those objectives were sufficiently accomplished on the level that the product of my effort will be useful for possible future research.

For the purposes of this thesis I have:

1. Studied the methods of many research papers.
2. Written more than 2000 lines of code Python 3 program.
3. Created the methodology for evaluating my algorithms.
4. Carried out over a 1000 CPU days of experiments.
5. Designed more than 40 types of grammars and 16 of them experimentally investigated.
6. Come up with 2 modularity operators for indirect edge encoding.
7. Created a programming language for synthesizing DNNs.
8. Created a graph edge encoding based interpreter for my language.
9. Designed 5 types of individual mutations.
10. Made a functional program for synthesizing valid CNN individuals from an input genome.
11. Implemented an evolutionary algorithm for grammatical evolution.
12. Made a SOTA comparison of related methods on the CIFAR–10 dataset.
13. Written this hopefully brief and useful report.

Mentioned requirement to be able to evolve scalable networks was being satisfied with an introduction of integer variable expressions with the combination of usage the REPEAT or FOR operators, that also incorporated the notion of repetitive actions. To some extent it was also achieved the synthesis of modular and hierarchical architectures and especially investigated the influence of modularity on the DNN performance on different tasks. It was shown, that this influence is *domain dependent*; it means that researcher cannot rely on the assumption, that modular architectures will perform well in general. There is also reported a final comparison of our designed algorithm and SOTA methods in terms of reached test accuracy on the CIFAR–10 dataset.

The performance of our algorithm cannot compete with other SOTA automatic architecture optimizers, as our whole algorithm was in the experimental/development stage, where the primary interest was not to limit the search space of CNN configurations by if possible any constraints. It was not meant to directly design an edge encoding based optimization algorithm, that would perform best on domain independent ANN classification tasks. Our solution rather brought several new ideas of how the algorithm could look like; especially its operators e.g. FOR and REPEAT and showed the ways of how to measure the modularity. Functional implementation can be used as a starting point for possible more serious, not purely experimental implementation. It is important to note that the whole approach is far away from its full potential.

Bibliography

[1] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014. [cit. 2018-04-17].

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. [cit. 2018-04-17].

[3] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning, 2016. [cit. 2018-04-17].

[4] Doležal Matěj. *Cellular Encoding for Deep Neural Networks. Master's thesis.* Czech Technical University in Prague, Faculty of Electrical Engineering, 2018, 2018.

[5] Benjamin Graham. Fractional max-pooling, 2015. [cit. 2018-04-21].

[6] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net, 2015. [cit. 2018-04-21].

[7] Dmytro Mishkin and Jiří Matas. All you need is a good init, 2015. [cit. 2018-04-21].

[8] Chen-Yu Lee, Patrick W. Gallagher, and Zhuowen Tu. Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree, 2015. [cit. 2018-04-21].

[9] Benjamin Graham. Spatially-sparse convolutional neural networks, 2014. [cit. 2018-04-21].

[10] Andrej Karpathy. Lessons learned from manually classifying cifar-10, 2011. [cit. 2018-04-21].

[11] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures, 2017. [cit. 2018-04-22].

[12] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2017. [cit. 2018-04-22].

[13] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Arshak Navruzyan, and Nigel Duffy. Evolving deep neural networks, 2017. [cit. 2018-04-22].

[14] Jan Drchal. *Base Algorithms For Hypercube-Base Encoding Of Aritifical Neural Networks.* Ph.d. thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, 2012. [cit. 2018-04-22].

[15] K. O. Stanley, David D'Ambrosio, and Jason Gauci[online]. A hypercube-based indirect encoding for evolving large-scale neural networks, 2009. [cit. 2018-04-17].

[16] Kenneth O.Stanley and Risto Miikkulainen [online]. Evolving neural networks through augmenting topologies, 2002. [cit. 2018-04-17].

[17] Esteban Real, Sherry Moore, Adrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V.Le, and Alexey Kurakin. Large-scale evolution of image classifiers, 2017. [cit. 2018-05-01].

[18] PonyGE Github. Ponyge2: grammatical evolution and variants in python, 2018. [cit. 2018-04-17].

[19] Conor Ryan, JJ Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language, 2006. [cit. 2018-04-24].

[20] Frédéric Gruau. *Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm.* Ph.d. thesis, Ecole Normale Supirieure de Lyon, 1994.

[21] Sean Luke and Lee Spector. Evolving graphs and networks with edge encoding. [cit. 2018-04-17].

[22] Matthew Walker. Introduction to genetic programming, 2001. [cit. 2018-04-17].

[23] John R. Koza. *Genetic Programming : On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1994.

[24] Jeff Clune Marcin Suchorzewski. A novel generative encoding for evolving modular, regular and scalable networks. [cit. 2018-04-17].

[25] M. E. J. Newman. Fast algorithm for detecting community structure in networks. [cit. 2018-04-17].

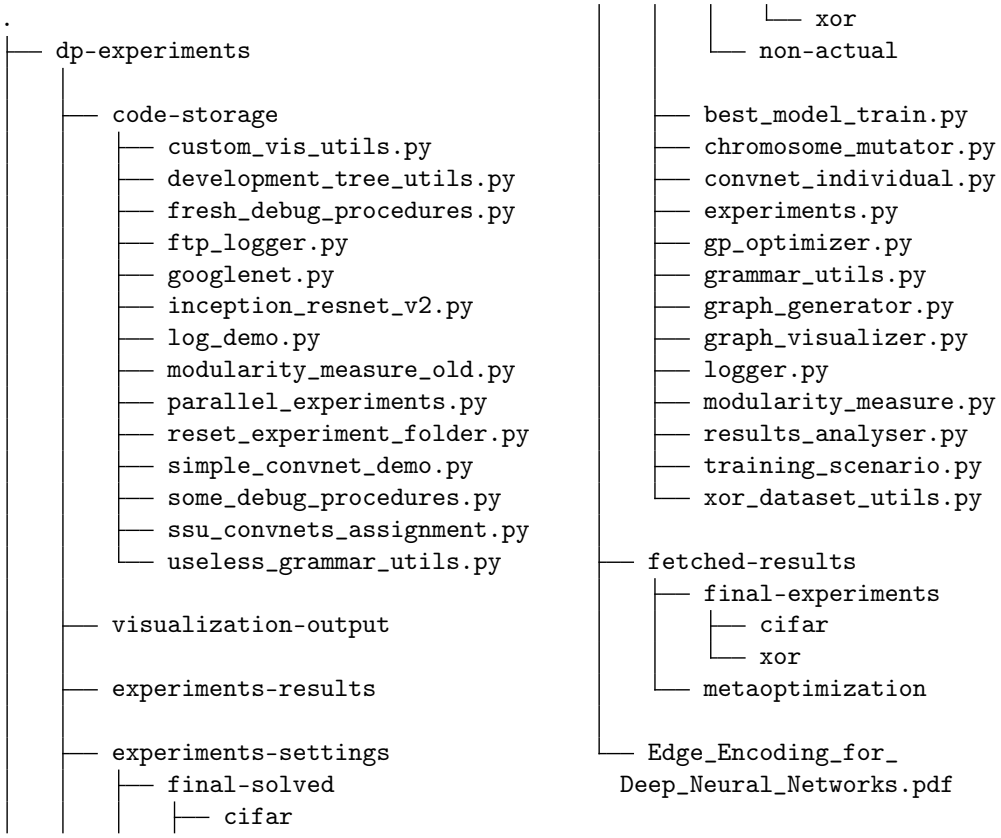
[26] Diederik P. Kingma and Jimmy Lei Ba [online]. Adam: A method for stochastic optimization, 2017. [cit. 2018-04-22].

[27] Alex Krizhevsky [online]. Learning multiple layers of features from tiny images, 8.4.2009. [cit. 2018-04-22].

[28] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning, 2016. [cit. 2018-05-01].

Appendix A

CD contents



Folder **fetched-results** contains 300 MB of experiments results. The entire source code is in the folder **dp-experiments**. Folder **code-storage** contains scripts that were used only for a development purpose. File **experiments.py** contains the default parameter settings as well as the main script for executing the experiments. Default parameter values, are overridden by particular experiment specific settings in the folder **experiments-settings**.