ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Doležal** Jméno: **Matěj** Osobní číslo: **393072**

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:

Celulární kódování pro hluboké neuronové sítě

Název diplomové práce anglicky:

Cellular 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 Cellular Encoding (CE) in particular.
2) Design and implement an algorithm based on CE 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:

Drchal, J. (2013). Base Algorithms for Hypercube-based Encoding of Artificial Neural Networks. Czech Technical University.
Gruau, F. (1994). Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm. Ecole Normale Supérieure de Lyon, France.
Miikkulainen, R., Liang, J., Meyerson et al. (2017). Evolving Deep Neural Networks. Retrieved from http://arxiv.org/abs/1703.00548
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: **21.06.2017** Termín odevzdání diplomové práce: **09.01.2018**

Platnost zadání diplomové práce:
do konce letního semestru 2018/2019

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 Open Informatics

Master's thesis

Cellular Encoding for Deep Neural Networks

Bc. Matěj Doležal

Supervisor: Ing. Jan Drchal, Ph.D.

January 6, 2018

Acknowledgements

I would like to express my gratitude to my supervisor, Ing. Jan Drchal Ph.D. He consistently provided me with guidance and steered me in the right the direction whenever I got lost. His patience and kindness have created a perfect environment for my research.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on January 6, 2018 .....

Czech Technical University in Prague
Faculty of Electrical Engineering

0.0.1 Citation of this thesis

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

Abstrakt

Umělá neuronová síť je výpočetní model, který je inspirován chováním a strukturou lidského mozku. Vetšina přístupu pro stavění neuronových sítí se snaží optimalizovat topologii a zároveň parametry sítě. Tento přístup se na první pohled může zdát ideální, ale pro velké neuronové sítě je nepoužitelný, především kvůli vysokému počtu vnitřních parametrů.

Námi zvolený přístup je zaměřený pouze na optimalizaci topologie za použití Buněčného Kódování, Evolučního Algoritmu a Gradientní Metody. Tato kombinace vytváří ideální nástroj pro hledání topologie neuronových sítí. Na základě vstupních dat a vstupních parametrů najde algoritmus neuronovou síť s nejvyší možnou přesností.

Klíčová slova neuronové sítě, nepřímé kódování, buněčné kódování, evoluční algoritmy, genetické programování, konvoluční neuronové sítě, hluboké neuronové sítě

Abstract

Artificial Neural Network (ANN) is a computational system based on the structure and behaviour of a human brain. Most of the current approaches for building neural networks (NN) are trying to optimise both topology and

the parameters of the neural network (synaptic weights, biases, etc.). Even though it seems like the ideal approach to let the algorithm optimise all the parameters it proved that for large networks it is unusable because the number of parameters is very high and with it the computational complexity.

Our approach is focusing solely on topology optimisation by exploring the possibilities of Cellular Encoding (CE), Evolutionary Algorithms (EA) and gradient method. This combination provides an ideal tool for evolving neural network structures. Based on the input dataset and the input parameters the algorithm searches for the NN topology with the highest accuracy.

Keywords artificial neural network, indirect encoding, cellular encoding, evolutionary algorithm, genetic programming, convolutional neural networks, deep neural networks

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial Intelligence (AI) is a science that focuses on creating intelligent systems and especially intelligent computer programs. It mostly focuses on simulating the human intelligence, but even a program that is capable of impressive performance on a specific task can be considered intelligent. The question that arises from that statement is: What is intelligence and how can we measure it? For measuring human intelligence, we use the IQ tests, which are capable of evaluating the intelligence of an individual but they are unusable for computers. The easiest explanation is that some parts of the IQ test are based on memorization and "in-mind" computations that can be quite difficult for a human brain but form no challenge even for a very low-performance computer. Probably one of the first mechanisms for computer intelligence testing was invented by the famous Alan Turing. For the first time, in 1950's article *Computing Machinery and Intelligence* [10], the Alan Turing introduced his idea of a Turing's test. He argued that any AI system that can successfully pretend to be a human and fool a knowledgeable observer should be considered intelligent. The test itself was based on interactions via teletype, which allowed the computer program not to have a voice imitation and visual appearance. The most noticeable flaw in this theory is the inconsistency, the same program may both fail and succeed the Turing's test based on the observer's IQ(low IQ = program succeed, high IQ = program may not succeed). Other than that the theory seemed logical and certainly very progressive for that period.

Since 1950 the AI has changed quite dramatically, and one might not even realise to what extent AI is part of our everyday life. Our generation had witnessed a real breakthrough in AI when robots with primitive intelligence started to help around the house in the form of vacuum cleaners, lawn mowers and others assistants. Recently we have even moved beyond that, trusting AI with our own lives by using car autopilots, which are offered for civilian use. The speed of the progress in AI is difficult to ignore, and we can only guess

where the AI is going to be in the future since we have already got to a point where the virtual reality and augmented reality is hard to distinguish from the real world. The Elon Musk said on stage at Recode's Code Conference that "There's a billion to one chance we are living in base reality".[11]. The idea that we are a simulation of a more evolved civilisation can be quite difficult to comprehend. If we take a look back at the progress since 1950, it is not impossible to imagine a future where we will be able to create a simulation like that. At the end of the conference, Elon stated this: "Either we are going to create simulations that are indistinguishable from reality, or civilisation will cease to exist."

Probably the most important part of AI is the ability to mimic the behaviour of a human brain. The structure that attempts to emulate the human brain is called the Artificial Neural Network (ANN). Even though we are nowhere close to simulating the human brain complexity, we mark the ANN as one of the most prominent approaches for solving complicated problems. Neural Networks (NN) already exist for some time, but it is only recently that we have reached the performance point where neural networks, and more specifically Convolutional Neural Networks (CNN), started to thrive. Having the performance for testing out the different structures, it now comes down to problems, like creating the better network topology and finding the correct initial parameters. To this day a lot of ANN topology optimisation is done by hand. Even though there exist algorithms for that, they may not always be the ideal solution. Problem with both hand and algorithm attitudes is the scalability. Setting up large networks by hand is very demanding on experience and knowledge in the field and also very tedious. Using algorithms is very hardware demanding, due to the enormous number of parameters that are being evolved along with the topology.

This thesis deals with the scalability issue by exploring the possibilities of Evolutionary Algorithms (EA). Evolutionary algorithms, also part of AI, is a population-based optimisation technique. The whole concept behind EA is based on biological evolution. Using populations of individuals, where each represents a solution to a given problem. Operations like mutation and crossover are used to combine positive attributes of different individuals.

Evolutionary algorithms have been used before to evolve neural networks, but mostly the experiments turned out inefficient due to the number of evolving parameters. We have decided to use a different approach, by using a gradient descent method for learning optimal parameters of the network, we have more resources both computational and time, to find the ideal network's topology. We believe that using a computational power, to find a suitable NN, should bring far superior results over manual approaches.

Most of the state of the art (SOTA) techniques for finding the ideal network's structure are experimental. The website from Rodrigo Benenson [8] offers an overview of SOTA approaches for some of the most common datasets, such as MNIST, CIFAR10, CIFAR100, STL10, SVHN and others. For each, dataset the overview offers most relevant approach and also the research paper explaining the details.

1.1 Problem Statement

The most significant problem this thesis is trying to deal with is a fact that there does not exist any general approach for finding an ideal network's topology. In most cases, this work is done by experts. People who have a significant amount of experience with neural networks, deep understanding of learning algorithms, networks structures and all the mathematical background. We need a new approach that is capable of finding NN on its own. The automatisation of this process offers two significant advantages. The first is saving the precious time of our expert. The second is that unlike the expert, the algorithm is able to find dependencies between modules.

1.2 Goals of the Thesis

The first goal is to create a knowledge foundation for the reader, to explain all the difficulties and all the necessary technical details of this work. After reading the theoretical introduction, the reader should be able to follow the procedure of evolving NN and understand the conclusion of this work.

The second goal is to build a functional algorithm that is capable of evolving neural networks. This section should also provide all the necessary instructions on how to prepare the environment with all its dependencies, how to run the program and how to set up all the parameters.

The third and last goal is to create enough experiments to prove the ability of evolutionary algorithms to find a suitable topology. Explain all the experiments and the outcomes. Last but not least, create a conclusion that evaluates the approach and offers some guidance for future work.

Chapter 2

Theoretical Background

This chapter provides all the vital information to understand the experimental work and its procedures. The topics explained in this chapter are Artificial Neural Networks (ANN), Evolutionary Algorithm (EA) and Genetic Programming (GP). All the theoretical background is explained only deep enough so that the reader can understand this work. Despite this section being a theoretical background, it serves more as an overview of the concepts used in this work. However, all the sections contain references to more in-depth theoretical materials.

2.1 State of the art

Most of the current state of the art approaches for Deep Neural Network (DNN) optimisation are using human-designed modules to operate. Convolutional networks are combining modules of a non-linear layer (e.g.ReLU), max-pooling layers, and fully connected layers for feature detection [12]. The SOTA approach for MNIST is using DropConnect modules [13]. Similarly, ResNet uses modules with skip connections and batch normalization [14] [15], Recurrent Neural Networks regularly contain LSTM modules for memory realization [16] [17]. Based on the dataset it is possible to find the correct research paper and use the appropriately prepared module. However, a more convenient approach would be to have an algorithm that is capable of incorporating such modules automatically into our network. Simply by presenting the input dataset, the algorithm will find the most suitable neural network that is trainable to a reasonable level of accuracy.

The Neuro-evolution approach for automatic ANN topology optimisation has already been considered, and the results proved that this direction is capable of finding very promising results. Optimization methods that optimise both structure and parameters are often marked with the TWEAN abbreviation

that translates to Topology and Weight Evolving Artificial Neural Network. Amongst the most famous algorithms are SANE [18], ESP [19] or NEAT [20].

The biggest problem with all the above-mentioned approaches is the scalability because all the algorithms turned out to be unusable for current deep neural networks. Due to the recent changes in HW performance, the sizes of DNN have changed quite dramatically, from tens to thousands of neurons. The inability of scalability is caused by the direct encoding approach, where network's representation is proportional to its size, which makes it unusable for large networks.

To be able to use bigger DNN we need to get rid of the direct correlation between the actual size and the representation. That can be done by using the indirect encoding [21] approach. The most relevant method is a Hypercube-based encoding [22] used in the HyperNeat (HN) [22] algorithm. HyperNeat does use, as the name suggests, the fundamental principle of the NEAT algorithm, but it is modified to offer a better scalability. HN also incorporates a new feature for weight computations called Compositional Pattern Producing Networks (CPPNs) [23]. However, even the CPPN is not able to evolve deep architectures. Fernando et al. introduced an interesting approach called Differentiable Pattern Producing Network (DPPN) [24] where the CPPN from HyperNEAT is optimised by a gradient method [25].

Our approach is inspired by the idea used in a research paper from Estaban Real et al. [26], where they use the cellular encoding and treat each node as a whole layer instead of one neuron. That approach significantly reduces the complexity of individuals. On top of that, we use a gradient method to optimise the network's parameters, more specifically a gradient-based method called Adam [27]. We believe that the combination of Gruau's, Estaban's, and Fernando's approaches may provide some interesting results and a solid foundation for future experiments.

2.2 Artificial Neural Network

The Artificial Neural Network (ANN) is a computational system inspired by the brain of a living organism. Despite the fact that first research papers appeared around the year 1940 (1943 McCulloch and Pitts), it is still a very fresh field of study. We do not understand how precisely the biological neural network works. Most of the work in ANNs is about mimicking a brain's behaviour and testing out new approaches. Fortunately for us, the hardware is becoming better and faster every year, and we can simulate more extensive and more complicated NN. However, we are still very far from simulating

networks close to human's brain complexity. This thesis is focused solely on deep neural networks and convolutional neural networks.

2.2.1 Neuron Model

A neuron in living organisms is a cell that allows transferring information, through electrical and chemical processes. To understand neuron cell's structure and its functionality is quite difficult, but to explain neuron's function in ANN, we can luckily omit a lot. Neuron model in ANN is a simplified version of a biological neuron cell, and it is a building block for neural networks. In this thesis, we only consider a perceptron type neuron.

By looking at the perceptron model in figure 2.1 we can see that it basically works in three steps/layers.

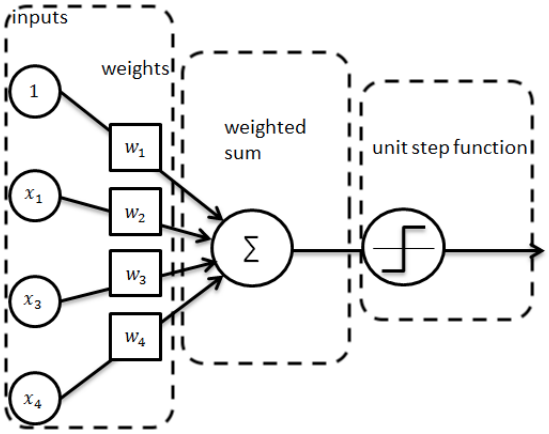

Figure 2.1: Perceptron model

Input, weight layer
Every input/signal that comes into the neuron cell gets multiplied by a weight assigned to the input. These weights are individually adjusted by a learning process of the neural network. From a mathematical point of view, the first step is only a sum of all inputs x_i multiplied by the synaptic weights w_i.

$$U_p = \sum_{i=1}^{n} x_i * w_i \tag{2.1}$$

Weighted sum layer
This layer takes all the weighted input values and sums them up to a single value. This final value is modified by the bias. Bias value helps to shift the behaviour of the neuron, and it is adjusted by the learning process. The second step represents a sum of weighted input sum U_p and bias value b_p.

$$Ub_p = (Up + b_p) \tag{2.2}$$

Activation function
The final stage of every neuron is an activation function that takes as an input a numerical value and outputs a number based on the selected activation function ϕ.

$$Y_p = \phi(Ub_p) \tag{2.3}$$

Mathematical background used for perceptron explanation is inspired by the materials in Simon Haykin's book [28].

2.2.2 Activation Function

To allow NN to learn a non-linear function, we introduce the activation functions. Every activation function takes a single number and performs a mathematical operation and outputs the resulting number. The most often used activation functions are sigmoid and hyperbolic tangent (TanH).

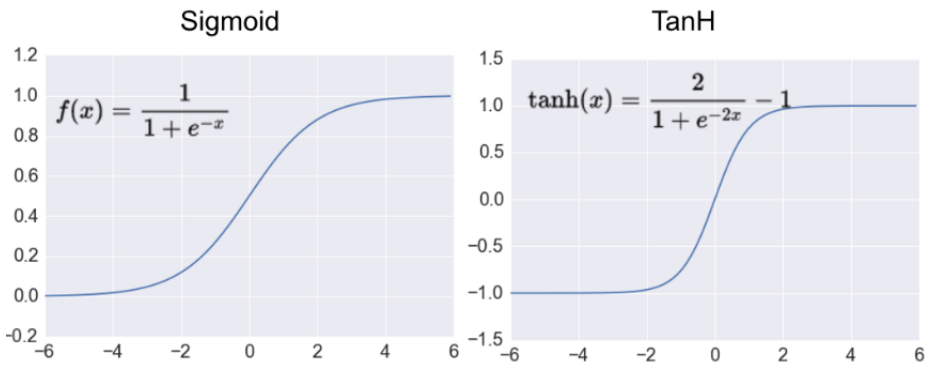

Figure 2.2: Sigmoid and Hyperbolic Tangent activation function (image from another source [1])

Recently a new activation function was introduced and it quickly became very popular. ReLU 'Rectified Linear Unit' can be implemented very efficiently comparing to a tanh/sigmoid function, and it was proved [29], that ReLU greatly accelerates the convergence of stochastic gradient descent.

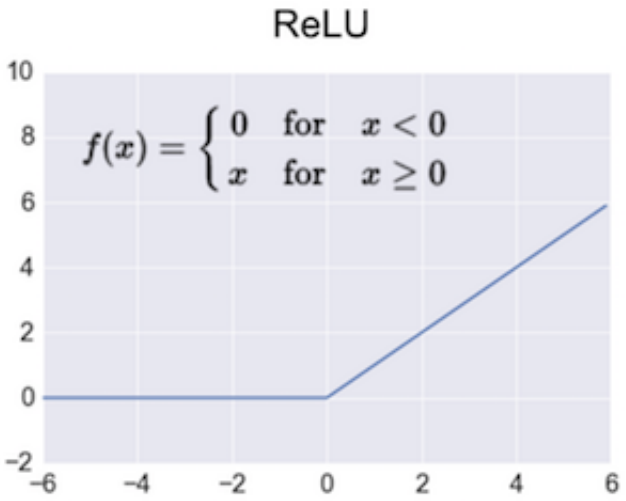

Figure 2.3: ReLU activation function (image from another source [1])

2.2.3 Data sets for ANN

For training and testing the neural networks, we need data. The input data type is based on the problem. It can be represented as vector, matrix, image, tree or any other data structure. The following conditions help to achieve better performance.

2.2.3.1 Data properties

Quantity
The size of the input dataset is important. For instance, MNIST dataset has 60.000 images. All pictures should be normalised, meaning that every picture has the same size and image quality. In the ideal case, the data should be as diverse as possible. For instance, for MNIST it is ideal to have a number of various types of people. People with high income, low income, high intelligence, low intelligence, with dyslexia or any other disability, both right and left hand. We should also consider different types of personalities since writing reflects all of that. The concept is

simple, gather input data with the most extensive variety, and the model will perform as expected for any test data.

Correct Labeling
Having correct labelling is crucial for a functional network. It might seem like an obvious statement, but labelling is one of the most tedious tasks when it comes to neural networks, and there is a big space for human error.

Class balance
To prevent the model from mistakes, it is imperative that every class represented in the data should have the same amount of samples. For MNIST it means that the number of images across the whole set, that is from zero to nine, is precisely the same (number of images with zero equals to the number of images with any other number in the dataset). Later in this thesis, we show how to correctly create a subset of MNIST.

When it comes to NN every dataset is most often split into three different sets. Training, validation and test, where the split ratio is usually close to [80%, 10%, 10%]

Training set
Training set is used for learning and finding the optimal parameters of the classifier.

Validation set
The validation set is used to tune up the parameters. By having a validation set, we reduce the possibility of having a bad result on the test data.
Using a validation set in optional.

Test set
Test set is used to determine the final performance on a real-life data.

2.2.3.2 Overfitting and Underfitting

Over and underfitting are terms used in machine learning that explain the behaviour of a classifier.

OverFitting is a case when the network's model performs flawlessly on the training set and poorly on a testing set. It is often described as a state, where the model starts to memorise the data, rather than understanding it. Over-

fitting can be caused by having a data set that violates any of the condition defined above 2.2.3.1 or by simply pushing too hard and forcing the model to reduce the final error. Such overfitted model may entirely lose the ability to generalise and perform well on unseen data.

Underfitting on the other hand, is the exact opposite, a state where your model performs very poorly on your training set because it is not able to fit the data well enough. That can be caused either by having an insufficient amount of training data or by fitting the data with a wrong model (liner non-separability).

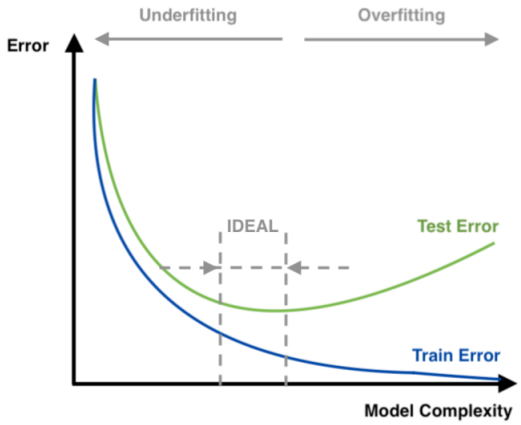

Figure 2.4: Overfitting and Underfitting

2.2.3.3 K-fold cross validation

K-fold cross-validation (CV) is a process that helps with finding the perfect model for given dataset while ensuring to avoid overfitting. The first problem when it comes to a dataset is how to split it into train and test sets. Ideally, we want the biggest test set possible to ensure the model can deal with any data, but at the same time, we need most of the data for training purpose. The fewer samples we have, the more relevant this problem becomes. The greatest advantage of K-fold CV is that we can use the whole set for both training and testing.

1. We divide our dataset to **K** folds with roughly the same size (some datasets may not be divisible by **K**).

2. Create a **K** iteration cycle. For every cycle, one fold represents a test set, and the rest is for training. See the graphical explanation in figure 2.5.

Figure 2.5: K-fold cross-validation

3. Calculate the cross-validation error

$$CV_{error} = \frac{1}{K}\sum_{k=1}^{K} E_k \tag{2.4}$$

E_k is a test error on kth fold

2.2.4 Learning process

Learning in ANN is a process that was already explained to some extent in previous sections. It is a process where all the internal parameters are tuned (weights, bias, thresholds, etc.). This section provides only a very superficial introduction to learning mechanisms used in ANN.

Supervised learning.
Supervised learning represents a learning where we have a set of pairs *(training set)*. Each pair is defined by an input x and output y vector. We use our algorithm to find a mapping function $y = f(x)$. The word supervised is derived from the fact that we have a training set/teacher, that helps us find the ideal mapping.
(classification, regression)

Unsupervised learning.
Unsupervised learning is a method used, for situations when we only have a training data x. The outcome of unsupervised learning is a to find a model that understands the data's distribution and nature. *(clustering, association)*

Reinforcement learning.
Reinforcement learning is probably the most interesting learning mechanism. It is a scenario where we do not have any data neither x nor y. All the data is generated by interactions with the system. The best explanation is imagining a robot controlled by AI. When we turn on the robot, we have no data, but when the robot starts interacting with the surroundings, the model receives all the data from all the sensors.

This thesis does not provide a mathematical background for understanding the learning process of neural networks. To understand the concept of a learning process, consider reading a book on neural networks from Simon Haykin [28].

2.2.5 ANN architecture

Using neurons as building blocks, we can construct sophisticated networks, with very specific properties. Different architectures vary from each other by size and number of connections between neurons. Finding the perfect network's topology for a given problem can create a task on its own. Following existing guidelines can help to select a network's shape, but the size and density are usually tuned by trial and error method.

2.2.6 Deep Neural Network

A Deep Neural Network(DNN) is a just a name convention used for neural networks with a certain depth. We can divide ANNs into two subgroups by its depth, shallow and deep neural networks. Regular NN has three different layer types, input, hidden and output. For NN to be considered shallow, it must have no hidden layer or at most one. For any other number of hidden layers, the NN is considered as deep.

Using more hidden layers created an innovation known as 'feature hierarchy'. Each hidden layer is trained for a distinct feature based on the previous layer's output. Deeper we go the more complex features NN recognises. That allows NN to handle high dimensional data sets.

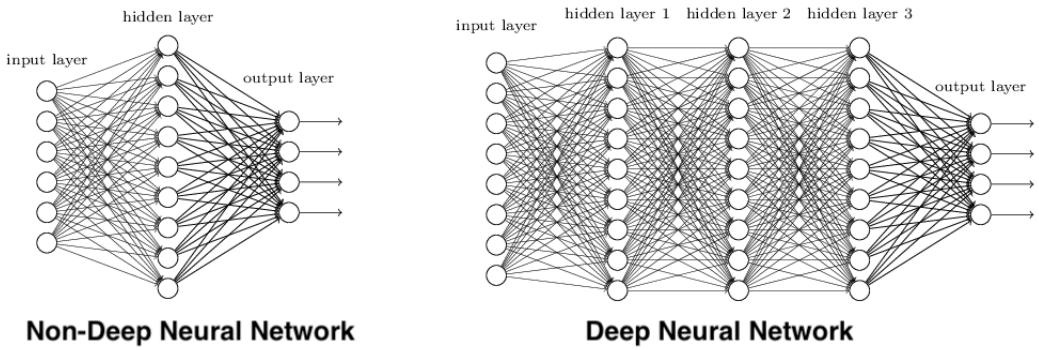

Figure 2.6: Deep Neural Network (image from another source [2])

2.2.7 Fully Connected Feed Forward Neural Network

A fully connected feed-forward neural network is one of the simplest neural network structures. It is a combination of two elementary rules. Fully connected layers represent architecture, where all neurons are connected to each other, and feedforward is a promise that the connections between neurons are only in on direction, no cycles. See the structure in figure 2.7.

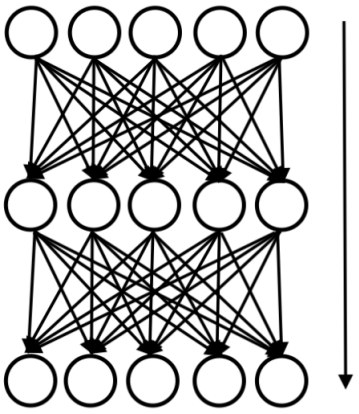

Figure 2.7: Fully Connected Feed Forward Neural Network

2.2.8 Recurrent Neural Network (RNN)

Recurrent neural network introduces quite a unique feature of neural networks, and that is a memory. Unlike feedforward networks, RNN allows connections that can create a directed cycle in the network. The possibility of having a cycle creates a structure that is capable of holding information about already processed inputs (memory). For more in-depth explanation consider reading the book on RNN [16].

2.2.9 Convolutional Networks (CNN)

Convolutional Neural Network (CNN) is a deep learning architecture that is designed to recognise and classify images. CNNs are extensively studied in last few years, due to the hardware innovations. Not only that we have better calculation power, but we also have vast labelled datasets, that are being created on an everyday basis due to the extensive use of internet in combination with mobile cameras.

CNN has many different components that you can see depicted in the CNN structure below, created by authors of a research paper [3] that nicely sums up all the recent advances in CNN (The original picture was slightly modified for better printability).

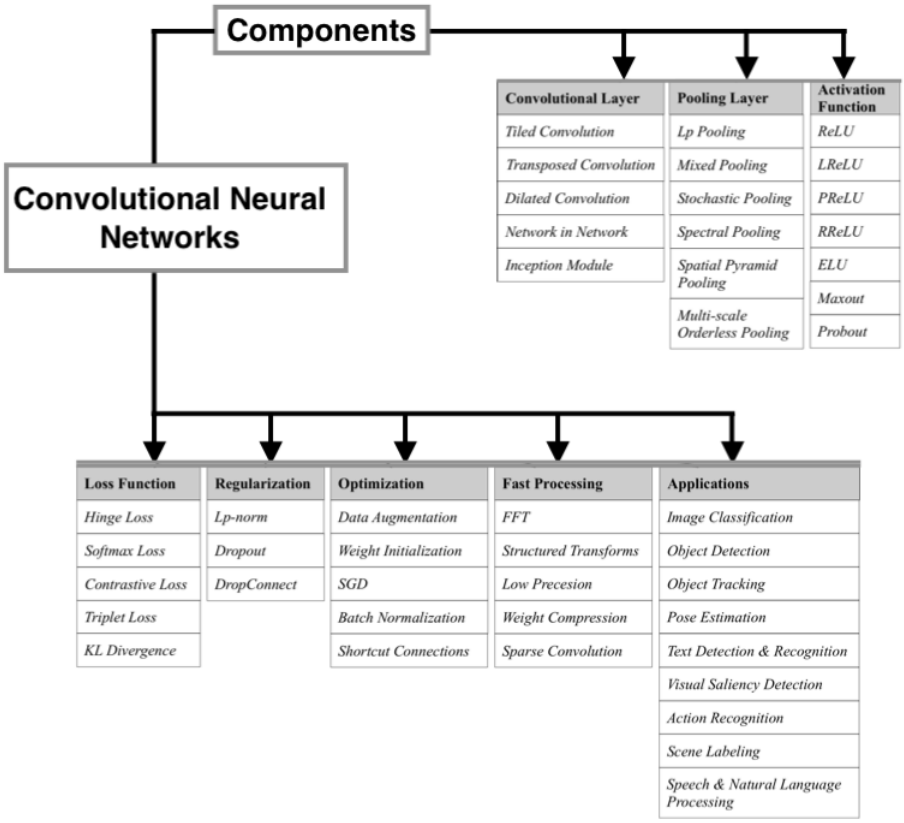

Figure 2.8: Convolutional Neural Network tree (image from another source [3])

Let's introduce the basic blocks in CNN networks that are common across all architectures.

2.2.9.1 Convolution

We have already stated that CNNs are used for recognising images. Any image can be represented as a series of matrices with numbers from 0 to 255. A regular image has three matrices one for each colour channel (RGB). Dimension of the image is the dimension of a matrix, where each position in the matrix is specific pixel in the image.

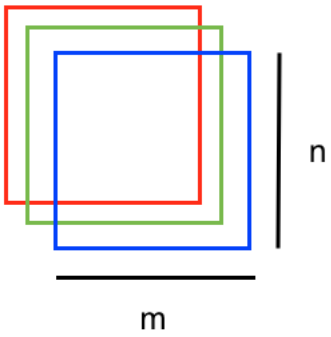

Figure 2.9: RGB - channel matrices

For an easier explanation of convolutional step we only consider the black and white image, that is an image with a single colour channel (black) with normalised values from 0 to 1 (usually black and white pictures have values from 0 to 255 to display grey and its shades). A convolutional step has the ability to extract specific features from the image. We will not go into detail on how exactly this works, but we will cover the basics to understand the overall idea.

One part of the convolutional step is our image. In our case a matrix with values $0, 1$ with dimensions $m * n$. The second part is usually a smaller matrix with values in the same interval. This second matrix is often called 'kernel', 'filter' or 'feature detector'.

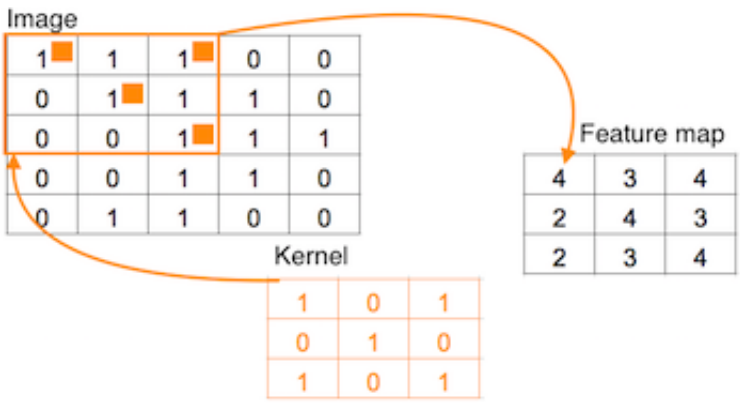

Figure 2.10: Convolutional step

By sliding the kernel over the original image and calculating the outputs we receive the final matrix. Every position in the final matrix is calculated as a sum of multiplications of elements at the same position. Calculation of the first value is depicted in figure 2.10, for other values, simply slide the kernel right or down. The example in in figure 2.10 is using a stride equal to one. The stride value determines by how much we slide the filter at each step.

The final matrix created by this procedure is called the 'convolved feature', 'activation map' or 'feature map'. By using different filters, we can filter out different features like edges curves etc. By specifying feature map parameters such as the number of filters, network's architecture, filter size etc. the CNN can create the necessary filters on its own.

2.2.9.2 Padding

Some of the mechanisms (convolutional step, pooling) used in CNN are changing the original image dimensions. The change in size may cause problems when it comes to concatenating layers because the inputs will probably have different dimensions. To ensure the dimensions stay the same we must use a zero padding. To understand how to apply the zero padding see figure 2.11.

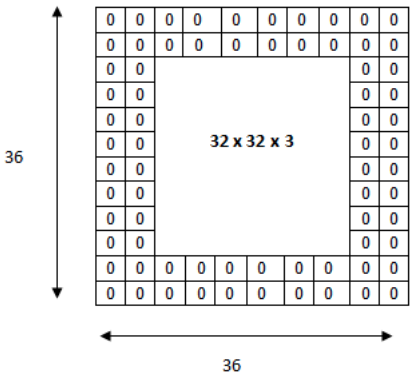

Figure 2.11: Zero padding (image from another source [4])

The output dimension can be calculated by the equation 2.5. From the equation 2.5 we can easily calculate what padding to add to keep the same dimensionality 2.6.

$$O = \frac{(W - K + 2P)}{S} + 1 \tag{2.5}$$

O output dimension W input dimension K a filter size P is the added padding S is the Stride

$$\begin{aligned} O &= \frac{(W-K+2P)}{S}+1 \\ &(S=1, O=W) \\ \\ O &= (W-K+2P)+1 \\ -2P &= -K+1 \\ P &= \frac{(K-1)}{2} \end{aligned} \tag{2.6}$$

2.2.9.3 Pooling

Pooling layer is a feature introduced for the purpose of reducing the dimensionality and detecting the most important segments in the picture. Spatial pooling reduces the image size while keeping the most important information alive. There are many different types of generic pooling such as MAX, AVERAGE, SUM and others.

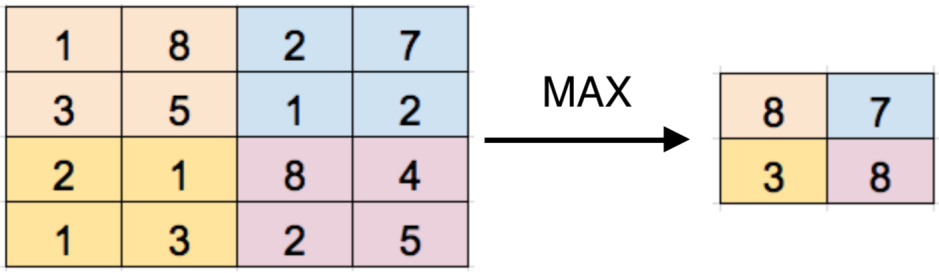

Figure 2.12: Pooling process

Pooling layer is an essential feature of ANN. It helps to reduce the number of connections between convolutional layers. Generic pooling methods are straightforward, but there are also more sophisticated mechanisms for pooling. Few of the most recent pooling techniques are Lp, Mixed, Stochastic, Spectral, Spatial Pyramid or Multi-scale Orderless pooling [3].

2.2.9.4 Dropout

Dropout is one of the most popular regularisation techniques for DNN [5]. It is designed to prevent the neural network from overfitting. During the learning process, the neural network temporarily discards a neuron with a probability `p`. The reason behind dropping the neurons is to train the neurons independently and thus forcing every neuron to perform well on its own.

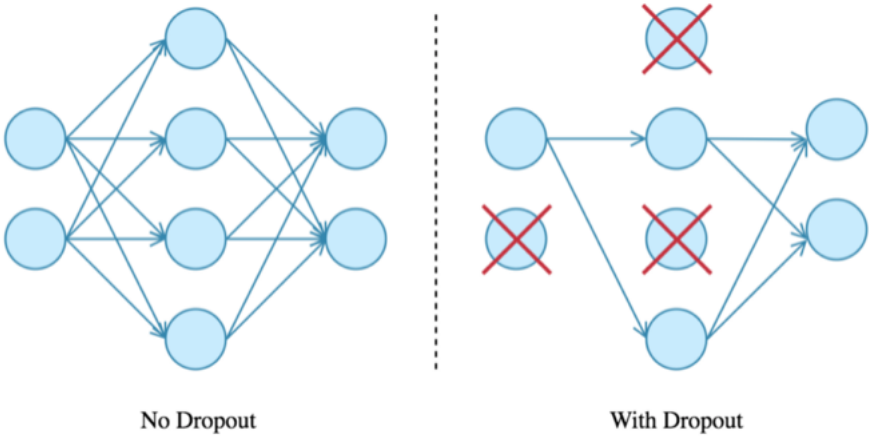

Figure 2.13: Dropout process (image from another source [5])

2.2.9.5 Fully Connected Layer

In combination with pooling layers, it is useful to use fully connected layers. Such layer takes all the neurons from the previous layer and connects them to all neurons in the current layer. The purpose of using fully connected layer is to classify the image based on the features obtained from pooling layers. It works as a multilayer perceptron that uses softmax or SVM classification method. On top of classification, a fully connected layer is also capable of finding non-linear combinations of features. Using a combination of features from pooling layers can create a better classifier than just a classifier from individual features.

2.2.9.6 Gradient Descent

Training NNs, in general, is a very complicated process. The number of parameters necessary to train is enormous and to include these parameters in EA

is impossible. In our thesis, we use EA only for topology-related parameters, and we leave the training entirely to a gradient descent method. Gradient descent is an optimization function that is capable of changing parameters to decrease error. Various intuitive explanations of what gradient descent exist.

Imagine a situation where a hiker is on the top of a mountain. He has a car in a parking lot at the very bottom of the hill. Unfortunately, he got to the summit later than expected and the night has already fallen. He has no visibility and no light. Intuitively he can feel the ground beneath his feet and determine if it tends to descent. The very basic strategy is always to go down if possible, and most probably he will reach the bottom. The graphical representation can look like the graph below.

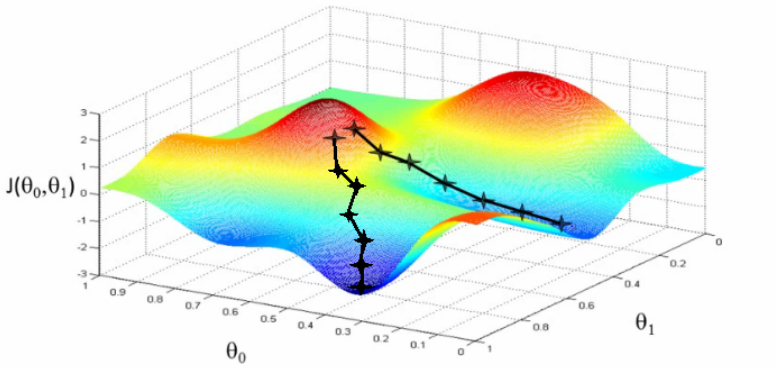

Figure 2.14: Gradient descent (image from another source [6])

The goal is to set parameters on y and x axis to get to blue areas which symbolise the optimum. There are different techniques on how to get to the bottom of the hill such as "Full Batch Gradient Descent Algorithm" or "Stochastic Gradient Descent Algorithm". Based on the research from Sebastian Ruder it appears that one of the most favourite gradient descent types is mini-batch gradient descent. There are many strategies to get to the optimum, to read more about individual optimizers read Sebastian Ruder's research paper [25] [30] [31].

2.3 Evolutionary Algorithms

This section is dedicated to Evolutionary Algorithms (EA). It covers basics of EA and all the knowledge necessary to understand this thesis and its experiments.

2.3.1 Overview

Evolutionary algorithms are an efficient and elegant way of finding a solution to a given problem. The biggest advantage of EA is the capability of finding solutions to problems with no assigned solving technique. It is an optimisation technique based on an evolution process entirely inspired by Nature. The algorithm itself finds a solution by evolving from poor solutions to very good ones.

The basic concept is straightforward. First, create an initial population, of an arbitrary size, filled with individuals, where each represents a solution to the problem. Every individual gets evaluated by a fitness function, which describes the quality of the solution. The search for the best solution is done iteratively, generation by generation. Slight combination and modification of individuals in each generation result in finding better solutions. Every generation is created by predefined rules. There exist two basic concepts on how to create a new generation. Every replacement strategy precisely defines how big portion of the current generation will be replaced and exactly which individuals are not fit enough for the new generation. Take a look at the following figure 2.15 to understand the evolution process.

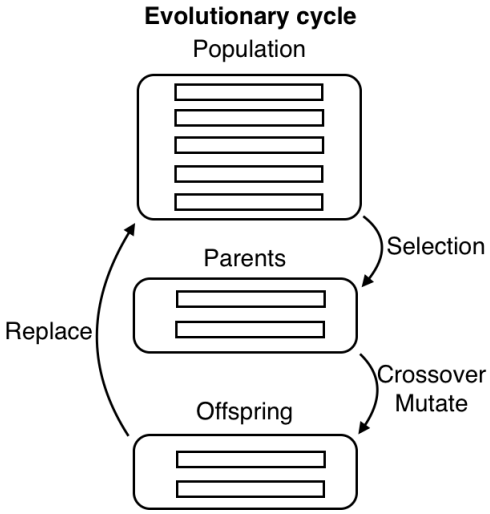

Figure 2.15: Evolutionary cycle

Generational strategy - the whole old population is completely rebuild in each generation (analogy of short-life species)

– **Generational Replacement Strategy** –
```
1 initialize(oldPopulation)
2 evaluate(oldPopulation)
3 while(termination condition)
4     newPopulation ← bestOf(oldPopulation)
5     while(newPopulation not full)
6         parents ← select(oldPopulation)
7         offspring ← crossover(parents)
8         mutate(offspring)
9         evaluate(offspring)
10         newPopulation ← offspring
11         swap(oldPopulation, newPopulation)
12 return bestOf(oldPopulation)
```

Steady-state - just certain individuals are replaced in a generation (analogy of long-life species)

– **Steady-State Replacement Strategy** –
```
1 initialize(population)
2 evaluate(population)
3 while(termination condition)
4      parents ← select(population)
5     offspring ← crossover(parents)
6     mutate(offspring)
7      evaluate(offspring)
8     population ← offspring
9 return bestOf(population)
```

The Evolutionary algorithm is a general bundle that contains many different approaches for finding solutions. Amongst the most popular are Genetic Algorithms (GAs), Evolutionary Programming (EP), Evolutionary Strategies (ES) and Genetic Programming (GP). Different problems require different approaches. The main difference between all these methods is a structure by which is the individual encoded or the process of creation.

Genetic Algorithms [GA]
Uses binary string for representation and genetic operators (mutation, crossover) for evolving.

Genetic Programming [GP]
Automated method for finding the best form of a program tree [symbolic regression].

Evolutionary Programming [EP]
It is similar to genetic programming, but the structure of the program to be optimised is fixed, while its numerical parameters are allowed to evolve.

Evolutionary Strategy [ES]
Searches through the space of real vectors.

2.3.2 Genetic Representation

Genetic representation as the name suggests is describing how the individual can be represented. Using the correct representation is very important. The structure has to be sophisticated enough to encode all the necessary properties and at the same time simple as it can be for the implementation. Representation of an individual is inspired by a living organism, and it has two forms a genome or a chromosome.

A genome is a genetical material of an organism that contains all the necessary information about the organism, and it is usually represented by a string of data. Such a string contains chunks of information that represents individual genes (organism's features).

Every individual is represented by a pair of a genotype and phenotype. Genotype is a specific genome setup. The phenotype is a set of actual features (weight, height, eye colour etc.). In most cases, both genotype and phenotype are used. Genotype is most often used for an evolution and phenotype for the evaluation process. Transformation genotype-to-phenotype is described by a mapping function.

2.3.3 Fitness Function

Overview section already mentioned a fitness function (FF), we will explain its importance. After sorting out the individual's representation, it is important to find a way of assigning a score to every individual to compare their quality. Fitness function does exactly that, it takes an individual in a genotype form, calls the mapping function to obtain a phenotype and then assigns a score in a meaningful way. In math terms, a fitness function is a function that can map any individual to a real number.

$$f : G \to R \tag{2.7}$$

G - is a space of all possible genotypes
R - real number value

A form of a fitness function should be easy to create because it always represents the quality of a solution. For robots, it could be a Manhattan distance to the finish location plus penalty collected on the way, for a math problem it could be the accuracy of a result. The following list of tips should provide guidance on how to create a fitness function.

Accuracy
The accuracy of an evaluation directly affects both precision and quality of the result.

Simplicity
Trying to keep the fitness function as lightweight as possible is a direction to take. Think about how big the population is going to be, how many generations and based on that derive the function's complexity. To avoid slow performing FF, it is important to focus on selecting the ideal representation and implementing efficient mapping function (genotype to phenotype).

Direction
It is a standard to define FF to return higher score for better individuals. For instance, using an error rate on a data set would not be a correct implementation, because it would get smaller over generations.

Invalid individuals
Genetic operators such as crossover and sometimes even mutation can create an individual that no longer represents a valid solution to the problem. For such individuals, a generic fitness function will not work correctly unless it is prepared for situations like that. The algorithm should be capable of detecting invalid individuals. Any of the following strategies can deal with this issue.

Discard
Check satisfiability after every operation that modifies the individual and if the new offspring is not valid, then discard it. Be aware that this strategy will change the size of the population which may be undesirable behaviour. Either generate a new individual, use parents (individuals that created invalid offspring) or use **Repair** strategy.

Repair
Create a method that is capable of analysing the genotype and

detecting the faulty section. Create a method that can fix the genotype and ideally keep the diversity introduced by the genetic operator.

Penalize
Prepare fitness function for invalid individuals and penalise them in a meaningful way. This penalisation will help the Evolution algorithm to detect such individuals and discard them over generations. However, keep in mind that using this strategy can easily discard an optimal solution, by an unfortunate crossover. This problem can be partially solved by including elitism in EA.

Avoid
This strategy is probably the best one, but it can be challenging to implement. Ensure that all genetic operators can create only valid offsprings.

Stability
It probably does not come as a surprise, but FF should always return the same score for two individuals. Using any randomness in the evaluation process can lead to mistakes. For cases with randomness, it is wise to use an average of multiple evaluations or at least try to minimise the error that the random generator can bring in the result.

Example:
In some cases, using random values is inevitable. A good example would be a neural network training. Some algorithms use the random generator for initialisation. Selecting a wrong set of parameters at the start can profoundly affect the network's accuracy. This error can be minimised by multiple runs and averaging all results. (for some networks, multiple runs can be impossible due to the time complexity)

2.3.4 Genetic Operators

Genetic operators are used for modifying individuals by mimicking a process of natural reproduction. Each operator has different properties and different goal in the evolution. For standard GAs, there are two operators, mutation and crossover.

2.3.4.1 Crossover

Crossover combines two individuals (parents), of the same size, and outputs two new individuals (offsprings). The process of crossover in GA is inspired

by the process of reproduction in natural evolution. Children are created by a random combination of parent's chromosomes.

Typical example is 1-point 2.16 or 2-point 2.17 crossover, having more points of crossing rarely makes any sense, but there exists multi-point crossover operator. To make a successful crossover, the algorithm has to randomly select a crossover point and after that create new offsprings by combining contents of parents. Rules are depicted in visual examples figure 2.16 and figure 2.17.

Correct implementation of the crossover operator is very important. Despite the fact, that it was designed to help with the evolution process, there are few traps, that can cause an error. In this particular case, the error is represented by invalid offsprings and possible loss of the optimal solution (parents are discarded). Read fitness function tips 2.3.3 on how to work with invalid individuals. Genetic Algorithms in most cases benefit from using the crossover operator, but there are some cases which can suffer by using it. In that case, it is recommended to either think of a different approach or does not use the crossover at all.

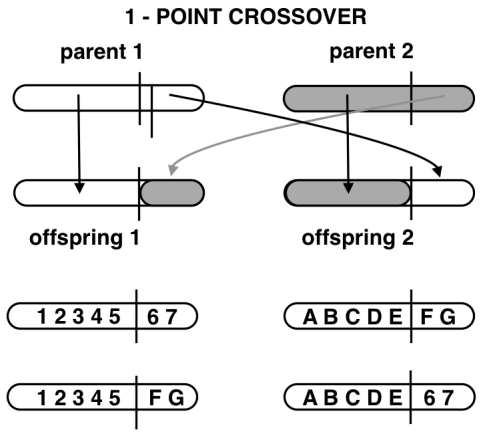

Figure 2.16: 1 - point crossover (non-binary values are used only for better explanation)

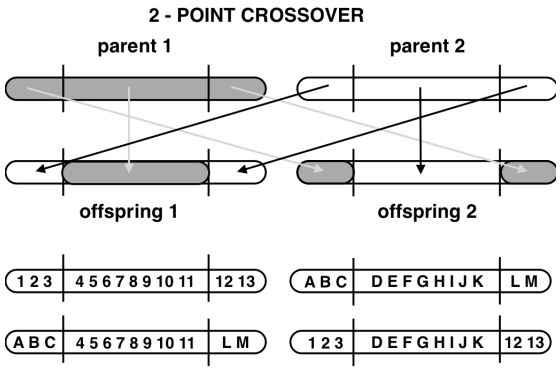

Figure 2.17: 2 - point crossover (non-binary values are used only for better explanation)

2.3.4.2 Mutation

Mutation is a simple and extremely helpful component, that should appear in every evolutionary algorithm. Unlike crossover, the mutation operator is capable of introducing a new genetic information to the individuals. The basic type of mutation is a single bit-flip mutation 2.18.

Figure 2.18: Single bit flip mutation

Implementations can differ based on the specific needs. In general, we mutate one bit per individual with a certain probability, but multiple bit-flip mutations may prove also beneficial. The variety is endless, but it it is important to keep the original idea alive. The mutation should always be able to intro-

duce new information to the individual. The following list explains the most important advantages of a mutation operator.

Diversity
Diversity is a serious topic when it comes to population. Problems with diversity may occur in very early stages. With small diversity, the chances of finding the optimal solution are slim to none. Having a diverse population is an essential prerequisite for finding the best solution. An ideal population has individuals spread out through the whole search space, which is not a trivial condition.

When creating the initial population, the best start is having the biggest diversity possible. By using the random generator, selection of individuals can result in having only a particular area of the search space or even having duplicates 2.19. A better choice is to create the initial population with heuristics or select the population wisely. An example of the wrong initialisation would be to use a first x permutations, that way all the individuals would be very similar to each other.

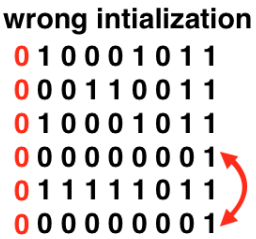

Figure 2.19: Wrong population initialization, duplicates and missing information in the population.

Finding dependencies
For complicated issues where the solution is difficult to imagine one may not realise, the existence of dependencies between bits. Mutation can help to discover certain correlations between bits.

Dead end
Mutation is beneficial when evolution gets stuck in local minima. Having a group of individuals all lacking specific information, no crossover can help to fill that gap, and the evolution gets stuck.

Final Stages

When evolution comes close to an optimal solution, the crossover operator begins to perform poorly, that it is given by the nature of the operator. Crossover usually introduces quite dramatic changes to the individual, whereas all the individual needs are tiny changes.

2.3.5 Selection

Selection is an important process of evolution, where individuals from the current population are inspected and evaluated. Only the fittest candidates have a chance to appear in the next generation. The quality of an individual is solely measured by its fitness score. Learning about the common selection strategies is a good start to understand the process and all its challenges.

2.3.5.1 Roulette Wheel

Roulette wheel selection is a technique for random selection of individuals. The process is best to be explained with a visual aid figure 2.20. The probability of selecting an individual is equivalent to its fitness value. Individual's fitness value is depicted by the size of assigned roulette wheel section.

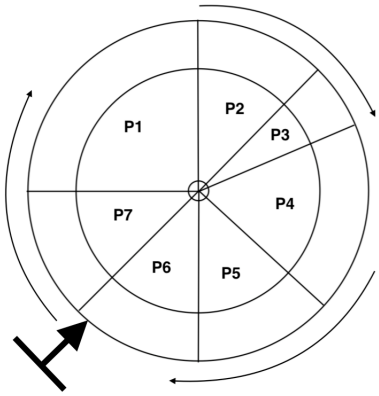

Figure 2.20: Roulette Wheel selection

$$P_i = \frac{f_i}{\sum_{i=0}^{P} f_i} \tag{2.8}$$

P - probability of selection, f - fitness value

Expected vs. Observed frequency
The probability of selecting an individual is proportional to individual's fitness value divided by a sum that represents population's fitness. The problem is that selecting an individual takes one spin at the roulette wheel and selection conditions are same for all the spins.

2.3.5.2 Stochastic Universal Sampling

Stochastic universal sampling is similar to a roulette wheel technique, but it tackles the problem with expected vs observed frequency. By spreading the pointers on the wheel with the uniform distance and using only one spin, that ensures to obtain expected frequencies in line with observed frequencies.

$$S = \frac{360}{n} \tag{2.9}$$

S - step in degrees, n - number of selections

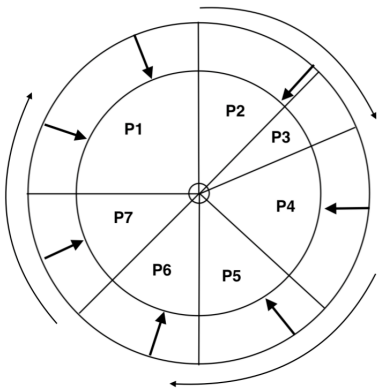

Figure 2.21: Stochastic Universal Sampling

2.3.5.3 Tournament Selection

Randomly selecting **n** individuals from the population and place them in a tournament. Tournament order is decided by the score from a fitness function.

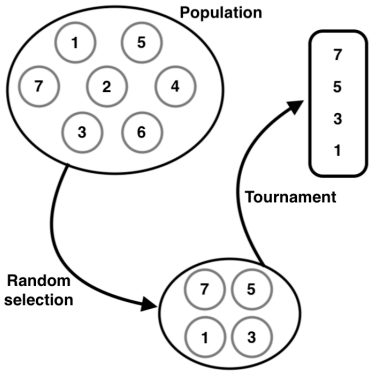

Figure 2.22: Tournament selection

2.4 Genetic Programming

Genetic programming (GP) [32] [33] is a form of an evolutionary algorithm, that evolves programs. Every individual in the population represents an executable program. The optimisation technique is the same as we described in Evolutionary Algorithms section, the only difference is that GP optimises tree structures.

The tree structure is a trivial representation that every reader should be familiarised with. Each node in the tree can represent terminal (constants, variables) or non-terminal (functions). Non-terminal nodes in the tree structure have an additional property called arity that defines the number of children. In our development, we only use functions of arity one and two, so we can consider our trees binary. When it comes to modifying a tree structure, it is important to be in control of the tree's size, because GP algorithms tend to bloat the tree out of desired proportion. In our case, we can control the bloat with tree's depth, since we only use binary operators the that also controls the width.

$$Nc = 2^{k+1} - 1; k \geq 1 \tag{2.10}$$

$$Dc = 2^{d} \tag{2.11}$$

(N_c is node count for the whole tree
Dc is depth count (number of neurons in the depth d))

2.5 Neural Network Encodings

To be able to work with NN and make modifications, there has to be a way of an effective encoding. This section is not focused on new approaches in the area of encoding since what we used in this thesis is already around for some time, and we only slightly modified it to work for our purpose. There are two main types of encoding direct and indirect and both approaches are important and still in use.

2.5.1 Direct Encoding

The name "Direct Encoding" is derived from the fact that this type of encoding provides a direct mapping from genotype to phenotype. Direct in a sense, that every network's property is unmistakably understandable from the encoding, both network's connections and their weights. Most often such encoding is represented by a matrix of $n * n$, where n is equal to the number of nodes in the network. Every number then describes the connection between the nodes. It follows the standard behaviour of graphs represented by a matrix. The main diagonal is for loop connections and under the main diagonal are the weights for backward connections. You can find an excellent explanation in the research paper for neural network topologies [34].

2.5.2 Indirect Encoding

The Indirect Encoding [34] [35] is used for problems, where we need to evolve larger networks. The problem with direct encoding is that it holds information about every potential connection, saved in the matrix $n * n$. Unless we use fully connected network, we store information about connections that do not exist, and for large networks, the matrix becomes unreasonably large. On top of that direct encoding does not offer the possibility to evolve any modularity or regularity [34].

2.5.2.1 Cellular encoding

Often used part of indirect encoding is cellular encoding. Gruau's (1993) Cellular Encoding (CE) method is inspired by the process of cell division in living organisms and used for NN evolution. The progress of network's developments is captured in a development tree. Development tree starts with a single node/root that symbolises the initial cell at the very beginning of the evolution. By using cell divisions, the tree is branched, and the cell evolves. There are various possible cell division operations. In our research, we used following genetic operators.

operator [arity] - description

SEQ [2]
Sequential division (has two descendants) new cell inherits mother cell's outputs, the input of a new cell is connected to the mother cell's output. Development instructions for the mother cell continue in a left subtree, while for the new cell in the right.

PAR [2]
Parallel division (two descendants) creates a new cell which has the same set of inputs and outputs as the mother cell. Again, development instructions for the mother cell continue in a left subtree, while for the new cell in the right.

REC [0]
This operation is a terminal state, and it introduces the ability of recurrent call. Instead of terminating the development, the genotype is processed from the root. This process of recursive call takes place immediately before continuing to any other operation.

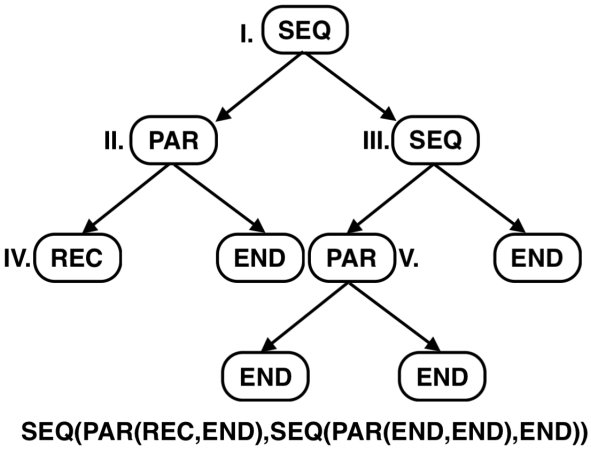

Figure 2.23: Genotype tree structure

The best explanation of cellular encoding is to show an example. Using the genotype in figure 2.23 we will show a step by step procedure that explains the behaviour of a genotype to phenotype mapping function. This exact example is used in the materials from Gruau [36], but the visualization of the mapping process is different.

Description of figure 2.24
I. `SEQ` - operator on node 0, create a new child [1]. Left subtree continues in mother cell [0] and right subtree in the new cell [1]
II. `PAR` - operator on node 0, create a new parallel child[2]. Left subtree continues in mother cell [0] and right subtree is finished.
III. `SEQ` - operator on node 1, create a new child [3]. Left subtree continues in mother cell [1] and right subtree is finished.
IV. `REC` - operator on node 0, The whole tree is called in recursive, `REC` operator is processed only once. This operator is processed in the figure 2.25.
V. `PAR` - after the `REC` operator we still need to process `PAR`.

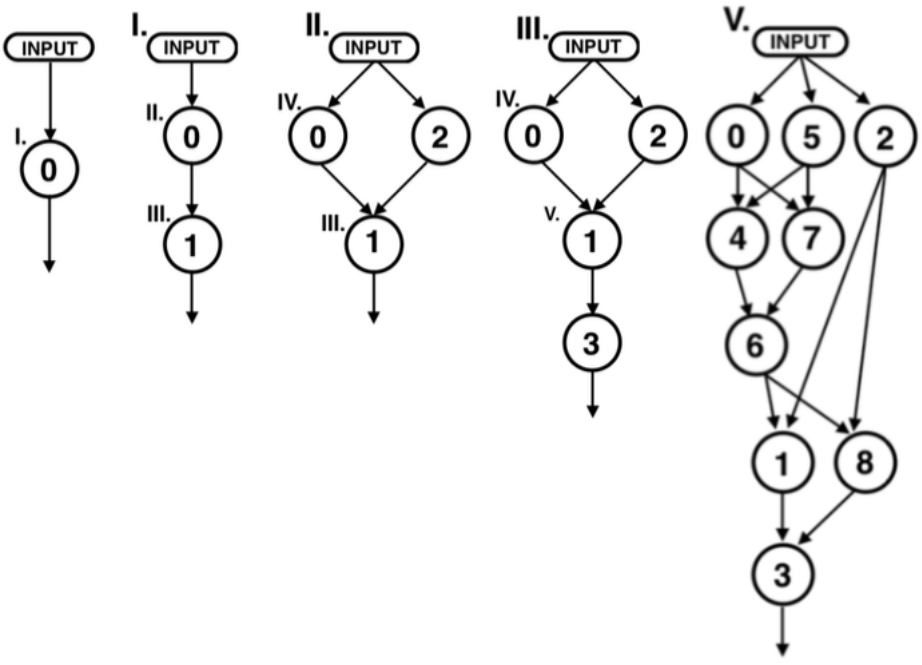

Figure 2.24: Genotype to phenotype mapping function

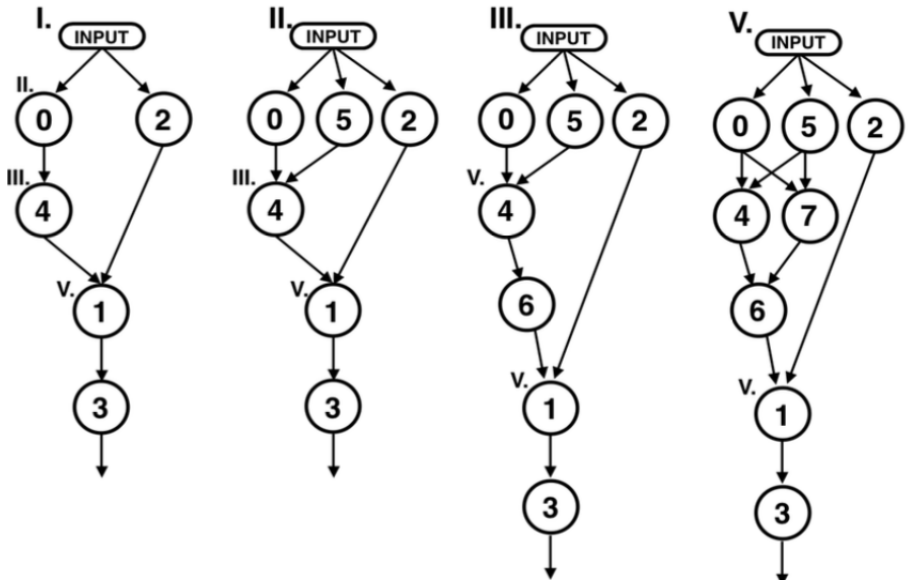

Figure 2.25: Genotype to phenotype mapping function, `REC` operator

Description of figure 2.25
II. `SEQ` operator on node 0, create a new child [4]. Left subtree continues in mother cell [0] and right subtree in the new cell [4].
III. `PAR` operator on node 0, create a new parallel child[5]. Left subtree is finished and right subtree is finished.
IV. `SEQ` operator on node 4, create a new child [6]. Left subtree continues in mother cell [4] and right subtree is finished.
V. `PAR` operator on node 4, create a new child [7]. Both subtrees are finished. The `REC` is complete.
V. `PAR` unfinished operation. Operator on node 1, create a new child [8]

2.5.2.2 Edge encoding

Based on the Gruau's cellular encoding, Luke and Spector published research describing some of the weakness of cellular encoding and providing solution that targets some of the mentioned issues, this new mechanism is called "Edge Encoding" [7] In Luke's and Spector's research paper, on edge encoding, is a nice paragraph that sums up some of the advantages of edge encoding over cellular encoding. Direct quotation from [7] :

> *Although cellular encoding is a powerful technique, it nonetheless has weaknesses. First, cellular encoding's chromosome traversal (breadth-first) and highly execution-order dependent operators can result in subtrees within the individual which, if crossed over to other individuals, would result in very different phenotypes than they expressed in the original individual. For many domains it may be more appropriate to use an encoding mechanism which better preserves phenotypes through crossover. Other disadvantages come from cellular encoding's use of graph nodes as the target of its operations: as cellular encoding modifies nodes, the edges multiply rapidly, but cellular encoding provides only a very limited mechanism, link registers, to label and modify individual edges. Additionally, the graphs cellular encoding produces tend to consist of highly interconnected nodes. This is useful for cellular encoding's primary focus, namely fully-connected graphs such as those found in feedforward networks or Hopfield networks. However, it may be less desirable in other domains.*

Edge encoding (EE) like cellular encoding (CE) is a tree-structured encoding that uses a direct graph as a phenotype. The biggest difference in EE is that grows the tree by modifying the edges instead of nodes like CE does. Another difference which is not that obvious is the fact that CE uses a breadth-first search for iteration in the tree, but EE uses a depth-first search. To demonstrate the underlying mechanisms of EE, we should introduce the basic set of operators. This set is used for a subgroup of EE that is called "Simple EE", and even though it cannot describe all the directed graphs, it is sufficient for any finite state automat NFA.

operator[arity] - description
DOUBLE [2] - Create an edge F (a, b)
BUD [2] - Create a node c. Create an edge F (b, c)
SPLIT [2] - Create a node c. Modify E to be E(a,c). Create an edge F(c,b)
LOOP[2] - Create a self-loop edge F (b, b)
REVERSE [2] - Reverse E to be E(b, a)

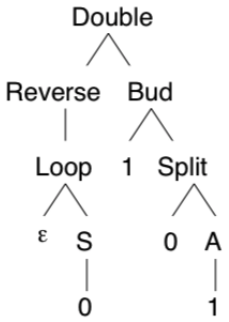

Figure 2.26: Edge encoding chromosome which describes an NFA that reads the regular expression ((01)*101 (image from another source [7])

Chapter 3

Program

An essential part of this thesis was to implement an evolutionary algorithm that is capable of finding an ideal network topology for given input dataset. The idea behind the whole algorithm is straightforward. We use the EA to evolve individuals and find better NN. Every individual can be represented by three different structures.

GENOTYPE
By using the cellular encoding, we create the genotype tree form 2.23 to represent the individual.

PHENOTYPE
Every genotype can be transformed by the mapping function 2.25 into phenotype. Phenotype structure is almost identical to the structure of the final NN.

NEURAL NETWORK
An NN form is a phenotype form with assigned inputs and output. The whole structure is defined by Keras objects, and it is the exact form that is used for training and evaluating the input dataset.

3.1 Analysis

3.1.1 Code Development

The programming language used for this thesis is Python. The primary reason for using a Python language is the fact that it provides the best possibilities for outsourcing essential parts from existing libraries. Libraries used in this thesis created a much easier environment for developing a functional product. The following list contains all the major libraries used in this project.

Tensorflow
TensorFlow is an open source software library for numerical computation using data flow graphs.
For installation follow the guide provided by TensorFlow [37]

Keras
Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow. It was developed with a focus on enabling fast experimentation with deep learning.
For installation follow the guide provided by Keras [38]

Deap
DEAP is an evolutionary computation framework for rapid prototyping and testing of ideas. It seeks to make algorithms explicit and data structures transparent. It works in perfect harmony with parallelisation.
For installation follow the guide provided by DEAP [39]

3.1.2 Environment

All the code included in this thesis was developed on macOS High Sierra platform using a Pycharm studio for IDE.

At first, the tests were calculated by the Apple MacBook Pro station. With parameters of a standard notebook.
MacBook PRO
processor : 2.7GHz Intel Core i5
Memory: 8GB 1867 MHz DDR3

After initial tests confirmed the functionality of the algorithm and all the major bugs were fixed, the setup proves itself insufficient. So we decided to find a new hardware for experimenting.

Server
processor : 2 x E5-2620v4 upto 3.00GHz (2 x 8 cores, each core handles 2 threads, 32 threads in total)
Memory: 6 X 16GB

3.1.3 Installation

To run the code that is included with this thesis, the environment needs the following list of dependencies.

- Python 2.7+

- TensorFlow [37] (we used virtualenv installation)
- Keras [38]
- DEAP [39]
- MatplotLib [40]
- GraphViz [41]

With all the necessary dependencies the project can be executed through the terminal by running the command `python -m src.Main` or from IDE by selecting `Main.py` as the main class.

3.1.4 Custom Genetic Operators

On top of the standard genetic operators used in cellular encoding we have added operators to modify the layer properties. The main reason behind adding custom operators is that we have changed the original idea where nodes in phenotype represents individual neurons to an implementation where a node represents a whole layer of neurons. Adding new operators for layer modification allows the EA to still have a full control over the structure and at the same time use more efficient encoding.

operator [arity] - description

DOUBLE [1]
Double the neuron count (one descendant) it doubles the number of neurons in the parent node/layer. This operation does not create a new cell, so development continues in the same cell.

HALF [1]
Halve the neuron count (one descendant) it reduces the number of neurons to one half in the parent node/layer. This operation does not create a new cell, so development continues in the same cell.

CONVOLUTIONAL OPERATORS
For developing the convolutional network, the library provides operators to set the properties of convolutional layers.

MAX P [1]
Enables the Max pooling operation after the convolutional step.

DROP 20 [1]
Applies the Dropout to the input and sets the fraction rate to 20%.

DROP 50 [1]
Applies the Dropout to the input and sets the fraction rate to 50%.

DOUB F [1]
Doubles the number of filters used in the convolutional step.

KER S [1]
Changes the size of the kernel filter by one, the filter is always squared and the stride is always one.

POOL S [1]
Changes the size of the pool filter by one, the filter is always squared.

3.2 Implementation

This section focuses on explaining the implementation details, and it provides all the necessary information to understand the functionality. It does not go into full detail on how everything works, but it explains the code flow with all the important sections. The structure of the whole project is depicted in figure 3.1. The core of this application is the Evolution class where most of the logic happens.

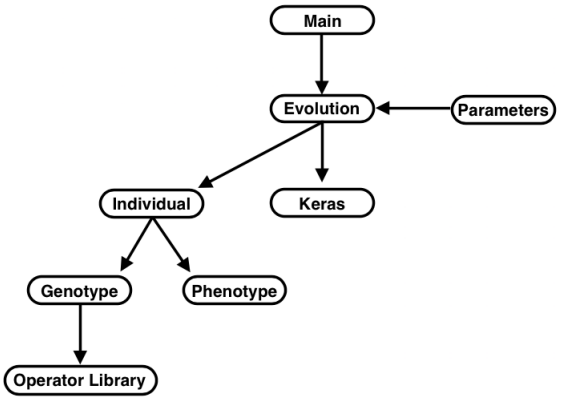

Figure 3.1: Implementation structure

3.2.1 Operator

[operator.py]
Purpose of this class is to have a possibility to easily add new genetic operators. We wanted to create a simple environment that allows very intuitive adding

of new operators and on top of that, we wanted to have only one place in the application to mention operator's name. In early versions, we had a hard-coded solution, and it created a lot of complications.

`Operator` class is an object that represents a genetic operator. Every operator has four attributes.

Name
Every operator is recognized by the name. This name is used in genotype representation and it has to be unique. In our application we used abbreviations.

Arity
Arity defines a number of descendants that the genotype operator creates.

Geno function
Based on the arity specified in the constructor, the initialisation process selects one of the predefined functions used for building the genotype tree.

Pheno function
Every operator introduces a new form/rule on how to modify the phenotype structure. This function is assigned in the constructor, and it has to be specified by the user.

3.2.2 Library

[operatorLib.py]
All the operators used in the genotype tree form are defined in the object `OperatorLib`. This object is the only place where an operator has to be defined, after correct initialisation the operator automatically appears in individuals.

The first step is to register the operator. We register new operator with the name `SEQ` with arity two. The second step is to create a phenotype functionality. To do that we need to summarise the expected operator's behaviour.

Sequential division (has two descendants) new cell inherits mother cell's outputs, the input of a new cell is connected to the mother cell, making it the only mother's cell output.

```
def addSEQ(self):
    Op = Operator('SEQ',2)
    def func(node, index):
        next = Pnode(index, node.neuron_count)
        next.addInput(node)
        next.copyOutputs(node)
        for n in node.outputs:
            n.inputs.remove(node)
        node.outputs = []
        node.addOutput(next)
        return node,next
    Op.setPhenoFunc(func)
    self.operators.append(Op)
```

Figure 3.2: SEQ operator definition

Phenotype function has two inputs, `node`(represents the mother cell) and `index` (index is the label for the layer name in the neural network). The code implements exactly the steps defined in the operator's description.

`node` represents the mother cell and `next` represents the newly created cell.

1. Create a new node `[pnode]` with label `index` and the same neuron count that mother cell has.
2. Create the input connection from the mother into the new cell.
3. Copy the output connections from the mother into the new cell.
4. Inform all mother's output nodes that they are no longer connected and link them to the new cell instead.
5. Remove all output connections from the mother cell.
6. Add output from mother cell into the new cell.
7. Return statements define where the next genotype operators should proceed.
8. Set the phenotype function reference by using `setPhenoFunc` after that add the operator to the list of operators.

Now the initialisation of the operator is complete, and the final step is to call the operator to register it. These calls are inside the function called `addOperators`.

3.2.3 Individual

[individual.py]
`Individual` is an object that helps with the evaluation process. It is important to state that this object is not used for evolution process. Individual used for the evolution is created by DEAP library, and it is explained in detail in the `Evolution` class description. `Individual` object has two important functions. One is that it has access to all genetic operators and in the `Evolution` class all the operators are registered in the DEAP environment via this connection. The other functionality is a phenotype converter wrapper. It provides all the necessary calls to obtain a phenotype from genotype.

3.2.4 Node

[gnode.py] and *[pnode.py]*
`gnode` and `pnode` are implemented as a building blocks for different tree structures. Both objects are used as a form of a linked list, meaning that every object has a saved reference to neighbour objects. To be able to iterate through the whole structure it is only necessary to hold the reference to its root node.

GNODE
Genotype node `gnode` is used for building a genotype tree, and the structure is quite simple. It has a `type` attribute that represents the node's functionality, and it is identical to the name of the genetic operator. Since we only use binary and unary operators the tree itself is at most in a binary form. Each node in the tree has a link to its left and right child. Unary operators only have a link to the left successor.

PNODE
Phenotype node `pnode` is used for building a phenotype structure. Unlike `gnode` the `pnode` can have multiple inputs and outputs. The connection is also represented by the reference to an object, and the only difference is that it is stored in an array. Besides all the connections stored in this object, there are also properties that define the neural network layer (neuron count, activation function, layer type, etc.) and functions that help to implement the phenotype functionality (copyInput, copyOutputs, divideNeuronCount, etc.)

3.2.5 Evolution

[evolution.py]
The `Evolution` class is the most crucial part of the application. All the logic behind the evolutionary process is hidden in this class. At the very early stages of the development, we decided to outsource evolutionary logic from existing library. We have ended up with a library called DEAP [39], it has

very detailed documentation, and it provides all the functionality we need for this project.

3.2.5.1 DEAP initialization

The first important section in the Evolution class is the initialisation of the DEAP environment. All the initialisation steps are in the figure 3.3.

```
pset = gp.PrimitiveSet("main", 0)
# --- Define functions ---
Ind = Individual()
for key, value in Ind.functions.items():
    pset.addPrimitive(value[0], value[1], value[2])

# --- Define terminals ---
pset.addTerminal('END')
if parameters.USE_MODULARITY:
    pset.addTerminal('REC')

# --- Define creator ---
creator.create("FitnessMin", base.Fitness, weights=(1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin,
pset=pset)

 # --- Define toolbox ---
toolbox = base.Toolbox()
toolbox.register("expr_init", gp.genFull, pset=pset, min_=1, max_=1)
toolbox.register("individual", tools.initIterate, creator.Individual,
toolbox.expr_init)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", evalGenotype)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("expr_mut", gp.genFull, pset=pset, min_=1, max_=3)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)
```

Figure 3.3: Initialization of DEAP enviroment

Define Non-terminals and terminals
By using the function `addPrimitive`, we can add all the non-terminals that we have previously linked to the individual from our library. After adding all the non-terminal operators, we add terminals, which in our

case is a string value 'END' and for modularity testing 'REC'.

Define creator
In DEAP library the creator is a helpful and simple way of dynamic class initialisation. It is used to define classes that help to run the EA. In our case we create a generic fitness function class and Individual class that uses a tree structure representation.

Define toolbox
Toolbox is used directly for the evolution process and it basically creates / register functions under certain aliases. The parameters of the `register` function are very simple [alias, function, ...arguments]. The process of registering may seem a little confusing at first, but in most cases, it has the same format

expr init
`Gp.genfull` generates a structure by using objects from `pset` with the size defined by min and max. In this case, we use only the size of one, for individuals in the initial population.

individual
`Tool.initIterate` calls the function `creator.Individual` with params `toolbox.expr init`. It means that calling a function `individual` creates an actutal individual with the shape defined by `expr init`.

population
`Tools.initRepeat` simply fills the container by using a `toolbox.expr init` function. Function `population` takes one argument and that is size.

evaluate
Registering the fitness function under alias `evolution`. `Evalgenotype` is a function defined in the code.

select
Assigning a selection strategy and the tournament size.

expr mut
The same process that is defined in `expr init`, only with different size parameter. The difference is that `expr init` is used for generating structures for initial population and `expr mut` is used for structures that are used for mutation process.

mutate

This section defines the mutation operation. `Gp.mutUniform` randomly selects a node in the tree and swaps it with another individual. The other individual is created by `expr mut`.

3.2.5.2 Fitness function and Mutation

DEAP initialisation will not work without defining the evaluation function because we registered it under the alias `evaluate`.

```
# --- Define Fitness function ---
def evalGenotype(individual):
    genotype = gp.compile(individual, pset)
    Ind.setGenotype(genotype)
    self.printGenotype(individual, genotype)
    result = Ind.getPhenotype(individual)
    krs = KerasConstructor(result)
    individual.score = krs.testAcc
    individual.trainAcc = krs.trainAcc
    scoreGen = krs.testAcc
    scoreGen = Ind.evaluatePhenotype(individual.height, krs.testAcc)
    return scoreGen,

# --- Define the mutation function --
def mutateWithLimit(ancestor):
    mutant = toolbox.clone(ancestor)
    toolbox.mutate(mutant)
    while mutant.height > BLOAT_LIMIT:
        mutant = toolbox.clone(ancestor)
        toolbox.mutate(mutant)
    del mutant.fitness.values
    mutant.fitness.values = toolbox.evaluate(mutant)
    return mutant
```

Figure 3.4: Fitness function and Mutation

Mutation - `mutateWithLimit`
`Toolbox.clone` creates a deep copy of an object and we use it to store the original individual. After that, the mutation process takes place. Newly created individual must satisfy the bloat condition otherwise the mutation is repeated. When the mutation is found, we call the evaluation process to obtain the new score.

Fitness function - `evalGenotype`
`EvalGenotype` is a function that performs all the necessary steps to find out Iindividual's score. The process is quite complicated and very time consuming, especially when we consider the fact that this code is called for every modified individual. The process of evaluation can be divided into four steps.

Genotype
The first thing the algorithm is doing is creating a genotype. The internal representation of DEAP individual is a string that contains the name of operators. Example:

$$PAR(PAR(END, END), SEQ(END, END))$$

All these operators were registered as functions at the very beginning of the DEAP initialization. The `compile` process executes all the functions, and that builds the genotype tree, an interesting fact is that the tree is built from leaves to the root.

Phenotype
With the structure stored in variable *genotype* we can build a phenotype that represents the final neural network. The algorithm performs breadth-first-search and calls the operator's phenotype functionality at every node in genotype tree structure.

Neural network
At this point, we have a network structure, and we have to call the Keras library that evaluates the neural network, this part is explained in Keras implementation.

Score
`EvaluatePhenotype` does the final step, it uses the calculated accuracy on NN and adds some custom penalisations for the layer count and the neuron count. The penalisations are designed to prefer smaller networks. In our thesis, we used it for some experiments, but all of the official testing use only model's accuracy.

3.2.5.3 Evolution process

The DEAP library offers so much functionality, and the code is very compact, for instance in one line we can simulate the whole evolution process, but in our case, we have decided to implement it on our own, step-by-step. The for-cycle is easily identifiable in the code, and it is the same for every population. Operations defined on individuals can be dived into two sections, modifying and tuning.

Modification

The modification is a process that shapes the population. For every individual in the population, we roll the dice and decide to mutate or not. Elitism is a standard operation defined in evolutionary algorithms, and the only difference is that our implementation is conditional. The swapping of best individuals from the previous population happens only if they are better than worst individuals in current population Code for modification is in figure 3.5.

```
# MUTATION
for ofIndex in range(len(offspring)):
    mutant = offspring[ofIndex]
    if random.random() < MUTPB:
        ancestor = toolbox.clone(mutant)
        offspring[ofIndex] = mutateWithLimit(mutant)
        self.printMutationChange(ancestor, mutant)

# ELITISM
for ind in range(ELITISM):
    potential = population[ind]
    if offspring[POPS-1-ind].score < potential.score:
        offspring[POPS-1-ind] = potential
```

Figure 3.5: Mutation and Elitism

Tuning

After initial experiments, we have decided to add one necessary feature along with one possible improvement. We were experiencing real issues with having duplicates in the population. The first occurrence appeared right in the initial population, that is caused by the low amount of terminals, non-terminals and fact that in the initial population only has individuals of size one. The second appearance was more serious. During the evolution process, the elitism functionality slowly started replacing bad individuals with the best individuals from the previous population. The algorithm was not able to find a better solution, and in combination with our conditional elitism (copy individual from the previous population only if the score is better), we ended up with a population that had only one type of individual. That is a sign of a premature convergence, and it is happening because the algorithm can not simply find a better solution. Instead of using some early stopping mechanism, that is able to recognise this premature convergence. We used a duplicate removing strategy to avoid such situation. The introduced solution can still allow the population to contain duplicates, but the probability is greatly reduced. Af-

ter sorting out the issue with duplicates, we still felt like we could push the algorithm a bit more. The final tuneup tries to mutate the best individual x times to see if mutating the best individual can result in finding a better solution (x is defined in parameters under constant).

```
# DUPLICATES
for outer in range(len(offspring)):
    for inner in range(outer+1, len(offspring)):
        if str(offspring[outer]) == str(offspring[inner]):
            offspring[inner] = mutateWithLimit(offspring[inner])

# TUNE UP
for index in range(parameters.ELITISM_TUNE_UP):
    best = toolbox.clone(offspring[0])
    newBest = mutateWithLimit(best)
    if newBest.score > best.score:
        offspring[POPS-1-ind] = newBest
        break
```

Figure 3.6: Removing duplicates and Tuning the individuals

3.3 Genotype to Phenotype

[phenoConverter.py]
Phenotype converter is a class where the actual mapping from genotype to phenotype takes place. It has three essential methods that are explained in this section, other methods in this class serve as assist functions with self-explanatory nature.

Method `resolvePheno` in Figure 3.7 is called with two arguments, where `root` is a [gnode] that represents the root of genotype tree and `individual` is [string] that contains the genotype encoded in string form. This method serves as an entry point for genotype conversion. In the first six lines, it creates the initial structure. Every network starts with the input layer `inputN` and one neuron layer `mother` connected to the input. The conversion itself is executed by the command `iterateThrough`. After this call, all the [pnode] nodes are created, and we only use breadth-first search for obtaining the correct order in which we later initialise the neural layers.

```
def resolvePheno(self, root, individual):
    self.individual = individual
    inputN = Pnode(0)
    self.input = inputN
    mother = Pnode(1)
    inputN.addOutput(mother)
    mother.addInput(inputN)
    self.nodeLib.append(inputN)
    self.iterateThrough(mother,root, 'basic')
    order = self.getOrderBFS(inputN)
    self.printPhenotype(self.individual, order)
    library = self.convertNodeLib(self.nodeLib)
    return library, order
```

Figure 3.7: Entry point for genotype to phenotype conversion

```
def iterateThrough(self, mother, root, level):
    backup = self.que
    self.que = Queue.Queue()
    self.que.put([mother, root, 1])
    while not self.que.empty():
        item = self.que.get()
        name = str(item[1].index)
        if item[0] not in self.nodeLib:
            self.nodeLib.append(item[0])
        if level == 'basic':
            item[1].rec_count = item[2]
        if item[1].isRECWeight():
            item[1].modRecCount()
        if item[1].isREC() and (level == name or level == 'basic'):
            item[1].rec_count -= 1
            if item[1].rec_count <= 0:
                item[1].type = 'END'
            else:
                self.iterateThrough(item[0], root, name)
        if not item[1].isFinal():
            self.iteratePheno(item[0], item[1])
    self.que = backup
```

Figure 3.8: Recursive processing

Method `iterateTrhough` in figure 3.8 receives three parameters, `mother`[pnode], `root`[gnode] and `level`[string]. The whole process of conversion is implemented in the same way as it is explained in the figure 2.24. To build the phenotype structure, the process requires two pointers. One points to the genotype tree node, that represents a genetic operator, and the second points to a phenotype tree node, a place where to apply the operator. The whole building process is sensitive to the operator order, so we are using a queue structure with the FIFO property (First-In-First-Out). The queue holds a list of three variables [pnode], [gnode] and [string] (two pointers and the string is an identifier for recursion). By using the `while` method, the algorithm iterates through all genetic operators and applies them based on the implemented conditions.

`if item[0] not in self.nodelib`
Append all newly created nodes [pnode] into `nodeLib` (list that holds the reference to all the pnodes).

`if level == 'basic'`
The *basic* level represent a first run through the tree structure. We have decided that operators that modify the number of recurrent calls are only recognized in the *basic* layer. Without this condition the `REC` operator would cause infinite loop.

`if item[1].isREC() and level == name or level == 'basic'`
The possibility of recurrent call creates complications. The biggest one was that we need to have a way how to count the number of iterations. If the `REC` operator has a weight of three, that means three recurrent calls in a row. We need to make sure that the counter gets lowered every time the recurrent call is processed. The real complication is that we must only decrease the correct `REC` counter since the tree can have more than one. For that reason, we are forced to push into the queue also a `REC` identifier. This identifier is a unique node name. In this `If` block we lower the counter and then either end the recursion or call another iteration.

`if not item[1]isFinal()`
For all non-terminals call the method `iteratePheno`, explained in following section.

First and last comment with backup `que` are used because of the recursive call. The process uses a queue in the global scope to be able to access it from the `iteratePheno` method. The `REC` operator has to be processed completely before any other genetic operator, and for that

reason, we have to use a new queue list to create the new scope.

```
def iteratePheno(self, node, genoNode):
    operator = self.phenoLib[genoNode.type]
    func = operator[0]
    arity = operator[1]
    result = func(node, self.gbIndex)
    if arity == 2:
        self.gbIndex = self.gbIndex + 1
    if arity == 1:
        self.que.put([result, genoNode.getLeft(), genoNode.rec_count])
    if arity == 2:
        self.que.put([result[0], genoNode.getLeft(), genoNode.rec_count])
        self.que.put([result[1], genoNode.getRight(), genoNode.rec_count])
    return
```

Figure 3.9: Genetic operator application

Method `iteratePheno` in figure 3.9 performs the actual operator's functionality. `PhenoLib` is an array that has access to all registered operators. So first step is to load the appropriate phenotype mapping function defined during the operator's initialisation. Then the algorithm performs the operation and obtains a `result` that is an array with two `pnodes` (child nodes). For all operators of arity two, we raise the global index which is used as a unique `pnode` identifier. A unary operator does not change the index because it only modifies properties and does not create new nodes. Based on the operator's arity the algorithm adds new triplets to the queue list. The last item in the array `genoNode.rec count` is simulating the inheritance of the recurrent counter.

3.4 Phenotype to Neural Network

[kerasConstructor.py]
In this file, we can find all the necessary code that performs the mapping from a phenotype to an actual neural network. A lot of code is directly affected by the switches defined in `Parameters.py`. This section only covers the most critical blocks.

Great think about using external libraries is that a lot of things that can be very tedious are already done for us and sometimes it is as simple as one line of code. Keras library already has prepared datasets and to load them, we

just call the load data function. In our application we use `MNIST` and `CIFAR10`. The actual dataset is divided into four parts x-train, y-train, x-test and y-test.

```
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

Figure 3.10: Dataset loading

x train - set of 60.000 train images (28x28 pixels)
y train - set of 60.000 image labels
x test - set of 10.000 test images (28x28 pixels)
y test - set of 10.000 image labels

Using a smaller dataset is not as trivial as it might seem because we need to divide the set perfectly so that every class has the same number of pictures and also keep track of all the correct labels. Luckily for us, this logic has already been solved by `scikit` and their `StratifiedShuffleSplit` function 3.11 that takes four arguments.

number of splits - a number of re-shuffling and splitting iterations
test size- a number of samples to create (takes [int] as a count or [float] as a ratio)
train size - a number of samples to create (takes [int] as a count or [float] as a ratio)
random seed - a number [int] that serves as a seed for the random generator

```
if train_size < 60000:
    sss = StratifiedShuffleSplit(n_splits=n_iter,
            test_size=parameters.OUTPUT_DIMENSION,
            train_size=train_size,
            random_state=seed)
    sss.get_n_splits(X_train, Y_train)
    for train_index, test_index in sss.split(X_train, Y_train):
        X_train, Y_train = X_train[train_index], Y_train[train_index]
```

Figure 3.11: Change the size of the dataset

With the desired number of samples, we need to shape the data into a form that is acceptable for our neural network. The best way is to shape the data into a form of a matrix and use the efficiency of matrix operations 3.12. For the `MNIST` case the `inpDime` is a 784 (28px*28px*1ch) that means that every

line in the matrix represents one picture. After that we use normalisation to convert all the values to [0,1], the reason we can do that is that in this case, the grey has no additional value for us. The last thing is to change the form of the labels from one integer value to list of ten distinct classes and their probabilities.

```
X_train = X_train.reshape(train_size, inpDime)
X_test = X_test.reshape(10000, inpDime)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
Y_train = np_utils.to_categorical(Y_train, num_class)
Y_test = np_utils.to_categorical(Y_test, num_class)
```

Figure 3.12: Prepare the data shape for NN

The building process 3.13 may seem a little complicated, but actually to build a deep fully connected NN it takes only a few steps. We initialise an array `modelArr` where we store all the Keras objects that represent individual layers. The first layer to add is an input layer with input dimensions defined by the user. We use the variable `order` to iterate through the phenotype, the content of this variable is an index of specific `pnode`, and the order is created by a breadth-first search iteration through the phenotype. The tricky part in building an NN in Keras is that we can create a layer only when all the input layers are already built. Through our experiments, we have found out that even though the order is BFS, it may not always follow the left-to-right behaviour. To avoid unexpected errors, we have implemented a safety feature that iterates through the phenotype in given order and chooses to build layer that has all the inputs already built. Once a layer is selected the iteration process starts from the beginning. To avoid an infinite number of loops, the building process allows only a certain amount of iterations.

Once the layer is ready to be build, the variable `isReady` is `True` and the `if` block is executed. The building process of an NN distinguishes two different layer types, with multiple entry points and with a single one. For multiple entry points, the algorithm first needs to connect all the inputs and then process the data. Keras offers a `keras.layers.concatenate` that concatenates all the layers, and that creates an input for a neuron layer. Every layer is initialised by the phenotype's node properties, neuron count and activation function. The newly created object is stored in `modelArr` and the order index is removed because the layer is done. During the iterative building process,

the algorithm stores reference to all output layers. Such layer is recognised by having no output layers.

The iteration cycle builds almost the whole network the only thing left to do after that is to format the output. The final layer always has to be predefined by the user to fit the expected result. In our case, we always want class probability, and for that reason, we use activation function `softmax` with output dimension that equals to the number of classes.

The building process for CNN 3.14 differs in few details. The initial input initialisation is a little bit different, but it is very simple and easily recognisable from the code. The biggest difference (see the figure 3.14) is the layer initialisation. Based on the parameters that are evolved during the evolution process the convolutional layer can have three different features, convolutional step, max pooling and dropout. It is absolutely essential to use the parameter `padding='same', strides=1`. These parameters ensure that the output dimension is always equal to the input dimension by using the zero padding. Without this key feature, the convolutional layer would never be able to concatenate layers with different dimensions. The last change is that the output layer is a little bit different from the deep fully connected NN. It contains some final flattening and normalisation, but the code is very simple and easy to understand.

```
modelArr = [None]*len(phenoArr)
modelArr[0] = (Input(shape=(inpDime,)))
output_layers = []
index = 0
safety = 3*len(order)
while len(order) > 0:
    if safety < 0:
        return 0
    safety -= 1
    index = index % len(order)
    order_index = order[index]
    layer = phenoArr[order_index]
    isReady = True
    for inp in layer.inputs:
        if modelArr[inp.index] is None :
            isReady = False
            break

    if isReady:
        if len(layer.inputs) > 1:
            layersToConcatenate = []
            for inp in layer.inputs:
                layersToConcatenate.append(modelArr[inp.index])
            x = keras.layers.concatenate(layersToConcatenate)
        else :
            for inp in layer.inputs:
                x = (modelArr[inp.index])
        modelArr[order_index] = Dense(layer.neuron_count,
                activation=layer.act_func)(x)
        if len(layer.outputs) == 0:
            output_layers.append(modelArr[order_index])
        order.remove(order_index)
        index = 0
    else :
        index += 1
# ----------- CREATE NETWORK -----------
input_layer = modelArr[0]
if len(output_layers) > 1:
    x = keras.layers.concatenate(output_layers)
else:
    x = output_layers[0]
output_layer = Dense(outDime, activation=actFuncExit)(x)
```

Figure 3.13: Bulding NN in Keras

```
im_dim = parameters.IMG_DIMENSION
filters = layer.filter_count
kernel_size = layer.kernel_size if layer.kernel_size < im_dim else im_dim
pool_size = layer.pool_size if layer.pool_size < im_dim else im_dim
dropout = layer.dropout
x = modelArr[inp.index]
x = Conv2D(filters = filters, kernel_size = kernel_size,
            strides=1, padding='same', activation = layer.act_func)(x)
if layer.maxPooling:
    x = MaxPooling2D(pool_size = (pool_size, pool_size),
            strides=1, padding='same')(x)
if layer.dropout > 0:
    Dropout(dropout)(x)
modelArr[order_index] = x
```

Figure 3.14: Initialisation of the CNN layer

```
 # ----------- MODEL EVALUATE-----------
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(loss=lossFunc, optimizer=optimizer, metrics=['accuracy'])
model.fit(X_train, Y_train,
            batch_size = batch_size,
            epochs = num_epoch,
            verbose = VERBOSE,
            validation_data = (X_test, Y_test))

# ----------- NETWORK OUTPUT ACCURACY-----------
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
testScore = score[1]*100
score = model.evaluate(X_train, Y_train, verbose=VERBOSE)
trainScore = score[1]*100
return trainScore,testScore
```

Figure 3.15: Model initialisation and NN evaluation

The last section is the part where we feed the model with all the important parameters and data. By using the metrics accuracy, the model will return the accuracy evaluated on given dataset. All the parameters are explained in the section 4.1

Chapter 4

Experiments

The last goal of this thesis is to perform experiments, and for that purpose, we have created a file called *parameters.py*, where all the relevant parameters are stored. The next section explains all the parameters.

4.1 Parameters overview

EVOLVE
[default: False]
The `EVOLVE` parameter is designed to help with debugging specific individuals. Default value is `False` but, when it is set to `True` the application does not perform an EA but it runs an `Evolution.Evaluate` function, that evaluates hard-coded individual.

USE CONVOLUTION NN
[default: False]
In later stages of development, we have switched from fully connected networks to convolutional networks. This is a switch that decides whether the phenotype is mapped to a CNN or DNN.

USE MODULARITY
[default: False]
Modularity switch allows that genotype can use `REC` terminals.

DATASET
This specifies the name of the dataset that is used for training and evaluating the NN quality. Program was prepared for MNIST and CIFAR-10 dataset.

TRAIN SIZE
`TRAIN SIZE` determines how big portion of the examples are used for

training. MNIST has 60.000 and CIFAR10 50.000 pictures for training. Using a smaller number for training has its impact on accuracy. Algorithm changes the number of samples by not just lowering the number of samples, but also making sure that all classes have the same amount of samples. Otherwise, the network might be overfitting for some input classes.

INPUT DIMENSION
`INPUT DIMENSION` determines the size of the input data. MNIST has a dataset of 60.000 images 28x28 pixels. Each individual is represented as a vector with 784 values (28x28=784). Use 784 for the input value. For CIFAR the calculation is 32x32x3 = 3072 (32 pixel image, 3 for 3 different color channels RGB).

OUTPUT DIMENSION
[default: 10]
`OUTPUT DIMENSION` represents the output size. For MNIST and CIFAR the output dimension is ten because for each image we want the probability of all the classes. This probability decides whether the classifier thinks the image is in the given class.

BATCH SIZE
[default: 128]
The `BATCH SIZE` is the number of training instances used in one iteration.

POPULATION SIZE
[default:]
`POPULATION SIZE` determines the number of individuals in a population. The size of the populations stays constant through all generations.

NUMBER OF GENERATIONS
`NUMBER OF GENERATIONS` affects the quality of the result. It takes some time for the evolution process to find out the ideal individual.

ELITISM
[default: PopSize / 5]
`ELITISM` is a process specific for evolutionary algorithm. It takes x best individuals from the previous population and swaps them with x worst individuals from new population.

MUTATION PROBABILITY
[default: 0.7]
Probability of performing a mutation on each individual. Algorithm iter-

ates through all individuals in given population and generates a number from 0 - 1, if this number is smaller then `MUTATION PROBABILITY`, then the individual is mutated.

BLOAT LIMIT
[default: 5]
After the mutation is done, new individual is inspected and the depth of its genotype tree representation must be smaller than the `BLOAT LIMIT`. If the new individual is bigger than the limit, the mutations process is repeated .

NEURON COUNT
[default: 100]
`NEURON COUNT` represents the default number of neurons for every layer. Throughout the evolution the number of neurons can be modified by certain genetic operators.

ACTIVATION FUNCTION
[default: ReLu]
This value represents the default activation function inside neurons for the whole layer. Throughout the evolution the type of activation function can be modified by certain genetic operators. The default activation function for our experiments was 'RELU'.

MAX NEURON THRESHOLD
[default: 10.000]
Maximal neuron threshold is a value that makes sure that some layers do not bloat out of proportion. Topology can be affected by having an extremely large layers that is capable of learning almost anything. For instance, imagine a layer with 100.000 neurons.

MIN NEURON THRESHOLD
[default: 100]
Opposite to maximal neuron threshold is minimum count. We do not want a layer that is not capable of learning. Imagine a layer with just one neuron.

ACTIVATION FUNCTION FOR EXIT
[default: SOFTMAX]
Depending on the nature of our algorithm, we need different output activation function. In our experiments we need class probabilities and for that the ideal function is 'softmax'.

OPTIMIZER
[default: ADAM]
`OPTIMIZER` defines the name of the optimizer used in learning process of the neural network in Keras. Our experiments used mostly "adam", only at the very beginning we used SGD.

LOSS FUNCTION
[default: categorical crossentropy]
A `LOSS FUNCTION` is used for mapping multiple values onto a real number, that represents some cost associated with the action . We used "categorical crossentropy", that is the computation of categorical crossentropy between predictions and targets.

LEARN EPOCH COUNT
`LEARN EPOCH COUNT` defines the number of epochs used for the learning process.

VERBOSE
[default: 0]
`VERBOSE` turns on and off log for Keras library. (0-OFF, 1-ON)

PRINT PERCENT THRESHOLD
[default: 97]
When learning process is finished, the network is evaluated by using the test dataset. The accuracy rating on the dataset must be higher then print percent threshold, only then is the network's topology model printed in a seperate file. Name of the file represents the achieved accuracy.

OUTPUT
Output variables determine paths for algorithms outputs.

4.2 Datasets

At the early stages of development, we only used XOR for training and testing the basic functionality. After XOR we used MNIST. MNIST obviously created a much better challenge, but even MNIST is not that difficult to learn. Almost any topology with a higher count of neurons, hundreds or thousands, is capable of achieving high accuracy. After achieving 98.3% on MNIST, we decided to use more challenging dataset, CIFAR10.

Both MNIST and CIFAR10 are included in Keras library and are very easy

to switch between.

4.2.1 MNIST

MNIST (Mixed National Institute of Standards and Technology) database is dataset for handwritten digits, provided by Yann Lecun's THE MNIST DATABASE of handwritten digits website [42].

The MNIST database is a set of images containing centered handwritten images in the range [0-9]. Every image has squared shape, and the size is $28x28$ pixels. A single pixel is defined by value from 0 to 255. (single channel image). The MNIST training set is composed of 60.000 images, and another 10.000 images are used for testing.

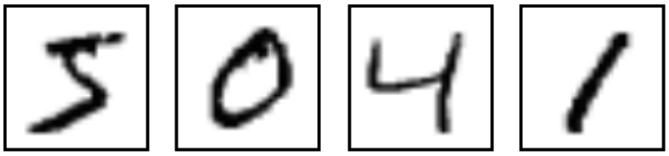

Figure 4.1: MNIST dataset example (image from another source [8])

State of the art approach is capable of learning the dataset with an accuracy 99.79% [13]. The most efficient solutions can be seen in a nicely created overview by Rodrigo Benenson at [8].

4.2.2 CIFAR 10

CIFAR (Canadian Institute For Advanced Research) [43] is a dataset constructed from colour images. Images can be divided into ten different classes - aeroplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

CIFAR database consists of images where every image has squared shape, and its size is $32x32$ pixels. A single pixel is defined by three different values from 0 to 255. (RGB 3 channel image). The CIFAR10 training set is composed of 50.000 images, and another 10.000 images for testing.

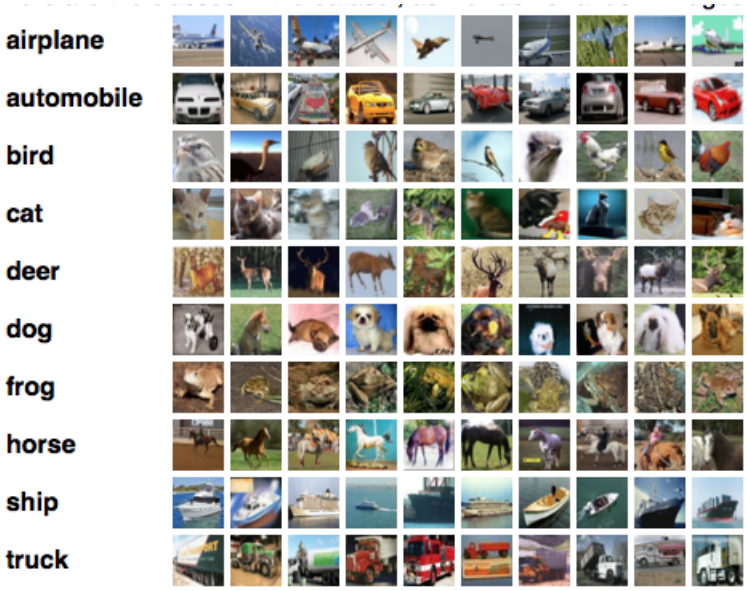

Figure 4.2: CIFAR 10 dataset example (image from another source [8])

State of the art approach in convolutional networks is capable of learning the dataset with an accuracy 96.53% [44]. State of the art approach in fully connected networks using ReLU activation function is capable of learning the dataset with an accuracy 54.91% [45]. The most efficient solutions can be seen in a nicely created overview by Rodrigo Benenson at [8].

4.3 Format

Data from all the experiments are a part of an attachment that is enclosed on CD to this Thesis. Every experiment has the same structure.

mutchanges.txt - list of all the performed mutations
geno.txt - list of all individuals in genotype form (graphviz structure)
trainAndTest.png - train and test accuracy graph
pheno.txt - list of all individuals in phenotype form
graph.png - accuracy graph
paremters.py - copy of all the parameters used for the experiment
time.png - runtime screenshot from terminal

4.4 MNIST experiments

All the parameters which value is not specified by the experiment are set to default. Default values are specified in parameters description section 4.1

4.4.1 Accuracy

Our experiments started with the MNIST dataset and a goal to achieve the highest accuracy possible, within a reasonable time frame. We used three different sets of parameters to find out more about EA's behaviour. Results are summed up in table 4.1.

Table 4.1: Accuracy on MNIST with full training set.

	Test 1.	Test 2.	Test 3.
Train size	60 000		
Population size	20		
Generation count	20		
Neuron count	100	500	500
Learn epoch count	5	5	15
RESULTS			
Best accuracy	98.41	98.37	**98.56**
Time (sec)	19010	42 549	78 843

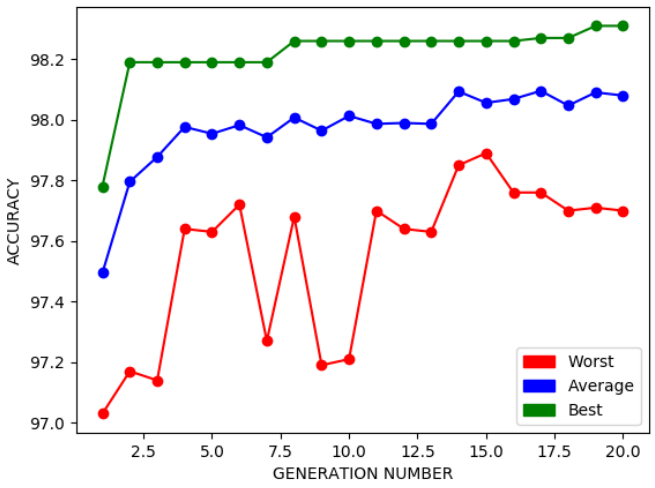

Figure 4.3: Accuracy graph [Test 1.]

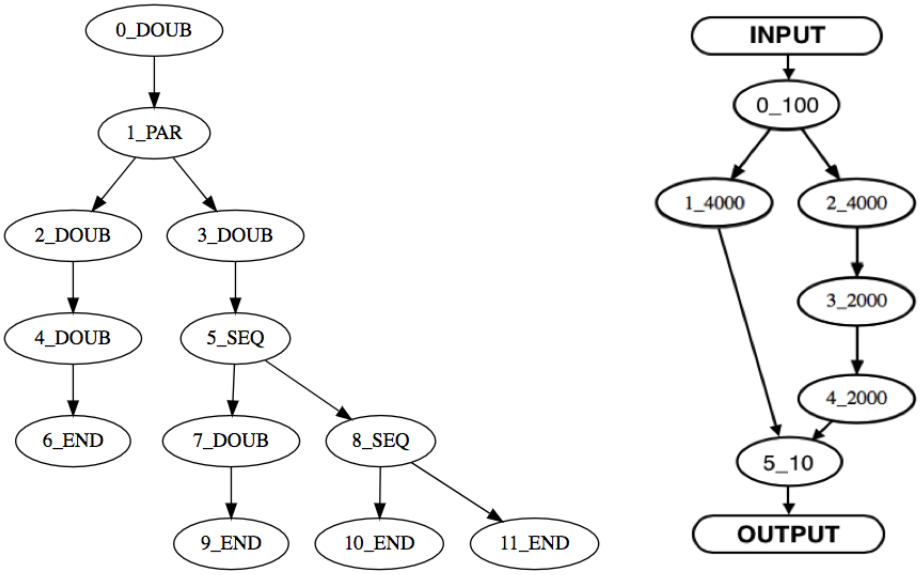

Figure 4.4: Genotype and phenotype [Test 3. (best solution)] (phenotype's node name is index-neuron count)

We have started out with relatively small evolution parameters (popSize = 20, genCount = 20). That may not seem like a lot, but there is a lot of computation behind the whole process. Roughly after five hours, we obtained the first result that scored 98.41% accuracy. We wanted to achieve higher accuracy, and so we have decided to change the default number of neurons in the layer (from 100 to 500). The result of the second experiment was quite interesting. The First thing is that the execution time changed quite dramatically from five to twelve hours. That is caused by the increased number of neurons and therefore an increased number of parameters in the network. The second more interesting thing is that the accuracy dropped. Even though higher neuron count provides a better foundation for learning, the algorithm was not simply capable to correctly set all the parameters within five learning epochs and thus the accuracy drop. Our reasoning behind the result suggests that with a higher number of learning epochs the network should perform better. When we used fifteen learning epochs, the algorithm did perform better and the accuracy improved to 98.56%. Ideally, we could have continued with this strategy to get closer to SOTA accuracy 99.79% [13], but it is simply not possible for us to use the same number of learning epochs that was used in SOTA approach (600-900), such a large experiment would take a big amount of time and also we believe the achieved accuracy is sufficient. Another major advantage of SOTA approach is using the Dropconnect strategy. Unfortunately, Keras interface does not provide this possibility.

4.4.2 Learning

After the accuracy testing, we have analysed the graphs. In figure 4.3 we can see that the evolution mechanism does find better solutions, but it seems like the impact is not that relevant. We thought that it is because the learning algorithm is so powerful and the dataset relatively simple. This combination creates an environment where almost any network with a reasonable amount of neurons is capable of scoring 90+% accuracy, and because of that, the topology becomes almost irrelevant. The best NN from `test 3.` is displayed in figure 4.4.

In this section, we wanted to prove the capability of EA by showing the ability to learn and to find better solutions over generations. The first strategy was to purposely handicap the learning algorithm and make the topology relevant again. We changed the number of learning epoch and training samples (to 6000) and ran the same tests again.

Table 4.2: Reduced training set to 6000 training samples.

	Test 4.	Test 5.	Test 6.
Train size	6000		
Population size	20		
Generation count	20		
Neuron count	100	500	500
Learn epoch count	20	20	50
RESULTS			
Best accuracy	96.29	96.52	**96.58**
Time (sec)	11 599	28 657	48 109

Table 4.2 shows the result of our handicapped set. The first thing we can notice is that accuracy is still relatively high, considering we dropped the size of the training set to 10% of the previous size, also the runtime is reduced quite substantially.

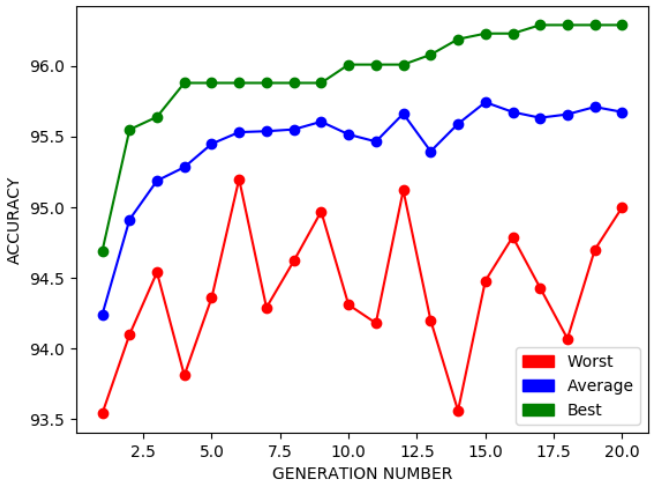

Figure 4.5: Accuracy graph with nice learning curve [test 4.]

Our focus in this section was to find proof that the EA is iteratively improving the solutions. In figure 4.5 we can see a beautiful green curve that is progressively growing, that means that the EA is indeed capable of finding better solutions and therefore not only our implementations works but the whole concept of using EA for NN is proving to be correct.

4.4.3 Scalability

Another significant milestone was to experiment with scalability. The scalability is an ability to evolve small NN and also big NN with a high number of neurons and more complicated structure. Only by looking at phenotype in figure 4.4 it is obvious that our algorithm has no problem to evolve large networks. `Test 3.` was not a particularly big experiment, and the phenotype has already approximately 12.000 neurons. Such amount of neurons significantly surpasses TWEAN structures. In our thesis, the ability to control the size of NN is encoded in five parameters.

POPULATION SIZE AND GENERATION COUNT
It is necessary to provide enough time for the EA to find and evolve the ideal individual. With a low number of individuals or generations, the EA is more likely to output small NN.

BLOAT LIMIT
Bloat limit is probably the only parameter that directly affects the size of the individuals. In our thesis, the bloat limit is controlling the depth of the tree, and since the tree is at most binary, it also controls the width.

LEARN EPOCH COUNT
Learn epoch count is a very important parameter that affects the size a lot. It is a fact that NN with more neurons and more layers is more likely to succeed and learn better, but it needs more time to learn. With the growing number of neurons also grows the number of parameters in the NN and if the learning algorithm does not have enough time for the learning process, then the fitness function will simply prefer smaller networks with higher accuracy.

TRAIN SIZE AND DIFFICULTY
The model of EA is finding the ideal individuals for given input dataset. When the dataset is too easy, the output NN is going to be small because even small NN can score high accuracy. We experimented with penalisation for the size, when we were trying to find the smallest NN possible, but then we have realised that this is already covered by the learn epoch count parameter. The evolution is going to find the best individual.

To prove the scalability even further we have selected an individual from the `Test 13.` to show that the algorithm has no problem to evolve large NN. The phenotype in figure 4.6 has almost 36.000 neurons with fully connected layers, and that is by no means a small network.

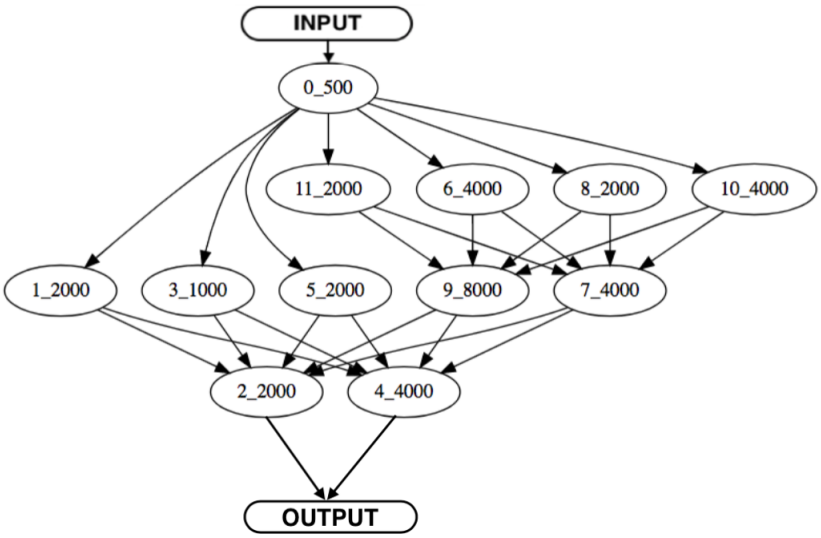

Figure 4.6: Large scale phenotype (36.000 neurons) [test 13.]

4.4.4 Modularity

Along with the scalability, we were also interested in modularity experiments. For this reason alone we created a `REC` operator and also `REC-D` and `REC-U`. The functionality of this operator trio was already covered in the implementation part. The quick summary is that `REC` calls the whole tree from the root node and `REC-D`,`REC-U` just specify how many times. We were curious to see if the network itself is going to create some modules with repetition. We used the same setup from previous tests 5. and 6., and the results are in the table 4.3.

Table 4.3: Testing modularity

	Test 7.	Test 8.
Use Modularity	True	
Train size	6000	
Population size	20	
Generation count	20	
Neuron count	500	
Learn epoch count	20	50
RESULTS		
Best accuracy	96.49	**96.58**
Time (sec)	70 808	135 890

The outcome of this test turned out very similarly to the previous tests. The big difference was in the runtime where the `test 8.` ran almost two days, but the accuracy score was very similar. After further investigation we have found out that the best individuals do not use the `REC` operator at all. It clearly provides the possibility to build bigger NN, understand that the `BLOAT LIMIT` does work only for genotype form and for that reason the NN of tests with `REC` operator are bigger. The actual limitation turned out to be the `LEARN EPOCH COUNT` again. To test out this theory, we lowered the number of individuals and increased the learning epoch count, to see `REC`'s behaviour.

Table 4.4: Testing modularity

	Test 8.
Use Modularity	True
Train size	6000
Population size	15
Generation count	15
Neuron count	500
Learn epoch count	100
RESULTS	
Best accuracy	96.51
Time (sec)	146 170

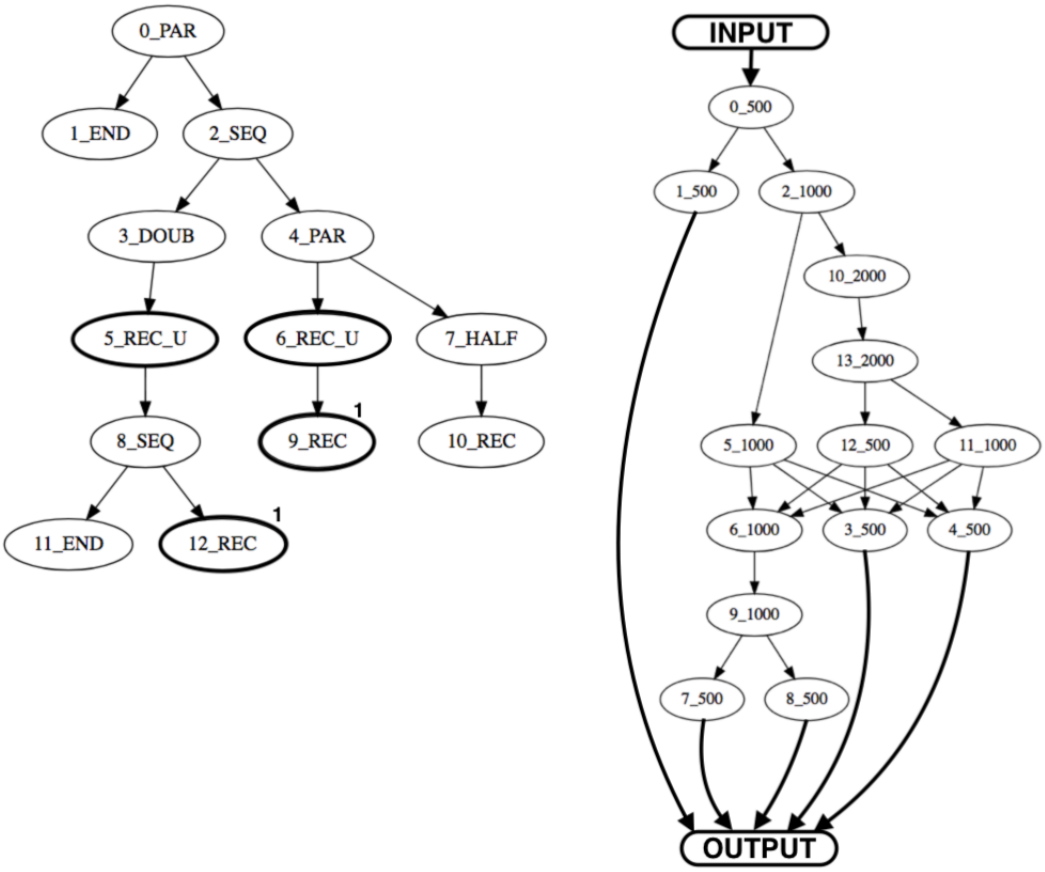

Figure 4.7: Genotype and phenotype with two `REC` calls [Test 13.]

The increased number of learning epochs resulted in genotypes with recurrent calls. Every `REC` terminal has by default a counter set to a zero. We have done this on purpose to force the EA first to set the counter in order to use the recurrent operator. In figure 4.7 we can see an individual with two recurrent calls. We have successfully implemented a possibility to encode modular and hierarchical architecture, but we are unable to go any deeper in experiments due to its complexity. Not only we would have to perform much larger experiments, but we would also have to implement a program that is capable of analysing the data and finding existing modularites in the NN.

4.5 CIFAR10 experiments

In previous experiments, we have used the strategy of reducing training samples to handicap the learning algorithm. For the purpose of showing the ability to learn it worked great. To find the limits of the EA algorithm, or at least get a little closer, we needed the algorithm to struggle. We tried to reduce the training set even further, but there is a limit to this strategy. We can obviously reduce the training set further, but then we experience the problem with overtraining, where the error for training set is close to zero and test set has significant error. That is caused by the difference in the set sizes. When we use a 600 training samples and 10.000 testing samples it is difficult for the model to generalise enough to perform well on the testing set. Even for 6000 samples, the model has 100% accuracy on the training set, see the figure 4.8, there is no more space for the EA to get better by decreasing the size of the training set because it will always be limited by the learning algorithm.

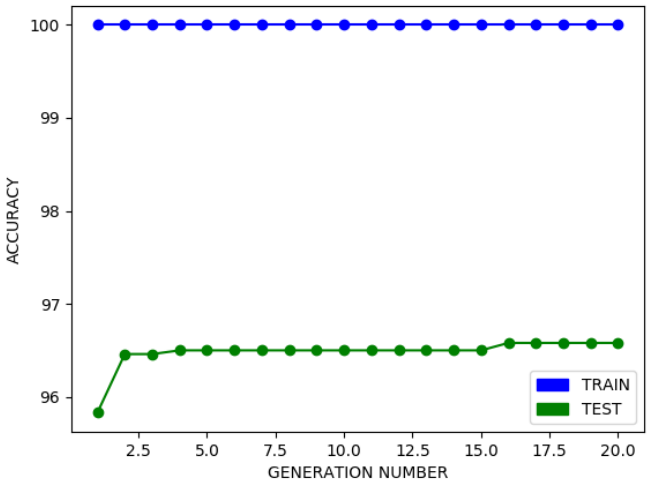

Figure 4.8: Train and Test accuracy[test 6.]

After few experiments, which are not included in this thesis, we have realised that we truly need a more complicated dataset. Ideally, we wanted to find a dataset that is more challenging to learn but has similar properties. CIFAR10 turned out to be the perfect candidate. To find out how complicated the dataset really is, we ran experiments identical to MNIST accuracy testing in table 4.1.

Table 4.5: Testing accuracy on CIFAR10

	Test 9.	Test 10.	Test 11.
Train size	50 000		
Population size	20		
Generation count	20		
Neuron count	100	500	500
Learn epoch count	5	5	25
Bloat limit	5		
RESULTS			
Best accuracy	49.36	48.98	**54.18**
Time (sec)	19 588	40 883	257 396

In table 4.5 we have the results from accuracy testing on CIFAR10. Immediately, we see that the accuracy dropped drastically comparing to MNIST, and it seems that CIFAR10 is truly way more challenging than MNIST.

For accuracy testing, we used the whole dataset (CIFAR10 - 50.000 samples), and the behaviour was similar to MNIST experiments in table 4.1. In the second test the accuracy went down, and when we have raised the number of learning epochs, it got better. Our DNN scored 54.18% accuracy, which seems like a bad result but SOTA approach scored 54.9% with ReLU activation function neurons. The CIFAR10 is so complicated that DNN is no longer able to perform well and it needs to be replaced by CNN. We could have probably get closer to SOTA accuracy with the higher number of learning epochs, we only used 25, but notice that the experiment took 257.396 sec - 72 hours.

4.5.1 CIFAR10 experiments CNN

Instead of running longer experiments to find better DNN we have decided to modify our algorithm to create CNN. Along with the change in layer type, we also added some new genetic operators that modify the properties of CNN. The results of the first experiment are in the table 4.6

Table 4.6: CNN accuracy testing on CIFAR10

	Test 12.
Train size	50 000
Population size	10
Generation count	10
Neuron count	500
Learn epoch count	30
Bloat limit	10
RESULTS	
Best accuracy	69.09
Time (sec)	402 466

The first CNN experiment achieved 69.09% which is nowhere close to SOTA accuracy, but it is much better than DNN's accuracy. We must consider that the EA has access to only a very basic CNN features and every feature must be triggered by a specific genetic operator. The score is not high, but even a simple CNN can outperform a DNN and the change from 54.18% to 69.09% is quite considerable. The only downside is the realisation that this experiment took almost 120 hours. In the figure 4.9 we can see the accuracy evolution.

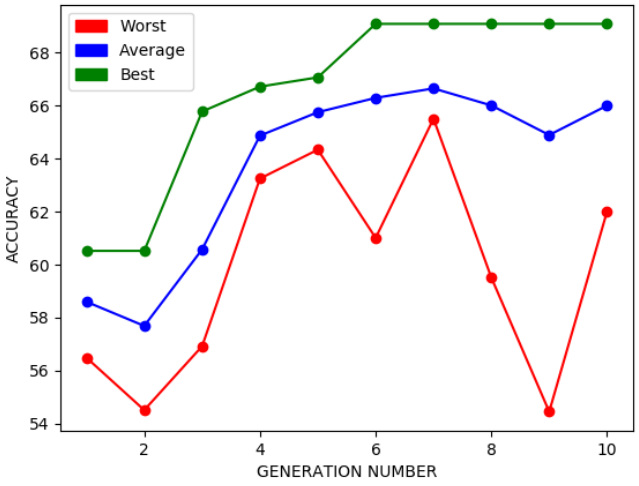

Figure 4.9: Accuracy graph with nice learning curve [test 12.]

Based on the results we have decided to repeat the experiment and change the default values for dropout (to 0.5) and max-pooling (always enabled). Although we are trying to achieve higher accuracy, we are well aware that to get anywhere close to SOTA [44] results we would have to implement the Fractional pooling and Dropconnect , and that is beyond the scope of this thesis.

Table 4.7: CNN accuracy testing on CIFAR10

	Test 12.
Train size	50 000
Population size	10
Generation count	10
Neuron count	500
Learn epoch count	30
Bloat limit	10
RESULTS	
Best accuracy	68.85
Time (sec)	803 714

The results in the table 4.7 showed that our decision to use a default dropout value 50% was probably not the ideal direction to take. Based on the genotypes with the high score, it appears that a reasonable value for dropout is 20%. Despite the fact that the overall accuracy dropped it is important to also notice the positive aspects. The most positive outcome of this experiment is the consistency. Even though we have changed the default parameters, the EA is still able to use the appropriate operators and reach almost identical accuracy. Another quite interesting thing is that the algorithm ran almost 224 hours and it finished without error or miscalculation, and that proves the stability of our implementation that is absolutely essential for future experiments.

Conclusion

The goal of this thesis was to study methods for Artificial Neural Network (ANN) architecture optimisation, with the focus on indirect encoding approaches with Cellular Encoding (CE) in particular. Based on the knowledge gained from the research, design and implement an algorithm allowing optimisation of deep neural network architectures and evaluate your algorithm experimentally.

Based on the knowledge we have gained during the extensive research, we have produced a knowledge base for the reader, that should provide all the theoretical background necessary to understand Evolutionary Algorithms (EA), Neural Networks (NN) and encoding for NN.

We have successfully implemented a fully functional program that is capable of finding the ideal neural network's topology for a given dataset. It is important to understand that the EA will not likely outperform State Of The Art (SOTA) approaches unless it has access to all the SOTA features. The EA only finds an optimal combination of all the features that are implemented.

We have spent most of our time experimenting with deep neural networks (DNN) to ensure the correct functionality of our algorithm. During the experimental phase, we have never encountered an error or any other malfunction. The development of our program for NN optimisation was very successful. We have met all the initial criteria and created an environment ideal for experimenting with DNN as well as CNN. The program alone has about 1500 lines of code in Python and combines three major libraries. We believe we have created a solid foundation for any future projects, concerning the EA for NN. To ensure a simple future extensibility, we have dedicated a whole section in our thesis to explain all the major implementation details. The guidance in the implementation section, along with the included code commentary should make the extensibility accessible to any developer.

The last goal of our thesis was to evaluate our algorithm experimentally. It took 2,2 M seconds in total to perform all the experiments included in this thesis. That is roughly 600 hours of calculation on a supercomputer with 16 cores, 32 individual threads and 64 GB of RAM.

Experiments performed in this thesis focused mainly on three different properties: accuracy, scalability, and modularity. The following section provides a summary of achievements and conclusion for individual properties.

Accuracy The accuracy in our experiments turned out very positive. We have successfully proven that the evolutionary algorithm is capable of finding neural networks with very high accuracy. Even though we have never got the exact accuracy that the SOTA approach has, we believe that it is possible as long as the algorithm has enough learning epochs and access to all the genetic operators to incorporate the SOTA features. In most of our experiments, we used only a fraction of the learning epochs that are used in the SOTA. To back up our belief, we have included graphs that prove that the EA has the ability to learn and iteratively find better solutions.

Scalalbility The introduction of this thesis reveals the problems with scalability of current approaches for NN optimisation and explains why it is essential to have the option to scale the size of the final NN. Our experiments have proven that the algorithm could easily produce NN with thousands of neurons and connections. The initial strategy, to use cellular encoding (Grau [36]), treat nodes as whole layers instead of single neurons (Estaban Real et al. [26]) and optimise the network by using the gradient method(Fernando et al. [24]), turned out to be very effective and functional. The size of the produced NN is entirely dependent on the input parameters, thus leaving the user complete control over the scalability. The only true limitation for large-scale NN is the time complexity.

Modularity The common belief is that the human brain contains a lot of repetitive modules. To incorporate any modularity in the NN, we have implemented the `REC` operator that is capable of recurrent calls and introducing the modularity in the NN. To allow the EA to control the modularity even further, we have added genetic operators to control the number of recurrent calls. Adding this functionality created substantial difficulties with the implementation. However we were able to make this feature 100% functional. We have dedicated a few experiments trying to find certain modularities, but we have quickly realised that to actually find and prove the modularity is beyond the scope of our thesis and it creates a challenge on its own. We have not proved any modularity in NN, but we have implemented a possibility to encode modularity and created a foundation for future research.

Our thesis consists of three major, equally important objectives that we have targeted individually and as extensively as it was possible within the time frame. The most difficult part of our thesis was indisputably the code development. The final code was not extremely complicated. The real challenge was understanding all of the individual segments prior to the development well enough to implement a functional program. During our research, we have struggled mainly with the time complexity of the experiments. Every time we found a bug in the program we had to erase all of the experiments and start over. Despite our wish for bigger and more complicated tests, we have ended up with no resources left. In the end, we are satisfied with the final form of our thesis and believe that it may prove beneficial for future experiments in the optimisation of NN.

Future Work

The experimental work with Artificial Intelligence is very entertaining and fascinating, mostly because a significant portion of the research has direct application in a real world. Neural networks are very interesting for so many reasons. Obviously, the excitement of the unknown is very high and since most of the major principles were not yet discovered there is a lot of room for possible breakthroughs. Another great reason would be the unlimited possibilities to modify NN and use them in so many different ways. Neural Networks create an enormous pool of opportunities with so much mystery around it that ideas for future work simply present themselves.

More experiments

The number of all the possible experiments is so large that is not possible to cover all the possible parameter combinations. However, with more available resources we would have covered the following points.

- - Implement all the relevant SOTA strategies for CNN and DNN.
- - Create Large-scale experiments to find the real limitations of EA.
- - Implement the early stopping functionality that recognizes the convergence of EA.
- - Increase the amount of NN evaluation to avoid possible inaccuracy caused by the wrong initialisation of SGD.

Modularity

Due to the time complexity of all the experiments we were unfortunately unable to go deeper in examining the possible modularities in FNN and CNN. The existence of modules in NN is obvious from the ratio of the human chromosome and the size of the neural structure in our brain. It could be interesting

to perform experiments with a higher number of learning epochs and then create an algorithm that can analyze the results and search for modularities. A possible way of detecting a modularity would be to implement an algorithm that can find the longest common substrings in the list of genotypes, in the string representation.

NORB dataset

We have only scratched the surface with EA for ANN. With more performance and more time we would most likely start to experiment with more challenging datasets. An interesting candidate could be a NORB dataset, the small set of normalised object sizes and uniform background. [9] 4.10. This dataset is intended for experiments in 3D object and shape recognition. It contains images of 50 toys belonging to 5 generic categories: four-legged animals, human figures, aeroplanes, trucks, and cars. The objects were imaged by two cameras under 6 lighting conditions, 9 elevations (30 to 70 degrees every 5 degrees), and 18 azimuths (0 to 340 every 20 degrees). The NORB dataset is marked as one of the more challenging datasets.

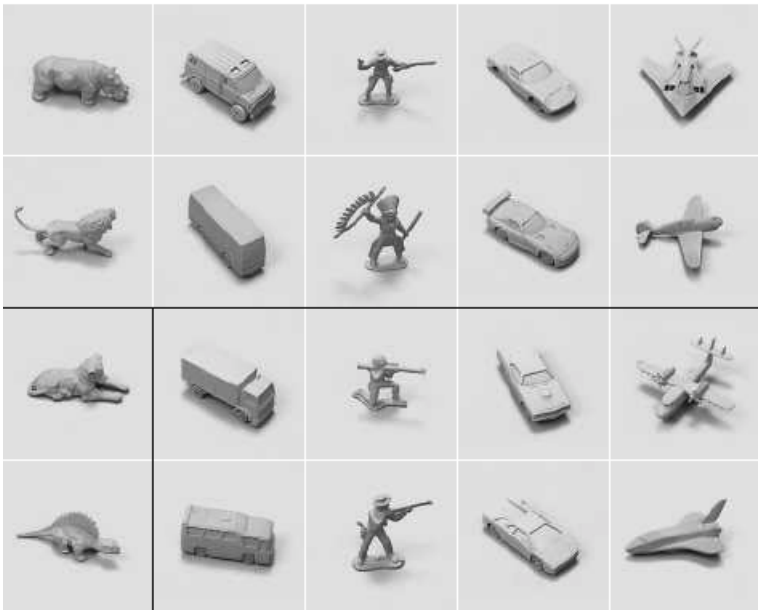

Figure 4.10: NORB dataset (image from another source [9])

Bibliography

[1] Moujahid[online], A. A Practical Introduction to Deep Learning with Caffe and Python. [cit. 2017-26-12]. Available from: http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe/

[2] Why are deep neural networks hard to train? [cit. 2017-18-10]. Available from: http://neuralnetworksanddeeplearning.com/chap5.html

[3] Gua, J.; Wangb, Z.; et al. Recent Advances in Convolutional Neural Networks. 2017, [cit. 2017-12-11]. Available from: https://arxiv.org/pdf/1512.07108.pdf

[4] Deshpande[online], A. A Beginner's Guide To Understanding Convolutional Neural Networks. [cit. 2017-26-12]. Available from: https://adeshpande3.github.io/A-Beginner27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

[5] Dertat[online], A. Applied Deep Learning - Part 4: Convolutional Neural Networks. [cit. 2017-26-12]. Available from: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

[6] What is gradient descent. [cit. 2017-16-7]. Available from: https://www.analyticsvidhya.com/blog/2017/03/introduction-to-gradient-descent-algorithm-along-its-variants/

[7] Luke, S.; Spector[online], L. Evolving Graphs and Networks with Edge Encoding. [cit. 2017-12-12]. Available from: https://cs.gmu.edu/~sean/papers/graph-paper.pdf

[8] [online], R. B. Classification datasets results. [cit. 2017-10-10]. Available from: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

[9] Huang, F. J.; LeCun[online], Y. THE NORB DATASET, V1.0. [cit. 2017-26-12]. Available from: https://cs.nyu.edu/~ylclab/data/norb-v1.0/

[10] Turing[online], A. M. COMPUTING MACHINERY AND INTELLIGENCE. [cit. 2017-26-12]. Available from: https://www.csee.umbc.edu/courses/471/papers/turing.pdf

[11] McCormick[online], R. Odds are we're living in a simulation, says Elon Musk (Recode's Code Conference). [cit. 2017-26-12]. Available from: https://www.theverge.com/2016/6/2/11837874/elon-musk-says-odds-living-in-simulation

[12] Simonyan, K.; Zisserman[online], A. VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION. 2015, [cit. 2017-10-12]. Available from: https://arxiv.org/pdf/1409.1556.pdf

[13] Wan, L.; Zeiler, M.; et al. Regularization of Neural Networks using DropConnect. [cit. 2017-10-11]. Available from: https://cs.nyu.edu/~wanli/dropc/

[14] He, K.; Zhang, X.; et al. Deep Residual Learning for Image Recognition. 2015, [cit. 2017-10-12]. Available from: https://arxiv.org/pdf/1512.03385.pdf

[15] He, K.; Zhang, X.; et al. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015, [cit. 2017-10-12]. Available from: https://arxiv.org/pdf/1502.01852.pdf

[16] P.Mandic, D.; Chambers, J. A. *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability*. ISBN 978-0-471-49517-8.

[17] Hochreiter, S.; Schmidhuber[online], J. Long Short-Term Memory. 1997, [cit. 2017-10-12]. Available from: http://www.bioinf.jku.at/publications/older/2604.pdf

[18] Moriarty, D. E. *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. Dissertation thesis, 1997, [cit. 2017-10-12]. Available from: http://nn.cs.utexas.edu/downloads/papers/moriarty.diss.tr257.pdf

[19] Gomez, F.; Miikkulainen[online], R. Incremental Evolution of Complex General Behavior. 1997, [cit. 2017-10-12]. Available from: http://www.cs.utexas.edu/users/nn/downloads/papers/gomez.adaptive-behavior.pdf

[20] O.Stanley, K.; [online], R. M. Evolving Neural Networks through Augmenting Topologies. 2002, [cit. 2017-20-10]. Available from: http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

[21] O.Stanley, K.; [online], R. M. A Taxonomy for Artificial Embryogeny. 2003, [cit. 2017-10-12]. Available from: https://pdfs.semanticscholar.org/2250/50ee487b17ced9f05844f078ff5345f5c9cc.pdf

[22] Stanley, K. O.; D'Ambrosio, D.; et al. A Hypercube-Based Indirect Encoding for Evolving Large-Scale Neural Networks. 2009, [cit. 2017-10-12]. Available from: https://pdfs.semanticscholar.org/e1df/8d30462995c299ac33e954dc1715d150cd83.pdf?_ga=2.203835155.262626272.1512929945-411408034.1512929945

[23] [online], K. O. Compositional Pattern Producing Networks: A Novel Abstraction of Development. 2007, [cit. 2017-10-12]. Available from: http://eplex.cs.ucf.edu/papers/stanley_gpem07.pdf

[24] Fernandoa, C.; Banarse, D.; et al. Convolution by Evolution. 2016, [cit. 2017-10-12]. Available from: https://arxiv.org/pdf/1606.02580.pdf

[25] [online], S. R. An Overview of Gradient Descent Optimization Algorithms. 2017, [cit. 2017-10-6]. Available from: https://arxiv.org/pdf/1609.04747.pdf

[26] Real, E.; Moore, S.; et al. Large-Scale Evolution of Image Classifiers. 2017, [cit. 2017-10-12]. Available from: https://arxiv.org/pdf/1703.01041.pdf

[27] Kingma, D. P.; [online], J. L. B. Adam: A method for Stochastic optimization. 2017, [cit. 2017-10-12]. Available from: https://arxiv.org/pdf/1412.6980.pdf

[28] Haykin, S. *Neural Networks and Learning Machines*. Pearson, third edition, [cit. 2017-18-10]. Available from: https://cours.etsmtl.ca/sys843/REFS/Books/ebook_Haykin09.pdf

[29] Krizhevsky, A.; Sutskever, I.; et al. ImageNet Classification with Deep Convolutional Neural Networks. [cit. 2017-10-9]. Available from: http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf

[30] [online], S. R. An Overview of Gradient Descent Optimization Algorithms. [cit. 2017-10-6]. Available from: http://ruder.io/optimizing-gradient-descent/

[31] Supervised learning tutorial [online]. [cit. 2017-10-9]. Available from: http://ufldl.stanford.edu/tutorial/supervised/

[32] [online], M. W. Introduction to Genetic Programming. 2001, [cit. 2017-20-10]. Available from: https://www.cs.montana.edu/~bwall/cs580/introduction_to_gp.pdf

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

[34] Fekiac, J.; Zelinka, I.; et al. A Review Of Methods For Encoding Neural Nework Topologiees In Evolutionary Computation. 2015.

[35] Gauci, J.; [online], K. O. Indirect Encoding of Neural Networks for Scalable Go. 2010, [cit. 2017-11-9]. Available from: http://eplex.cs.ucf.edu/papers/gauci_ppsn10.pdf

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

[37] How to install TensorFlow online manual [online]. [cit. 2017-2-11]. Available from: https://www.tensorflow.org/install/

[38] How to install Keras online manual [online]. [cit. 2017-2-11]. Available from: https://keras.io/installation

[39] How to install DEAP online manual [online]. [cit. 2017-2-11]. Available from: http://deap.readthedocs.io/en/master/installation.html

[40] How to install MATPLOTLIB online manual [online]. [cit. 2017-9-11]. Available from: https://matplotlib.org/users/installing.html

[41] How to install GraphViz online manual [online]. [cit. 2017-9-11]. Available from: https://pypi.python.org/pypi/graphviz

[42] [online], Y. L. The MNIST Database. [cit. 2017-10-11]. Available from: http://yann.lecun.com/exdb/mnist/

[43] [online], A. K. Learning Multiple Layers of Features from Tiny Images. 8.4.2009, [cit. 2017-10-11]. Available from: http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[44] [online], B. G. Fractional Max Pooling. 12.5.2015, [cit. 2017-10-11]. Available from: https://arxiv.org/pdf/1412.6071.pdf

[45] Kishore Konda, Z. L.; [online], R. M. How Far Can We Go Without Convolution: Improving Fully Connected Networks. 9.11.2015, [cit. 2017-10-11]. Available from: https://arxiv.org/pdf/1511.02580.pdf

Appendix A

Acronyms

AI Artificial Intelligence
ANN Artificial Neural Network
CE Cellular Encoding
CNN Convolutional Neural Network
CPPN Compositional Pattern Producing Network
CV Cross Validation
DNN Deep Neural Network
DPPN Differentiable Pattern Producing Networks
EA Evolutionary Algorithm
EE Edge Encoding
EP Evolutionary Programming
ES Evolutionary Strategy
ESP Enforced Sub Populations
FF Fitness Function
FNN Fully connected Neural Network
GA Genetic Algorithms
GP Genetic Programming
HN Hyper Neuro Evolution of Augmenting Topologies
HW Hardware
IDE Integrated Development Environment
LSTM Long Short Term Memory
NEAT Neuro Evolution of Augmenting Topologies
NFA Nondeterministic Finite Automaton
NN Neural Network
RGB Red Green Blue (color channels)
RNN Recurrent Neural Network
SANE Symbiotic Adaptive Neuro-Evolution
SOTA State Of The Art
SVM Support Vector Machine
TWEAN Topology and Weight Evolving Artificial Neural Network

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `Main.py` ..... code executable
- `thesis` ..... the directory of LaTeX source codes of the thesis
- `experiments` ..... the directory of LaTeX source codes of the thesis
  - `Test(1-14)` ..... output of individual experiments
- `thesis.pdf` ..... the thesis text in PDF format