UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

JONAS DA SILVEIRA BOHRER

Neuroevolution of Neural Network Architectures Using CoDeepNEAT and Keras

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Márcio Dorn
Coadvisor: Bruno Iochins

Porto Alegre
Dezembro 2019

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Profª. Jane Fraga Tutikian
Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Engenharia de Computação: Prof. André Inácio Reis
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço ao Prof. Dr. Márcio Dorn e ao doutorando Bruno Iochins pela orientação, correções e grande ajuda e influência em questões bibliográficas e tecnológicas. Agradeço também aos amigos, colegas de curso, professores e colegas de trabalho que influenciaram direta ou indiretamente a escolha de estudar esta área. Agradeço em especial à minha mãe e à minha irmã por serem exemplos de empenho e distinção, por me incentivarem na escolha de estudar na UFRGS e pelo contínuo apoio durante os anos. Finalmente, agradeço aos meus amigos mais próximos pelo apoio emocional e motivação durante a realização deste trabalho.

Neuroevolução de Estruturas de Redes Neurais usando CoDeepNEAT e Keras

RESUMO

Aprendizado de máquina é um extenso campo de estudo nas áreas de ciência da computação e estatística dedicado à execução de tarefas computacionais através de algoritmos que não requerem instruções explícitas, mas dependem do aprendizado de padrões em conjuntos de dados para o propósito de automatizar inferências. Grande porção do trabalho envolvido em um projeto de aprendizado de máquina é definir o melhor tipo de algoritmo para resolver um dado problema. Redes neurais, especialmente redes neurais profundas, são o tipo de solução predominante no campo de estudo de aprendizado de máquina, mas as próprias redes podem produzir resultados muito diferentes de acordo com as decisões arquiteturais feitas para as mesmas. Encontrar a topologia de rede neural e as configurações adequadas para um dado problema é um desafio que requer conhecimento de domínio e esforços de teste devido à imensa quantidade de parâmetros que devem ser considerados. O propósito deste trabalho de conclusão de curso é propor uma implementação adapatada de uma técnica evolutiva do campo de neuroevolução que consegue automatizar as tarefas de seleção de topologia e hiperparâmetros, usando um framework de aprendizado de máquina acessível e popular - Keras - como base, apresentando resultados e mudanças propostas em relação ao algoritmo original.

Palavras-chave: Redes neurais, Topologia de rede neural, Técnica evolutiva, Neuroevolução, Keras.

ABSTRACT

Evolution of Neural Network Architectures Using CoDeepNEAT and Keras Machine learning is a huge field of study in computer science and statistics dedicated to the execution of computational tasks through algorithms that do not require explicit instructions, but instead rely on learning patterns from data samples for the purpose of automating inferences. A large portion of the work involved in a machine learning project is to define the best type of algorithm to solve a given problem. Neural networks - especially deep neural networks - are the predominant type of solution in the field, but the networks themselves can produce very different results according to the architectural choices made for them. Finding the optimal network topology and configurations for a given problem is a challenge that requires domain knowledge and testing efforts due to the large amount of parameters that need to be considered. The purpose of this work is to propose an adapted implementation of a well-established evolutionary technique from the neuroevolution field that manages to automate the tasks of topology and hyperparameter selection, using a popular and accessible machine learning framework - Keras - as back-end, presenting results and proposed changes in relation to the original algorithm.

Keywords: Neural networks. Network topology. Evolutionary technique. Neuroevolution. Keras.

LIST OF FIGURES

Figure 2.1 A visualization of the components of a neural network. 19
Figure 2.2 A sigmoid function for varying slope parameter a. 19
Figure 2.3 Feedforward network structures 20
Figure 2.4 NEAT crossover operation. 23
Figure 2.5 HyperNEAT neuron configuration examples 24
Figure 2.6 CoDeepNEAT network assembling for fitness evaluation 26

Figure 3.1 Different views on the assembling of a Network 34
Figure 3.2 Crossover example. 35
Figure 3.3 Graph mutation examples. 37
Figure 3.4 Node content mutation effects on a network. 38

Figure 4.1 MNIST dataset samples 40
Figure 4.2 MNIST network features evolution 42
Figure 4.3 MNIST network scores progress 42
Figure 4.4 Training and validation metrics for MNIST 43
Figure 4.5 CIFAR-10 dataset samples 44
Figure 4.6 Best scoring network for CIFAR-10 at generation 1 45
Figure 4.7 CIFAR-10 network features evolution 46
Figure 4.8 CIFAR-10 network scores progress 46
Figure 4.9 Training and validation metrics for CIFAR-10 47
Figure 4.10 Best scoring network for CIFAR-10 at generation 40 48

LIST OF TABLES

Table 3.1 Example hyperparameter table. 31
Table 3.2 Example component parameter table for convolutional layers. 31

Table 4.1 Experiment hyperparameter table. 39
Table 4.2 Experiment parameter table for Convolutional layers. 40
Table 4.3 Experiment parameter table for Dense layers. 40

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
EC	Evolutionary Computation
EML	Evolutionary Machine Learning
EA	Evolutionary Algorithm
SGD	Stochastic Gradient Descent
GA	Genetic Algorithm
NN	Neural Network
CNN	Convolutional Neural Network
NEAT	Neuroevolution of Augmenting Topologies
HyperNEAT	Hypercube-based Neuroevolution of Augmenting Topologies
DeepNEAT	Deep Neuroevolution of Augmenting Topologies
CoDeepNEAT	Coevolution DeepNEAT
SANE	Symbiotic, Adaptive NeuroEvolution
ESP	Enforced Sub-populations
CoSyNE	Cooperative Synapse Neuroevolution
LEAF	Learning Evolutionary AI Framework

CONTENTS

1 INTRODUCTION **10**
1.1 Motivation **12**
1.2 Proposed methodology **13**
2 COMPUTATIONAL METHODS AND CONCEPTS **15**
2.1 Genetic algorithms **15**
2.2 Clustering algorithms **17**
2.3 Artificial neural networks **18**
2.4 Neuroevolution algorithms **21**
2.4.1 NEAT 21
2.4.2 HyperNEAT 22
2.4.3 DeepNEAT and CoDeepNEAT 24
2.5 Keras framework **27**
2.6 Chapter summary **27**
3 IMPLEMENTATION **28**
3.1 Initializing populations **29**
3.2 Parameter tables **30**
3.3 Initializing and managing species **32**
3.4 Neural network assembling **33**
3.5 Elitism, crossover and mutations **34**
3.6 Chapter summary **36**
4 EXPERIMENTS AND RESULTS **39**
4.1 Datasets **39**
4.2 MNIST experiment **40**
4.3 CIFAR-10 experiment **43**
4.4 Discussion **47**
5 CONCLUSION **50**
REFERENCES **51**

1 INTRODUCTION

Evolutionary computation can be shortly described as the use of evolutionary systems as computational processes for solving complex problems (JONG, 2016). As discussed in Jong (2016), although one can trace its genealogical roots as far back as the 1930s, it was the emergence of relatively inexpensive digital computing technology in the 1960s that served as an important catalyst for the field. The availability of this technology made it possible to use computer simulation as a tool for analyzing systems much more complicated than those analyzable mathematically.

Around the same time, machine learning emerged as a branch of AI proposing a more probabilistic approach to the search of artificial intelligence with systems that aimed to learn and improve without being explicitly programmed. Though it had an interesting premise, it only rose to its recent level of popularity in the last few decades, justified by the increasing availability of large amounts of data and computing resources required for the extremely complex algorithms the field proposed.

These two fields, evolutionary computation and machine learning, come together in what is usually described as Evolutionary Machine Learning (EML), which presents hybrid approaches that use algorithms from one field in the search of better solutions in the other. These resulting approaches have been widely applied to real-world problems in various situations, including agriculture, manufacturing, power and energy, internet/wifi/networking, finance, and healthcare (AL-SAHAF et al., 2019).

Out of the many branches in EML, one of the most widely studied is Neuroevolution (FLOREANO; DÜRR; MATTIUSSI, 2008), which is characterized by the act of building different aspects of neural networks through evolutionary algorithms (EAs) (BäCK, 1996). EAs are especially well suited for this task because of their remarkable ability to find good solutions in highly dimensional search spaces, such as exploring the multiple possibilities surrounding the definition of a neural network structure. Nonetheless, neuroevolution enables important capabilities that are typically unavailable to the more traditional gradient-based approaches like stochastic gradient descent (SGD) (RUMELHART; HINTON; WILLIAMS, 1986), raising the level of automation beyond the initial perspective of only setting weights to pre-configured network topologies. These new capabilities include the search of ideal hyperparameters, architectural parts and even the rules for learning themselves (STANLEY et al., 2019).

Of course, despite the great benefits described of using neuroevolution, gradient-

based methods still dominate many areas in machine learning where classification problems are easily differentiable with known topologies, as calculating weights through gradient descent methods is frequently more efficient than most evolutionary techniques. Still, neuroevolution finds its niches in domains where ideal topologies are yet to be discovered, such as in the meta-learning field (PAPPA et al., 2014) and the reinforcement learning field (SUCH et al., 2017), proving to be a scalable option in these domains (SALIMANS et al., 2017).

A great example of early neuroevolution approach successfully applied to a wide range of problems is the NeuroEvolution of Augmenting Topologies (NEAT) algorithm (STANLEY; MIIKKULAINEN, 2001), which is the starting point of this work. NEAT's main idea was to generate neural networks by associating similar parts of different neural networks through mutations (adding or removing nodes and connections) and crossovers (swapping nodes and connections) with a historical markings mechanism that simplified the identification of network similarities. Most importantly, it managed to implement a remarkable diversity preservation mechanism (named speciation), enabling the evolution of increasingly complex topologies by allowing organisms to compete primarily within their own niches instead of with the population at large.

But NEAT did not age well throughout the last decade, despite of its remarkable success in multiple use cases (STANLEY et al., 2019) - like the notorious discovery through NEAT of the most accurate measurement yet of the mass of the top quark, which was achieved at the Tevatron particle collider (AALTONEN, 2009) - where minimal structure was a lot more of a priority. Considering the state of art in modern deep learning research, the networks generated by the original NEAT algorithm are easily surpassed in dimension and consequently effectiveness when compared to networks used in currently popular problems like image recognition or text recognition, where thousands of nodes and hundreds of thousands to millions of connections are necessary to process information of complex data sources accordingly. The growth tendency in network dimensions comes directly from the availability of unprecedentedly cheap and powerful computing resources and large datasets as seen in the latest years, not only reducing the need for minimal structures in standard neural networks but also resulting in the perfect conditions for the practical usage and consequent popularization of all sorts of creative solutions involving different approaches to the traditional neural network topology, such as deep networks, convolutional networks, LSTM networks, graph networks, relational networks and more, contributing to yet one more weakness in standard NEAT.

This rapid popularization of different types of neural networks brought into the neuroevolution field challenges to try new techniques by combining and expanding these varied components into appropriate topologies and configurations to solve problems even more effectively, being also referred as the neural architecture search problem (ZOPH; LE, 2016). Adaptations in the traditional neuroevolution algorithms to face this evolving environment of possibilities and need for larger structures are largely popular at the moment (STANLEY et al., 2019).

These adaptations can be seen in succesful recent approaches like network generation and feature selection in highly dimentional datasets (Watts; Xue; Zhang, 2019), applications to the identification of gene expression patterns in cancer research (GRISCI; FELTES; DORN, 2019), reinforcement learning tasks (SALIMANS et al., 2017), learning policies for data augmentation (CUBUK et al., 2018) and in the multiple successors of NEAT throughout the years, like the notorious HyperNEAT (STANLEY; D'AMBROSIO; GAUCI, 2009), DeepNEAT and CoDeepNEAT variations (MIIKKULAINEN et al., 2017), which are the focus of this work.

1.1 Motivation

Although algorithms like NEAT and its variations have existing implementations directly from its authors, they consist of self-contained code that can be expanded but presents barriers in terms of directly connecting to other popular Machine Learning frameworks that researchers, students or scientists are more likely to be familiar with. Keras (CHOLLET et al., 2015), Tensorflow (ABADI et al., 2015), PyTorch (PASZKE et al., 2017) and other similar frameworks contain a number of functionalities that may come in handy when developing or analyzing machine learning models, which is a key element in validating the resulting models from neural architecture search algorithms in practical scenarios.

As of the moment of this work, both NEAT and HyperNEAT have been explored in public implementations [1,2] using these frameworks but few or lacking implementations of CoDeepNEAT have been found, presenting an opportunity to bring this method to a more accessible context. On the other hand, it is quite unclear from the original work (MIIKKULAINEN et al., 2017) whether the algorithm is suitable for practical applications

[1] https://github.com/crisbodnar/TensorFlow-NEAT
[2] https://pypi.org/project/neat-python/

and can be used in simple hardware environments or not. Verifying these aspects allows us to identify possible improvements to the base algorithm, such as different crossover operations, or mutation operations. Additionally, having an implementation based on a widespread framework facilitates these experiments for the overall scientific community.

With these aspects in mind, this work stablishes the implementation of an algorithm based on CoDeepNEAT in an accessible and popular framework and adapted based on different approaches seen in literature. The objective is to validate the complexity of the process of implementing such algorithm and to verify in practice if this type of solution is useful without massive hardware requirements. The framework of choice for this implementation is Keras, a user-friendly and high-level Python package for machine learning development and management, as opposed to the low-level and complex usability found in other popular options like directly using Tensorflow or Pytorch, for instance. Still, the backend used for Keras is Tensorflow.

1.2 Proposed methodology

The implementation requires the following fundamental working parts before initial testing:

- Genetic algorithm structure (to support iterations).
- Graph generation structure (to generate graphs for modules and blueprints).
- Module population management structure (to generate modules, manage speciation, fitness sharing).
- Blueprint population management structure (to generate blueprints, manage speciation, assembling, trainings, fitness evaluation and fitness sharing).
- Similarity metric and clusterization technique used for speciation (to compare individuals).
- Crossover technique used for reproduction (to evolve individuals through sexual reproduction).
- Mutations (to evolve individuals through asexual reproduction).
- Logging structure (to follow up the iteration process).

Additional changes are expected to be explored during development, such as:

- Alternative crossover operations.

- Alternative mutation techniques.
- Alternative similarity metrics.

Once the implementation is defined, initial experimentations will be done using the MNIST (Lecun et al., 1998) dataset. Final experiments will be executed using the CIFAR-10 (KRIZHEVSKY; NAIR; HINTON, 2009) dataset as done in the original paper to compare results and discuss the amount of time and computing power required for this approach considering academic use. Preliminary tests point that the required time for complete runs of the implementation using these datasets vary around 6 and 12 hours, considering limited hardware configurations and reduced parameters which will be described in the process. Most of the computation necessary is dedicated to training the networks for fitness evaluation during generations.

Chapter 2 will explore the necessary algorithms and concepts to develop the proposed work. Chapter 3 will detail the implementation and define the usage of the concepts described in chapter 2, while chapter 4 will describe the experimentation performed using the implementation and discuss the results, comparing them to the original CoDeepNEAT experiments and highlighting possible improvements.

2 COMPUTATIONAL METHODS AND CONCEPTS

This chapter briefly describes the most important algorithms and concepts related to the execution of this work and similar works in the EML field. Most recurrent terms are explained here and referenced in the next chapters.

2.1 Genetic algorithms

Genetic algorithms (GAs) are computational methods whose basic principle is the evolution of candidate solutions over iterations. Strongly based on behaviours of populations of biological organisms, they represent a predominant type of evolutionary algorithm (EA) in the evolutionary computation field, having been applied for decades in the solution of optimization problems since their first concrete description by J.H. Holland (HOLLAND, 1992).

As described in Beasley, Bull and Martin (1993), in nature individuals in a population compete with each other for resources such as food, water and shelter. Also members of the same species often compete to attract a mate. Those individuals which are most successful in surviving and attracting mates will have relatively larger numbers of offspring. Poorly performing individuals will produce few or even no offspring at all. This means that the genes from the highly adapted or "fit" individuals will spread to an increasing number of individuals in each successive generation. The combination of good characteristics from different ancestors can sometimes produce "superfit" offspring whose fitness is greater than that of either parent. In this way species evolve to become more and more well suited to their environment.

Adapting these concepts into a generalistic environment, standard GAs work with populations of "individuals" that represent solutions to a given problem. During multiple generations, these individuals are evaluated by a fitness function and are assigned a score. The scores are then used to decide what are the most "fit" individuals for reproduction or simply survival. Through reproduction, "offspring" is generated by combining the "genetic" features of their parents, occasionally generating better scoring solutions in the process. Individuals not fit enough for reproduction usually represent "bad" solutions, being less favored during the reproduction processes and commonly replaced by new individuals.

With the iteration of generations, genetic features that produce good solutions

are likely to spread across the population of individuals, being passed on to offspring generated through reproduction or simply by preservation mechanisms such as elitism, which consists in preserving a portion of the best scoring solutions over generations. GAs tend to converge over generations to optimum solutions, but require attention to questions such as keeping diversity (or, in other words, avoiding solutions to become extremelly similar genetic representations). To address these matters, additional techniques can be implemented, such as preserving groups of similar solutions as "species", or including "mutations" by altering genetic features in a defined fashion and consequently introducing changes to the populations. A pseudocode representing a standard procedure for GAs based on Davis (1991) can be seen in Algorithm 1, employing fitness evaluation, elitism, crossovers and mutations to individuals over generations.

Algorithm 1: Basic genetic algorithm structure

Data: N: number of generations, S: population size, E: elitism rate, C: crossover rate, M: mutation rate

Result: evolved candidate solutions

```
1  begin
2      initialize population with S individuals;
3      for individual in population do
4          evaluate fitness;
5      end
6      for generation in N do
7          apply elitism to E × S most fit individuals;
8          apply crossover to C × S most fit individuals;
9          apply mutations to M × S random individuals;
10         for individual in population do
11             evaluate fitness;
12         end
13     end
14 end
```

Being an extremelly popular algorithm branch, GAs have evolved over time to different approaches and have been succesfully applied to a wide variety of optimization problems, such as protein folding (UNGER; MOULT, 1993), selection of subsets of features to represent classification patterns (YANG; HONAVAR, 1998), optimum container placement in container loading problems (BORTFELDT; GEHRING, 2001), optimization of bank lending decisions (METAWA; HASSAN; ELHOSENY, 2017), increasing the longevity of wireless sensor networks (YUAN et al., 2017) or approximating the mass of the top quark, which was achieved at the Tevatron particle collider through NEAT (AALTONEN, 2009).

2.2 Clustering algorithms

Clustering algorithms are a class of methods that focus on grouping or classifying representations of data in a common environment to sets of members called "clusters". They are well suited for data domains with no pre-defined classes, generating classifications based on custom metrics that evaluate the distance or similarity between the data samples.

One specific clustering method used in this study is K-means (LLOYD, 2006), a popular partitioning algorithm based on specifying an initial number of groups, and iteratively reallocating objects among these groups until convergence. The algorithm assigns each data vector to the cluster whose center (also called "centroid") is nearest to the sample in the dimensional space that represents the data. The center is the average of all the points in the cluster, which means that its coordinates are the arithmetic mean for each dimension separately over all the points in the cluster (MADHULATHA, 2012). Algorithm 2 adapted from (MADHULATHA, 2012) describes the standard procedure for K-means.

Algorithm 2: K-means algorithm

Data: K: number of clusters, $samples$: vectors representing the data, $tolerance$: minimum improvement rate to continue processing
Result: Clusterization of the vectors

```
1  begin
2  |  Initialize the vectors of the K clusters (randomly, for instance);
3  |  while not converged according to tolerance do
4  |  |  for every sample vector in samples do
5  |  |  |  Compute the distance between the sample vector and every
   |  |  |    cluster's vector;
6  |  |  |  Re-compute the closest vector to the sample vector, using a
   |  |  |    learning rate that decreases in time;
7  |  |  end
8  |  end
9  |  Return the clusterization;
10 end
```

After clusters are defined, new samples of data can be integrated without the need of recreating the clusters. This can be done simply by using the *nearest centroid* method, which is the execution of Algorithm 2 from the $while$ step in line 3, without initializing the K clusters (HASTIE; TIBSHIRANI; FRIEDMAN, 2009).

Specifically in GAs, clustering algorithms can be applied to generate species or groups that share similar genetic information in the individual population. The species

then can be used in multiple strategies such as increasing population sizes without increasing the amount of fitness evaluations (Hee-Su Kim; Sung-Bae Cho, 2001), by evaluating one representative of the species at a time, or to ensure diversity by preserving different groups of solutions, as in the case of NEAT (STANLEY; MIIKKULAINEN, 2001).

2.3 Artificial neural networks

Artificial neural networks (or directly "neural networks") are machine learning models composed of multiple information-processing units called "neurons", which are connected in varying fashions to represent and approximate mathematical functions. Based on human biology, these models aspired to mimic the capability of the human brain to organize its structural constituents, known as neurons, so as to perform certain computations (e.g., pattern recognition, perception, and motor control) many times faster than the fastest digital computer in existence today (HAYKIN, 2009).

In practice, the neurons (also called nodes) that constitute neural networks are simple representations of mathematical functions that process the inputs they receive and output a value. They are composed of three main parts:

- A set of synaptic weights, each representing a value to be multiplied by the input signal of each connection they are assigned to.
- Summing junction, usually a linear combiner that sums the weighted input signals from the connections.
- Activation function, which models the output signal of the neuron to a defined amplitude. One of the most common activations functions, the sigmoid function, is represented in Figure 2.2.

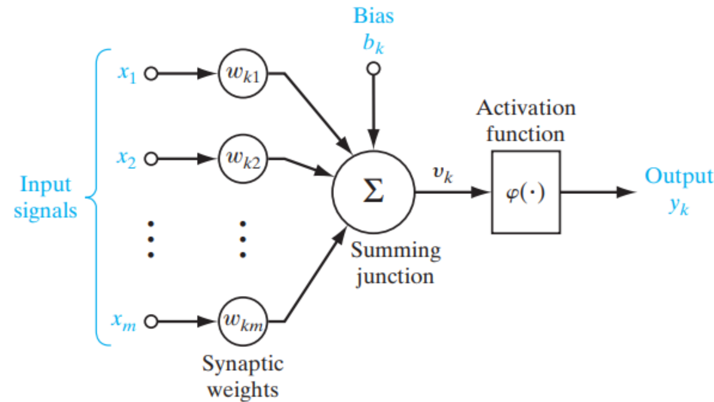
In addition to the described components, Figure 2.1 also displays the presence of a bias factor. The bias has the effect of increasing or lowering the net input of the activation function, depending on whether it is positive or negative, respectively (HAYKIN, 2009).

In mathematical terms, the neuron k depicted in Figure 2.1 is described in Haykin (2009) by the three equations:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{2.1}$$

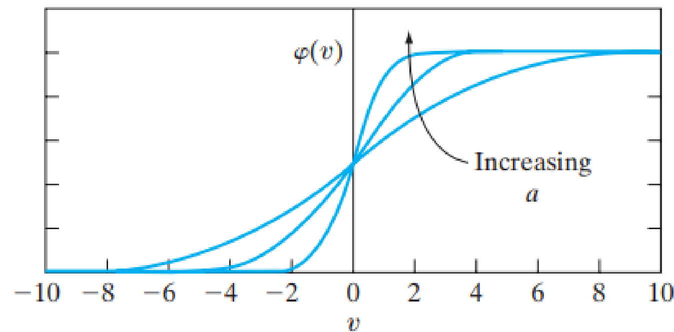
where x_1, x_2, ..., x_m are the input signals, w_{k1}, w_{k2}, ..., w_{km} are the respective synaptic

Figure 2.1: A visualization of the components of a neural network.



Source: (HAYKIN, 2009)

Figure 2.2: A sigmoid function for varying slope parameter a.



Source: (HAYKIN, 2009)

weights of neuron k, u_k is the linear combiner output due to the input signals;

$$v_k = (u_k + b_k) \tag{2.2}$$

where b_k is the bias and v_k is the resulting value from the summing junction after including the bias to u_k; and

$$y_k = \phi(v_k) \tag{2.3}$$

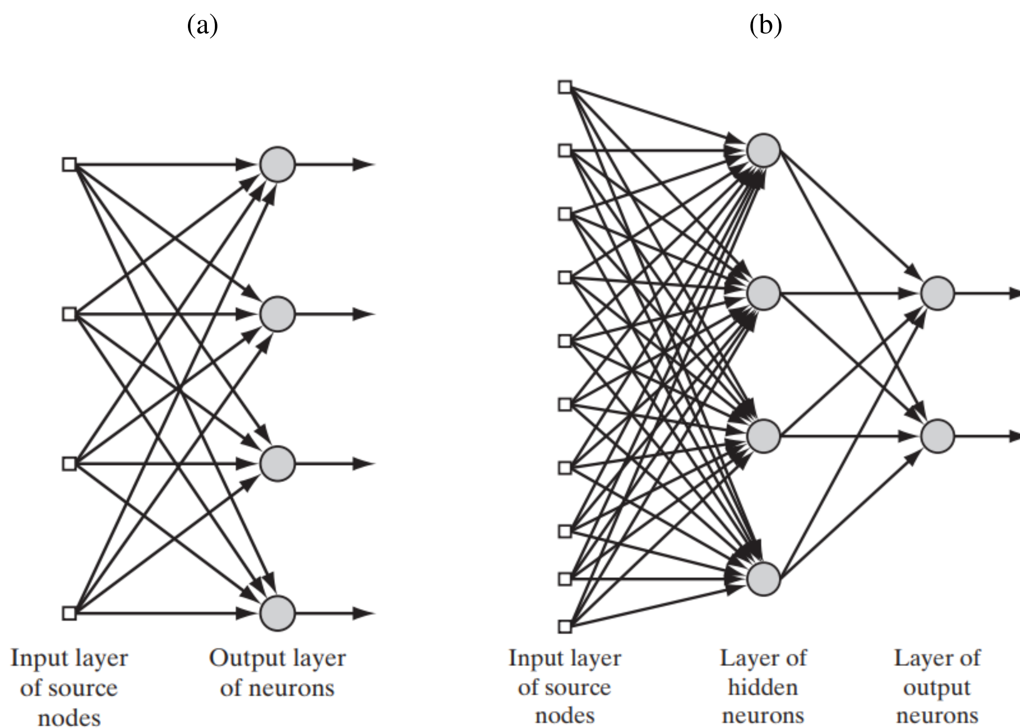
where ϕ is the activation function and y_k is the output signal of the neuron. One example of activation function is depicted in Figure 2.2, where a sigmoid function is applied to an output signal generating a new output signal contained inside a restricted amplitude.

The interconnections of the signals inside neurons and between neurons can be easily represented as signal-flow graphs (Mason, 1953), where the neurons are usually defined as "nodes". The connections can take multiple forms and are ruled by the synaptic weights. The synaptic weights that regulate these connections are subject to adjustments

through procedures called "learning algorithms", and represent the knowledge acquired during the learning process. The learning algorithms are constituted frequently by the act of exposing the model to data samples and modifying the synaptic weights as the model "learns" the patterns of the data. This procedure of exposing the model to data and evaluating its response is called supervised learning, the most common approach to "training" neural networks to date.

Network architectures resulting from the interconnection of nodes can be classified in multiple definitions, but the most important initial taxonomies for this study are the feedforward networks and the multilayer feedforward networks. Feedforward networks are simply networks organized in a way that input nodes connect to output nodes in a direct way to produce output signals, as in Figure 2.3a. Multilayered feedforward networks implement the same logic, but include nodes in divisions called "layers", representing sets of nodes that connect to other layers. Intermediate layers are commonly addressed as "hidden layers". An example can be seen in Figure 2.3b, where the input nodes connect to an intermediate layer, which connects to the output layer.

Figure 2.3: A classic feedforward network structure (a) and a classic multilayer feedforward network structure (b).



Source: (HAYKIN, 2009)

From this initial notion of stacking layers was created, for example, the cur-

rently extremelly popular *deep learning* branch (DENG; YU, 2014) in the machine learning field. Deep learning architectures are commonly characterized by the connection of multiple layers of neurons in neural networks that take profit of extracting different levels of patterns in the input data with each layer. These architectures have been applied to fields including speech recognition (AMODEI et al., 2016), image classification (RAWAT; WANG, 2017), natural language processing (GOLDBERG; HIRST, 2017), medical image analysis (LITJENS et al., 2017) and more, in some cases reaching levels of confidence superior to those of human experts (Ciregan; Meier; Schmidhuber, 2012).

2.4 Neuroevolution algorithms

Neuroevolution is a field of study dedicated to the generation and improvement of neural networks using evolutionary algorithms (EAs). Traditionally associated with the generation of neuron weights through evolution, current approaches associated with the field focus on multiple aspects of the construction of a network, such as learning their building blocks (for example activation functions), hyperparameters (for example learning rates), architectures (for example the number of neurons per layer, how many layers there are, and which layers connect to which) and even the rules for learning themselves (STANLEY et al., 2019).

One famous neuroevolution approach called Neuroevolution of Augmenting Topologies (STANLEY; MIIKKULAINEN, 2001) and some of its variations will be explored in the next subsections and exemplify some use cases that benefit from the capacity of EAs to find adequate solutions for extremelly complex problems, like the weight search, topology search and hyperparameter search topics for neural networks.

2.4.1 NEAT

Neuroevolution of Augmenting Topologies (STANLEY; MIIKKULAINEN, 2001), also called NEAT, is an algorithm designed for neural network topology construction. NEAT uses a genetic algorithm structure to generate small initial networks that evolve and grow over generations by adding neurons and connections and adjusting their weights to generate structures capable of performing well while keeping them minimal in size. This minimalist aspect of NEAT is one of its core differences to other neuroevolution algo-

rithms, as it focuses on only adding neurons or connections when they have an active impact in the network's performance (STANLEY; MIIKKULAINEN, 2001).

Starting with an initial population of small networks based on a common topology, NEAT evaluates changes to these networks iteratively by adding and removing neurons and connections across generations. The algorithm defines the genome that describes the nodes and connections by a mechanism called genetic encoding, used in the operations that modify the network structures through classic genetic algorithm operators such as crossover and mutations.

Mutation operators in NEAT work by adding nodes and connections or by disabling existing connections, avoiding changes that affect the functionality of the network. Crossover operators, on the other hand, are a much more complicated operation as it requires huge exchanges of genetic information that may cause resulting networks to not work correctly. To solve this, NEAT implements a historical markings mechanism, identifying nodes and connections with numerical identifiers. Parts of networks that share the same origin will share the same identifiers, thus the algorithm is able to recognize common structures in a simple way and exchange genetic informations without generating defective networks (Figure 2.4).

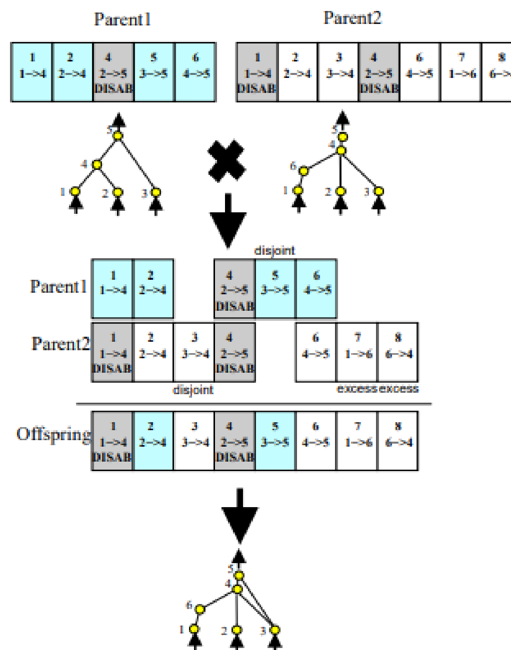
Before applying crossover operations between networks, NEAT must ensure that the chosen networks are compatible to a certain degree. The algorithm manages this situation by applying a speciation technique to the population of solutions, dividing it in different species generated by similarity, allowing organisms to compete primarily within their own niches instead of with the population at large. With this factor, different network topologies have a chance to evolve in their own pace instead of being instantly replaced by fast-converging networks that achieve better results in early generations.

After its conception, NEAT was used in a wide variety of use cases, specially in settings where small networks were required because of performance constraints, such as in robotics (D'SILVA, 2006), physics (AALTONEN, 2009), content generation for video games (HASTINGS; STANLEY, 2010) and more, as well as inspiring multiple variations of its core ideas, as explored in the next subsections.

2.4.2 HyperNEAT

HyperNEAT or hypercube-based NEAT (STANLEY; D'AMBROSIO; GAUCI, 2009) is probably the major extension of NEAT to date, having become a complex topic

Figure 2.4: NEAT crossover operation example. Although Parent 1 and Parent 2 look different, their historical markings (shown at the top of each gene) tell us which genes match up with which. Even without any topological analysis, a new structure that combines the overlapping parts of the two parents as well as their different parts can be created. Matching genes are inherited randomly, whereas disjoint genes (those that do not match in the middle) and excess genes (those that do not match in the end) are inherited from the more fit parent.



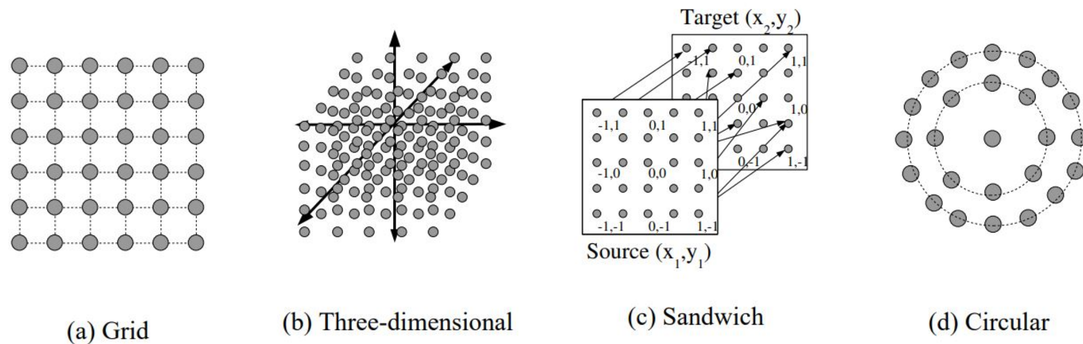
Source: (STANLEY; MIIKKULAINEN, 2001)

on its own and inspiring multiple approaches based on its success. Using connective CPNNs (Compositional Pattern Producing Networks) to represent connectivity patterns as functions of the Cartesian space (STANLEY; D'AMBROSIO; GAUCI, 2009), HyperNEAT exploits regularities in the data domain to evolve larger neural networks. In other words, the use of CPNNs enables indirect encoding, a principle based on attributing the discovery of patterns and regularities to the algorithm itself, relying as little as possible in direct encoding from designers.

Moreover, indirect encoding aims to access regularities not commonly addressed by conventional neural network learning algorithms, being capable of inferring constructions like, for instance, convolution. Examples of node configurations obtained using HyperNEAT are shown in Figure 2.5.

HyperNEAT also means a breakthrough from NEAT by allowing the evolution of much larger neural networks than the previous algorithm. By abstracting the mapping of spatial patterns generated by small CPNNs into connectivity patterns, HyperNEAT al-

Figure 2.5: Examples of configurations obtained using HyperNEAT. This figure shows (a) a traditional 2D substrate of connections, (b) a three-dimensional configuration of nodes, (c) a "state-space sandwich" configuration in which a source sheet of neurons connects directly to a target sheet, and (d) a circular configuration. Different configurations are likely suited to problems with different geometric properties.



(a) Grid (b) Three-dimensional (c) Sandwich (d) Circular

Source: (STANLEY; D'AMBROSIO; GAUCI, 2009)

lows the generated networks to be scaled in a customizable manner (up to millions of connections, for instance), better adapting to more complex applications such as evolving controller parts of legged robots (Clune et al., 2009), learning to play Atari games (HAUSKNECHT et al., 2013), combining SGD and indirect encoding for network evolution (FERNANDO et al., 2016) and even directly evolving modularity of components (VERBANCSICS; STANLEY, 2011).

2.4.3 DeepNEAT and CoDeepNEAT

Alternatively, a more recent path taken from NEAT was the DeepNEAT variation and, subsequently, the CoDeepNEAT variation (MIIKKULAINEN et al., 2017). Both cases, which are very tied, differ from HyperNEAT in that they don't aim to learn connectivity from geometric regularities in the data, but instead in assembling nodes based more directly in adaptations of the fitness evaluation process of NEAT.

DeepNEAT can be summarized as an extension of NEAT that considers entire layers as genes instead of considering single neurons when forming structures. The focus now is to define compositions of layers instead of picking neurons and their connections one by one, generating larger and deeper networks suited to solving larger scale problems then the ones NEAT was meant to solve in the past, while not minding the indirect encoding factor of HyperNEAT and considering pre-established components like different

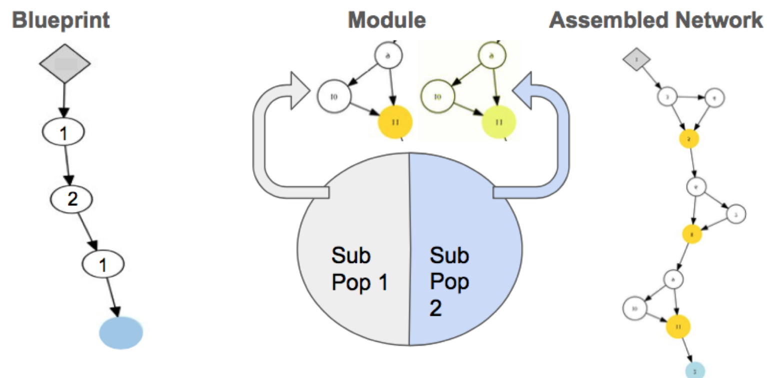
types of layers.

Similarly to the original NEAT algorithm, DeepNEAT follows a standard genetic algorithm structure to find its solutions: it starts by creating an initial population of individuals, each represented by a graph, and evolves them over generations. During these generations, the individuals are recreated by adding or removing structural parts (nodes and edges) from their graphs through mutation, while keeping track of changes through a historical markings mechanism. Using the historical markings, chromosomes are compared in every generation using a similarity metric, being classified into subpopulations called species. Each species is evaluated by the shared fitness of its individuals, calculated by a fitness sharing function. This shared score is used to evaluate the quality of the species in each generation. Finally, the surviving species evolve separately from each other through crossovers (exchanging genetic information) among its constituent individuals, and the next generation takes place.

The changes to the main algorithm of NEAT in how nodes now represent layers imply additional aspects that must be considered when defining a layer in DeepNEAT: what is the type of layer (convolutional, dense, recurrent), the properties of the layer (number of neurons, kernel size, stride size, activation function) and how nodes connect to each other. This is handled by considering a table of possible hyperparameters as the chromosome map for each node and an additional table of global parameters applicable to the entire network (such as learning rate, training algorithm, and data preprocessing) (MIIKKULAINEN et al., 2017). This makes the algorithm not only define topological information, but diverse network configurations more broadly.

Investing in the same perspective of focusing on layers instead of single neurons, CoDeepNEAT extends DeepNEAT by dividing the construction of a topology into two different levels: module chromosomes and blueprint chromosomes (Figure 2.6). Modules are graphs representing a small structure of connected layers. Blueprints are graphs representing a composition of connected nodes that point to module species, which can be assembled into complete networks by joining a sample of the module species pointed by each node. In other words, instead of evolving network species, CoDeepNEAT evolves module species and blueprint species which are assembled together into networks. The algorithm is inspired mainly by Hierarchical SANE (MORIARTY; MIIKKULAINEN, 1997) but is also influenced by the component-evolution approaches called Enforced Subpopulations (ESP) (GOMEZ; MIIKKULAINEN, 1999) and Cooperative Synapse Neuroevolution (CoSyNE) (GOMEZ; SCHMIDHUBER; MIIKKULAINEN, 2008).

Figure 2.6: A visualization of how CoDeepNEAT assembles networks for fitness evaluation. Modules and blueprints are assembled together into a network through replacement of blueprint nodes with corresponding modules. This approach allows evolving repetitive and deep structures seen in many successful recent DNNs.



Source: (MIIKKULAINEN et al., 2017)

Considering these two different chromosome types, CoDeepNEAT requires evolving separate populations for each one of them and scoring them individually. The genetic algorithm behind this is very similar to the one described for DeepNEAT, with the only effective changes being the population management and the assignment of scores by the fitness function. Now, instead of having one score for each individual and a shared score for its species, the score needs to be assigned both to the blueprint and to the modules used in its composition, and later shared between their respective species. At the same time, when modules are used in multiple blueprints, all the respective blueprint scores must be considered when assigning a score to a module (averaging them, for instance). Apart from these changes, CoDeepNEAT works very similarly to DeepNEAT while also bringing module evolution as an addition to the standard evaluation process.

The original paper presents results showing that CoDeepNEAT can indeed be implemented and generate high scoring networks for simple datasets such as CIFAR-10 and much more complex problems like image captioning using MSCOCO (CHEN et al., 2015). Of course, large datasets require longer training times and more computing resources, which lead CoDeepNEAT to be recently expanded to a platform called Learning Evolutionary AI Framework, or LEAF (Liang et al., 2019), taking advantage of cloud computing services to parallelize the algorithm for demanding use cases like pulmonar desease detection on high-resolution chest x-ray images (RAJPURKAR et al., 2017).

2.5 Keras framework

Keras (CHOLLET et al., 2015) is a popular [1] and high-level neural networks API, written in Python and capable of running on top of TensorFlow (ABADI et al., 2015) and other lower level frameworks. It was developed with a focus on enabling fast experimentation and allows for easy and fast prototyping (through user friendliness, modularity, and extensibility). Keras supports multiple types of neural network components [2], such as dense layers, convolutional layers, recurrent layers, dropout layers, and supports combinations of them.

The framework automatically manages resources such as CPU and GPU, making efficient use of them. It also has implementations of activation functions [3], optimizers [4], metric calculations [5] and procedures needed to manage training sessions with ease.

2.6 Chapter summary

In this chapter were presented the basic concepts of genetic algorithms and some of its applications, as well as a brief description of clustering algorithms, which can be used in genetic algorithms for aggregating solutions in similarity groups. In the subsequent sessions, were presented basic concepts of neural networks, the neuroevolution field and finally, in more detail, the NEAT algorithm and some of its most succesful variants. The next chapter will describe the usage of these concepts in the work proposed in the previous chapter.

[1] https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a
[2] https://keras.io/layers/about-keras-layers/
[3] https://keras.io/activations/
[4] https://keras.io/optimizers/
[5] https://keras.io/metrics/

3 IMPLEMENTATION

The general structure of Algorithm 3 is derived directly from the descriptions presented in the original NEAT paper (STANLEY; MIIKKULAINEN, 2001), the CoDeep-NEAT paper (MIIKKULAINEN et al., 2017) and its latest implementation, the LEAF platform (Liang et al., 2019).

Algorithm 3: Genetic algorithm structure for implementation

Data: hyperparameter tables, global parameter tables
Result: evolved candidate solutions

```
1  begin
2  |  initialize module and blueprint populations considering parameter tables;
3  |  initialize module and blueprint species;
4  |  for generation in generations do
5  |  |  for individual in individual population do
6  |  |  |  assemble respective blueprint;
7  |  |  |  generate Keras model;
8  |  |  |  score model;
9  |  |  |  assign score to blueprint and modules;
10 |  |  end
11 |  |  for species in module species do
12 |  |  |  calculate shared fitness;
13 |  |  |  apply elitism;
14 |  |  |  reproduce through crossover and mutation considering parameter
   |  |  |    tables;
15 |  |  end
16 |  |  for species in blueprint species do
17 |  |  |  calculate shared fitness;
18 |  |  |  apply elitism;
19 |  |  |  reproduce through crossover and mutation considering parameter
   |  |  |    tables;
20 |  |  end
21 |  |  speciate modules;
22 |  |  speciate blueprints;
23 |  end
24 end
```

Even though the algorithms are described in detail in the original work, a formal pseudo-algorithm is not specified, thus the procedure described in Algorithm 3 is an abstraction of that description. Specific genetic algorithm parameters such as elitism rate, crossover rate, mutation rate, number of allowed species, minimum and maximum number of individuals per species, etc., are not described in depth in the original work and are implemented as adjustable parameters, as are the tables of possible components of

modules (layer types, layer sizes, kernel sizes, strides, activation functions) and hyperparameters (learning rates, optimizers, loss functions).

The general procedures referenced in the algorithm are described in the following subsections, making references to algorithms described in chapter 2.

3.1 Initializing populations

Populations in genetic algorithms are groups of a certain type of individual that will be evolved over generations (section 2.1). Initializing populations requires a clear description of the involved entities that represent their respective individuals. In the case of the proposed algorithm, these individuals are module entities and blueprint entities, each one being initialized in their respective population (Section 2.4.3).

Module and blueprint entities are represented by a common graph design, only differing in the semantics of their nodes. Even though they represent different levels of abstraction of a single neural network, the graph structures of these entities are generated by a common graph generation procedure, as both , they need to follow a shared set of rules designed to correctly project a NN structure. At the same time, they need to respect Keras limitations when it comes to properly connecting layers.

The graph structures are generated according to the set of rules:

- The base structure of graphs are directed acyclic graphs that map the flow of signals from the input layer to the output layer.
- Graphs must follow the limitations stablished in parameter tables (as showed in Table 3.1), such as the allowed range of nodes.
- Graphs must have exactly one input node and one output node. Both nodes are used to connect graphs to other graphs, despite the internal structure of the graph. This connection takes place in Module to Module connections (in the case of Blueprint graphs) or Layer to Layer connections (in the case of Module graphs). This, in other words, implies the graph can only be connected through its input or through its output, not through intermediate nodes.
- Nodes in graphs must receive at most two input edges. One input edge directed to a input node means the origin output node can be directly connected to the input node, but more than one input edges connecting to an input node require the inclusion of a merge procedure between them, merging the edges into a single connection, as

Layers in Keras can only receive one input signal. For this purpose, Merge layers are implemented in Keras supporting the merging of two input signals each time, meaning that multiple input signals would require constructions of multiple Merge layers. For simplification purposes, this rule guarantees we only have one or two input edges at a node.

- Nodes can have multiple output edges. Multiple output signals don't require special treatment.

The graphs are managed in the implementation with the support of NetworkX (HAGBERG; SCHULT; SWART, 2008), an open-source framework for graph operations in Python, but the rules and graph structure definitions are implemented apart.

Along with structural definitions, graph creation must also handle definitions for the content represented by their nodes and edges. Nodes in these graphs are generated by a common routine designed for the creation of node content using custom content creation functions passed as parameters. In the case of modules, which represent assembles of layers, the standard node content creation functions are functions designed to generate new Layers. In the case of blueprints, which represent assembles of modules, the node content creation functions are functions designed to return existing modules from the module population.

For the last part of initializations, individuals are created to represent an instance of a blueprint to be evaluated. Instead of directly evaluating the blueprints, the individuals are instantiated to represent them, because a single blueprint can be trained with different combinations of hyperparameters not associated with the NN structure itself, such as the learning rate, optimizers, loss function or other parameters chosen from a hyperparameter table, explored in Section 3.2.

3.2 Parameter tables

Before assembling and trainings take place, a handful of parameters require management. Mostly addressed as hyperparameters, they relate to decisions made before training starts, like the loss algorithm used, the optimizer for the learning rate, the evaluation metrics to be considered during the training and so on. In this algorithm, additional parameters such as module or blueprint sizes, choices of layer types, configurations of layers and activation functions can be included in this group.

Table 3.1 examplifies some of these decisions. The table specifies parameters, the types of decisions required for them and their range of options, being yet another possible point of optimization in the algorithm. In this specific example, "Module size" is chosen as a "Random integer" ranging from 1 to 3.

Table 3.1: Example hyperparameter table.

Parameter	*Type*	*Options*
Module size	Random integer	[1, 3]
Blueprint size	Random integer	[1, 3]
Component types	Random choice	["Convolutional", "Dense"]
Loss functions	Fixed	["categorical_crossentropy"]
Optimizers	Random choice	["Adam", "RMSprop"]
Evaluation metrics	Fixed	["Accuracy"]

Source: The Author

Similarly, specific component tables can be stablished to define the configuration of layers during the module constructions. Table 3.2 examplifies the parameters considered during the instantiation of a convolutional layer. For instance, the table specifies that any convolutional layer will need to range their "Filters" as a "Random integer" between 32 and 64, while chosing "Kernel sizes" among 3, 5 and 7.

Yet another possible point of optimization in the algorithm, these tables could be evolved in their own populations during generations, similarly to modules and blueprints. This would allow the improvement of the usage of different hyperparameters in the trainings of different individuals, evaluating the table setups and evolving them over time. Though the current implementation only uses fixed tables as the ones represented in 3.1 and 3.2, a similar hyperparameter optimization procedure is implemented in LEAF (Liang et al., 2019), adding an additional dimension to the evolution process.

Table 3.2: Example component parameter table for convolutional layers.

Parameter	*Type*	*Options*
Filters	Random integer	[32, 64]
Kernel size	Random choice	[3, 5, 7]
Stride	Random choice	[2, 3]
Activation function	Random choice	["relu", "tanh"]
Dropout	Random float	[0, 0.7]

Source: The Author

3.3 Initializing and managing species

Speciation plays an important role in NEAT and its variants, ensuring the diversity inside populations over generations, as described in Section 2.4. The species must be initialized along with the populations so important procedures like elitism, crossover and mutation can take place.

The method used to approximate module and blueprint similarity to generate species is K-means. The original paper for CoDeepNEAT comments their usage of the same speciation schemes used for NEAT, but doesn't specify in detail how these schemes translate when dealing with the different representations of individuals used in CoDeepNEAT. This specific part was abstracted and implemented in this work using K-Means, whose functionality is described in Section 2.2.

K-Means is used to cluster module and blueprint graphs based on three main structural informations:

- the size of the network, such as the sum of the amount of filters in convolutional layers of neurons in fully-connected layers, or simply the amount of neuron connections;
- count of nodes, representing the amount of layers or modules in the graph;
- count of edges, representing the amount of connections between nodes in the graph.

This choice of clustering features can be easily changed in the implementation, as it is simply a parameter for K-means. This specific set of features evaluates only quantitative informations, but qualitative informations such as training scores (which would require training before evaluation) or other types of scores could be used. The clusterization generates an automatic (or custom) amount of clusters, which are used as species. The k-means implementation used is from the open-source framework Scikit-Learn (PEDREGOSA et al., 2011).

After species initialization, the nearest centroid method explained in Section 2.2 is used to assign new members to an existing species, allowing species to grow and change over generations. Centroids are calculated based on the features of the current species members every time new members need to be assigned to a species. New members are then assigned to the closest centroid. This way, members that already have an assigned species (e.g. members kept by elitism or new members that were assigned a species by any other method) still belong to their original species, but entirely new members are assigned

to the most adequate species according to the species' current demographic. An accuracy threshold can be specified so new species are generated in case new members don't fit the existing centroid with a satisfactory proximity. The nearest centroids implementation used is from the open-source framework Scikit-Learn (PEDREGOSA et al., 2011).

3.4 Neural network assembling

Transitioning the graphs to Keras model representations is required to take profit of the training procedures available in the Keras framework. After modules and blueprints are created, they need to be assembled into a unique graph, which is subsequently processed, node by node, creating the respective Keras layers and connecting them one by one following a topological sorting [1].

The transition scheme implemented handles the necessary interactions between layers, such as including Merge layers between two inputs directed to one layer, or adding Flatten layers to adjust the connections between Convolutional and Dense layers [2]. After the model is completelly connected and a set of hyperparameters is set, it is trained using the standard Keras training functions and a specified dataset. An assembled graph containing joined blueprint and module informations can be seen in Figure 3.1c and its respective network can be seen in Figure 3.1d.

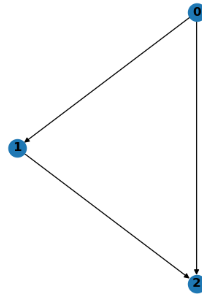
Figure 3.1d shows an example assembled Keras network generated by the algorithm. This network is based on the assembled graph shown in figure 3.1c, which is structured by the network's blueprint shown in figure 3.1b. Figure 3.1a shows the graph structure for the common module used by the three intermediate nodes in the blueprint graph in figure 3.1b.

The resulting scores from the Keras scoring procedures are then extracted from the trained model and assigned to the individual and its respective blueprint, which propagates them to the underlying modules involved. This score is used in the evaluation procedures to decide which entities survive elitism and which entities are candidates for reproduction in crossover or mutation schemes.

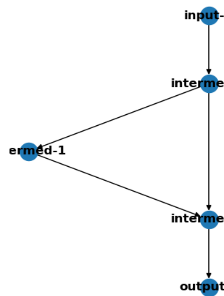
[1]https://networkx.github.io/documentation/stable/reference/algorithms/dag.html
[2]https://keras.io/layers/core/

Figure 3.1: Different views on the assembling of a Network.

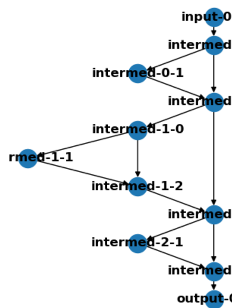
(a) Module graph structuring the connections of 3 different layers.



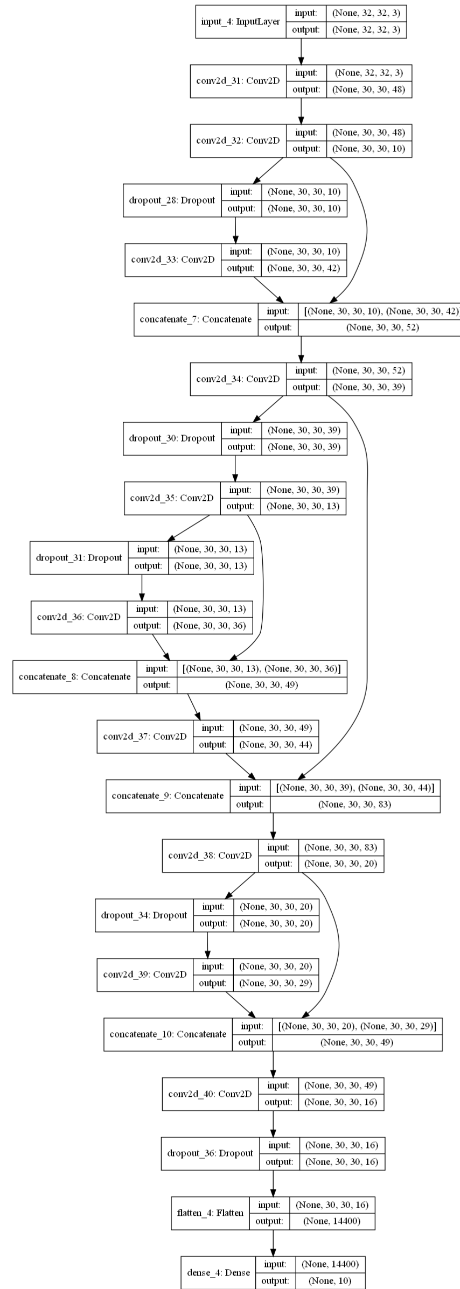
(b) Blueprint graph structuring the connections of 3 instances of module (a), plus additional input and output layers.



(c) Graph assembling the module (a) and blueprint (b) informations into one structural representation.



(d) Final Keras model representation generated from the assembled graph (c).



Source: The Author

3.5 Elitism, crossover and mutations

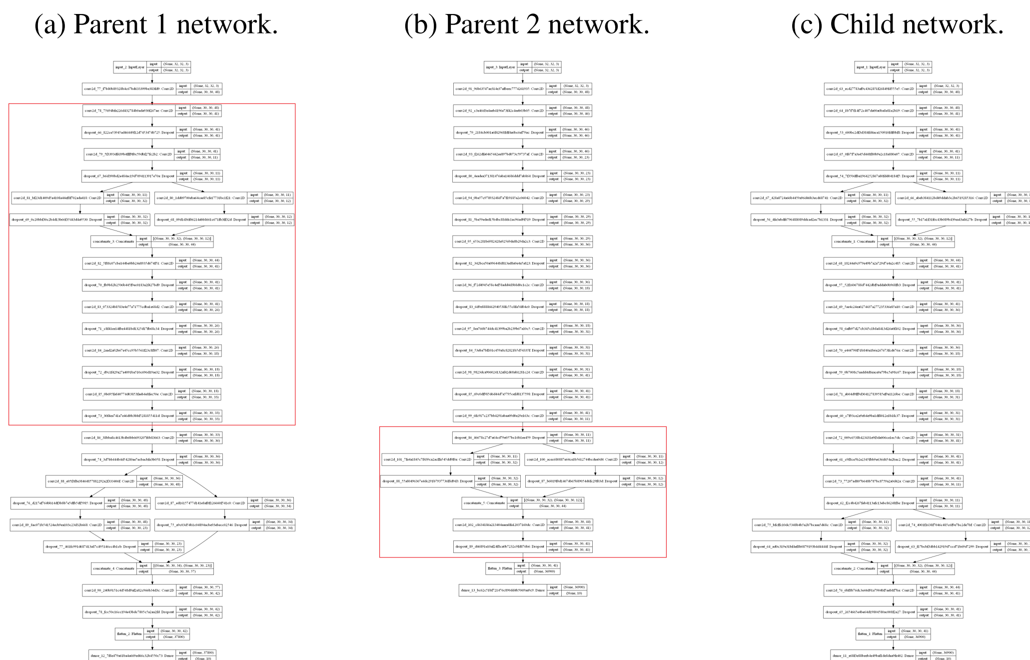
Population management procedures such as elitism, crossover and mutations are handled by the algorithm after the current populations are evaluated by training and

scores. Currently, the proportional amount of subjects to these procedures needs to be defined by the user but could, alternatively, be explored and evolved in the hyperparameter tables.

The implemented elitism mecanism follows the standard definition of the concept, preserving a certain percentage of individuals in populations through generations, ensuring the survival of the best solutions.

Crossover is implemented following a uniform crossover technique (HOLLAND, 1975), switching node contents of two graphs with a fixed chance. The effects are different in blueprints and modules, representing whole layer connection switches in the first case, and simply layer definition switches in the latter. A visual representation of the crossover operation effects can be seen in figure 3.2, where a complete network is generated from two parent networks. The crossover handles cases where layers or modules are not compatible to be switched by only exchanging regions with common origins, as in NEAT.

Figure 3.2: A crossover example. Genetic informations from Parent 1 (a) and Parent 2 (b), highlighted in red, are combined to generate a new network (c). The figures depict the network representation of the three blueprints involved in the crossover process.



(a) Parent 1 network. (b) Parent 2 network. (c) Child network.

Source: The Author

Mutations are implemented similarly to the original proposal of NEAT, representing edge and node alterations such as node or edge removal, creation or reconnection.

The main differences when comparing to what is proposed by CoDeepNEAT are that while NEAT represents the node content as activation functions and edge contents as weights, CoDeepNEAT's representations of node content are much more complex (NN structures!), implying that more complex interactions need to be considered. For this reason, the mutations must also follow the same set of rules as specified for graphs in the population initializations subsection (3.1).

Mutations take place in the graph representations of modules and blueprints as structural changes, as represented in Figures 3.3 and 3.4. Mutation operators are implemented as:

- Node additions, creating nodes and connecting them to other nodes either by inserting them between two nodes that already have an existing mutual connection (Figure 3.3b), or by connecting them to any pair of nodes that support new connections (Figure 3.3c).
- Node removals, creating a new connection between the direct neighbors of the former node (Figure 3.3d).
- Edge removals or additions.
- Node replacement, changing current node content, such as replacing modules in blueprint graphs (as in Figure 3.4) or layers in module graphs.

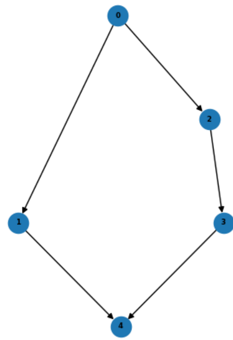
The results from these changes can be seen mostly in the final representations of the models, when the assembling process is finished. In Figure 3.4, a node content change in the original blueprint of a network results in major changes to its structure after the assembling process.

3.6 Chapter summary

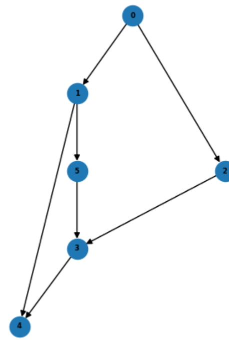
In this chapter were detailed the implementation characteristics of a genetic algorithm for neural network topology and hyperparameter search highly based on CoDeepNEAT (MIIKKULAINEN et al., 2017). The essential steps such as population initializations, parameter definitions, species management, neural network assembling and other aspects like elitism, crossover and mutations are described in detail in the previous sections using the concepts and methods explored in the previous chapter. Next chapter focuses on experimentation performed using the implementation described here.

Figure 3.3: Mutation examples. Figure (a) shows the original graph before any mutations take place. Figure (b) shows the mutation of the original graph by inserting a node between an existing connection. Figure (c) shows the Mutation of the original graph adding a node and preserving existing edges. Figure (d) shows the mutation of the original graph by removing an existing node.

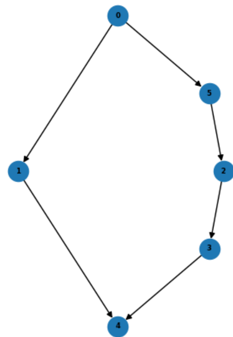
(a) Original graph.



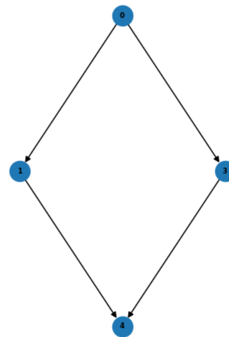
(c) Mutation by node addition.



(b) Mutation by node addition.

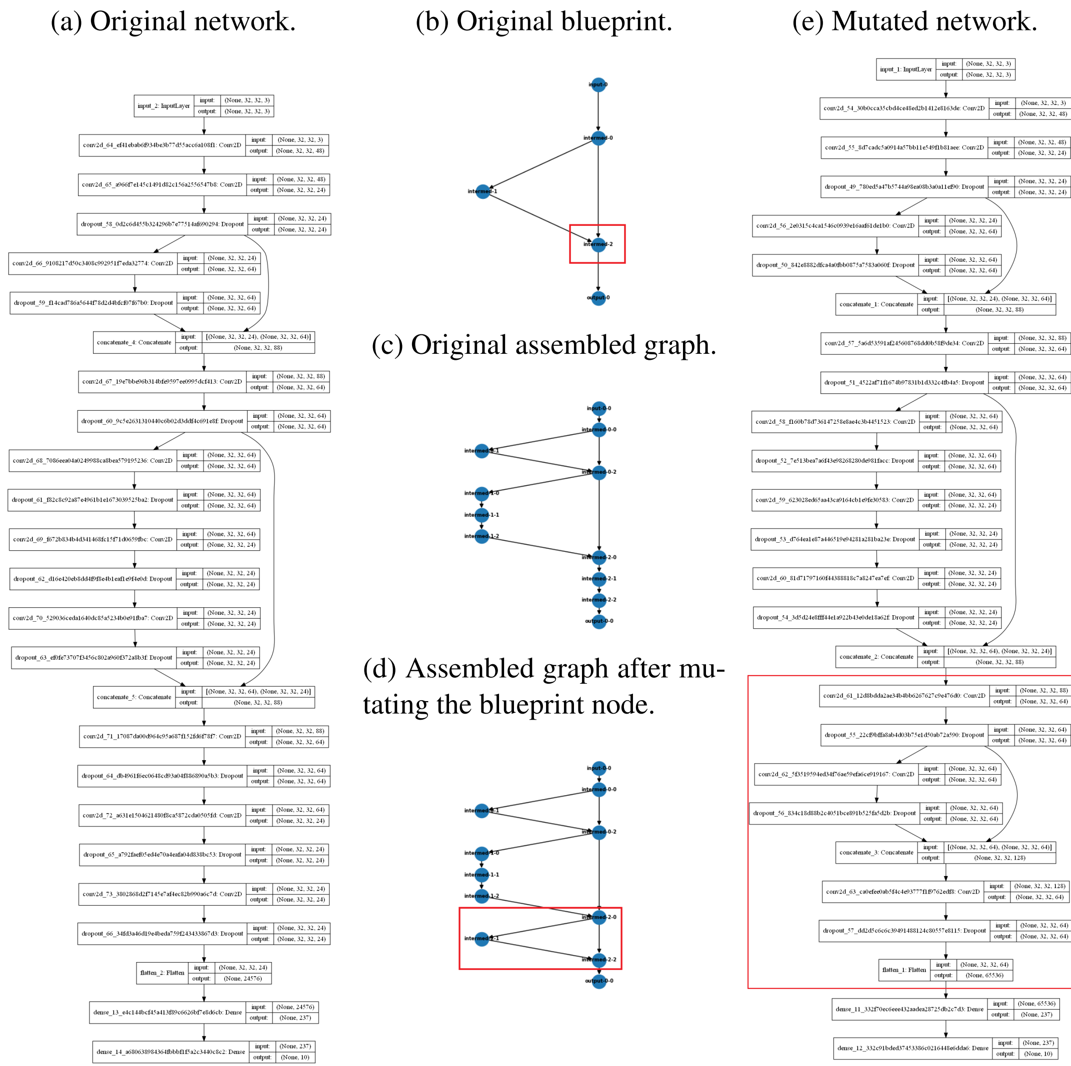


(d) Mutation by node removal.



Source: The Author

Figure 3.4: Node content mutation effects on a network. (a) Original assembled network. (b) The original blueprint used in the network and the indicated node (a module) to be replaced, hightlighted in red. (c) The assembled graph of the used blueprint before mutation switches the indicated module. (d) Assembled graph of the blueprint after mutation switches the indicated module. (d) Resulting network from mutation, with the affected part highlighted in red.



(a) Original network.

(b) Original blueprint.

(c) Original assembled graph.

(d) Assembled graph after mutating the blueprint node.

(e) Mutated network.

Source: The Author

4 EXPERIMENTS AND RESULTS

Experimentation on the algorithm followed consecutive executions in two different image datasets. This chapter describes the datasets, the experiments and results, as well as a discussion on the results and practical usability of the algorithm.

4.1 Datasets

The chosen datasets for experiments using this implementation were MNIST (Lecun et al., 1998) and CIFAR-10 (KRIZHEVSKY; NAIR; HINTON, 2009). Both datasets are simple image datasets containing 10 different classes of images. They are frequently used in benchmarks or experiments using convolutional or fully-connected networks, and have been used before in the task of topology selection in other approaches (MATTIOLI et al., 2019).

The parameter tables used in both experiments can be seen in table 4.1. The amount of modules and layers used for blueprint and module constructions, respectivelly, are specified to be in the range between 1 and 3. The intention is to build minimal structures at first, and progressivelly grow these structures as mutations and crossovers take place as generations pass. Convolutional layers are specified to be used in the intermediate layers, while dense layers are used in the output layers. Including dense layers in the last layer before outputs is a common practice in succesful convolutional networks (SIMONYAN; ZISSERMAN, 2014). Tables 4.2 and 4.3 specify the possible configurations of these two layer types.

Table 4.1: Experiment hyperparameter table.

Parameter	*Type*	*Options*
Module size	Random integer	[1, 3]
Blueprint size	Random integer	[1, 3]
Intermediate component types	Fixed	["Convolutional"]
Output layer component types	Fixed	["Dense"]
Loss functions	Fixed	["categorical_crossentropy"]
Optimizers	Fixed	["Adam"]
Evaluation metrics	Fixed	["Accuracy"]

Source: The Author

Table 4.2: Experiment parameter table for Convolutional layers.

Parameter	*Type*	*Options*
Filters	Random integer	[16, 48]
Kernel size	Random choice	[1, 3, 5]
Stride	Fixed	[1]
Activation function	Fixed	["relu"]
Dropout	Random float	[0, 0.5]

Source: The Author

Table 4.3: Experiment parameter table for Dense layers.

Parameter	*Type*	*Options*
Units	Random integer	[32, 256]
Activation function	Random choice	["relu"]

Source: The Author

4.2 MNIST experiment

Initial experimentation took place using MNIST, a fast-converging and widely used dataset in handwritten digit recognition tasks and overall convolutional network experiments (WITTEN; FRANK; HALL, 2011). MNIST is composed of 60000 28x28 pixel grayscale images of handwritten numerical digits divided in 10 classes (Lecun et al., 1998). The images are simple and placed in neutral backgrounds that simplify predictions.

Figure 4.1: Samples from the MNIST dataset, a handwritten numerical digit dataset.

Source: (Lecun et al., 1998)

Experimentation with MNIST was done using 40 generations, populations of 10

individuals, 10 blueprints and 30 modules, as well as a starting number of species set to 3. For each population, a global set of configurations was used to define elitism, crossover and mutation rates. Elitism rate is set to 20%, preserving the best scoring solutions every generation. The crossover rate is set to 30%, replacing the same proportion of populations' individuals with offspring from good scoring parents. The remaining 50% of the population is subject to mutation operations, generating random changes to existing solutions.

Training using MNIST usually divides the dataset in three parts: a training dataset, composed of 42500 images; a validation dataset, composed of 7500 images; and a test dataset, composed of 10000 images. For this type of experiment, training the network to their full lenght is a hardware and time consuming task. Topology selection methods commonly reduce the sizes of these datasets to smaller proportions to achieve faster results and discard bad solutions in early generations, avoiding the waste of resources in long training procedures, as in Mattioli et al. (2019). For this reason, the training sessions over generations used random samples of 10000 images from the original 60000 divided into 8000 training samples and 2000 validation samples.

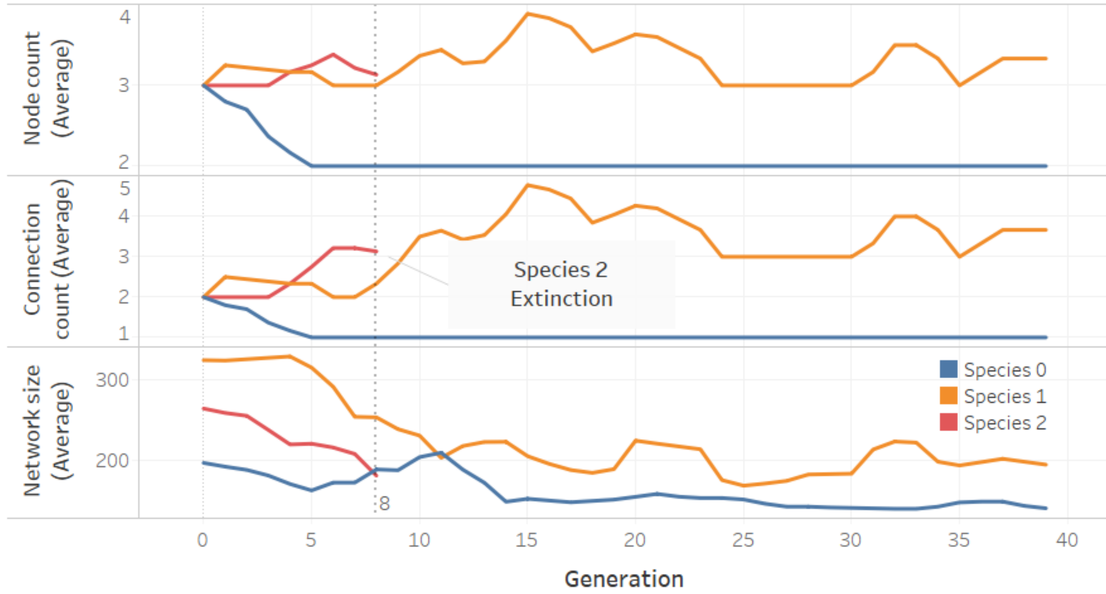
As mutation and crossover operations take place along generations, the features of the networks are expected to change and adapt to reach better accuracy and loss scores during early training. At the same time, elitism ensures these operations don't change actual good results. Figure 4.2 depicts the changes in the counts of nodes, connections and the overall network size of the blueprint population as the generations pass.

In the experiment history shown in Figure 4.2, the networks tend to decrease in size even when increasing the amount of nodes or connections (as in Species 1). This means that the networks are using less filters or neurons in their layers, which is an expected behavior due to MNIST being a very simple dataset that doesn't require complex structures to achieve high loss and accuracy metrics (Lecun et al., 1998). The reduced dataset sizes and training epochs used in topology selection also tends to favour fast-converging networks, as seen in the experiments of (MIIKKULAINEN et al., 2017). Also, Figure 4.2 shows how the evolution of features results in the populations taking certain paths, leading some species (in this case, Species 2) to eventually cease to have representatives, even when not directly interacting with other species through crossovers.

Changes and adaptations to the network features results in changes to the species scores (Figure 4.3), reducing the loss metrics and increasing the accuracy metrics.

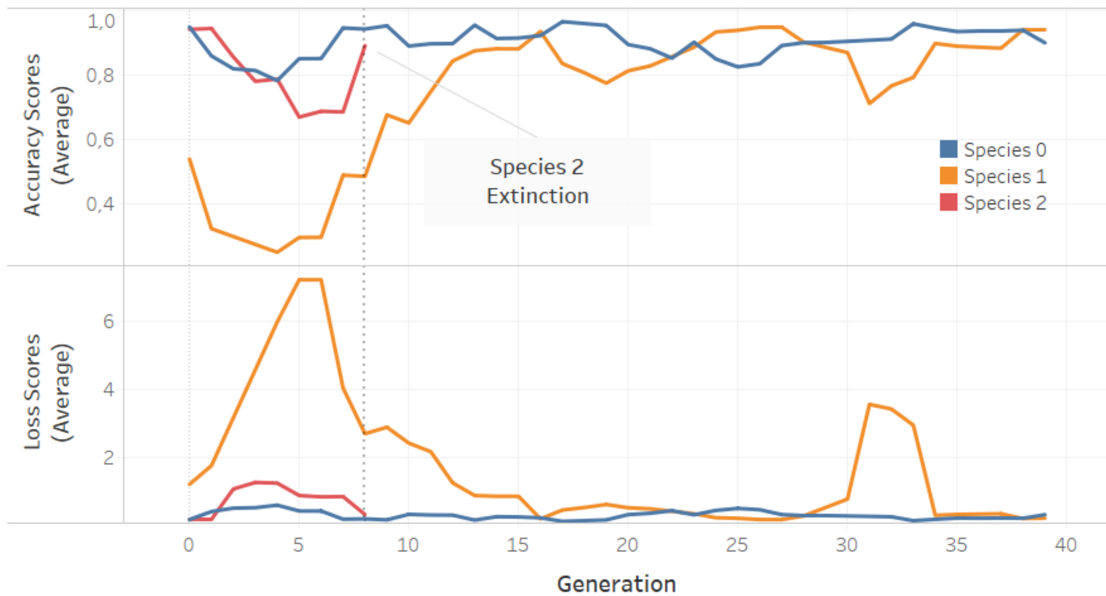
After the 40 generations, a network was selected by the highest accuracy and loss

Figure 4.2: Progress of the features of the three species of networks generated for MNIST over generations. The representation shows the average of each feature for each species. The different line colors represent different species.



Source: The Author

Figure 4.3: Progress of the average accuracy and loss scores of the species over generations for MNIST dataset. The different colors represent different species.

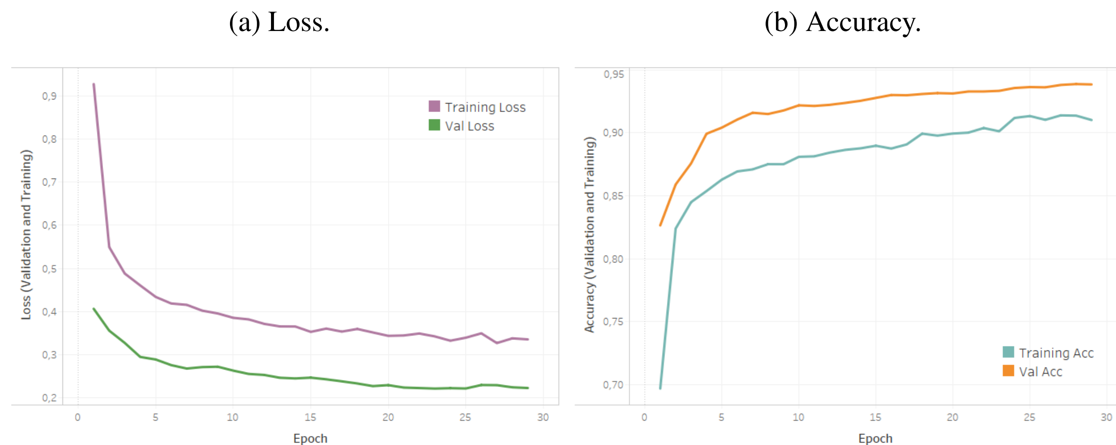


Source: The Author

scores. The chosen network was then trained using the complete MNIST training dataset for 30 epochs to validate whether it would generate acceptable results or not. The training metrics are shown in Figure 4.4 and demonstrate that even in early epochs the resulting model achieves more than 90% validation accuracy. The accuracy using the test dataset achieved a peak 92% accuracy at epoch 30. Of course, the MNIST dataset is supposed to be easy to predict upon and achieve very high accuracy metrics (98.5%[1], for instance). This result shows that the algorithm was able to achieve an acceptable result with few generations, even though the initial network size could be smaller.

Figure 4.4: Training and validation metrics for the best network generated for MNIST after 40 generations.

(a) Loss. (b) Accuracy.



Source: The Author

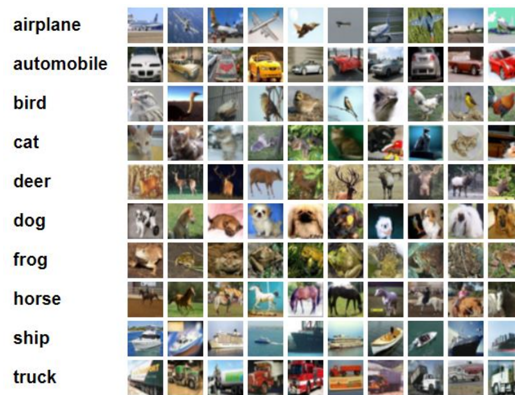
4.3 CIFAR-10 experiment

Experimentation continued using CIFAR-10, a slightly more complex dataset than MNIST. CIFAR-10 is composed of 60000 32x32 colored images of different objects divided in 10 classes (Figure 4.5). Even though CIFAR-10 is similar to MNIST in sample quantity and size, its images represent much more complex and diverse object structures for each class when comparing to MNIST. Training is more exhausting, as well as the required model structure for better results is usually bigger (KRIZHEVSKY; NAIR; HINTON, 2009).

For CoDeepNEAT's original CIFAR-10 experiment, the authors describe an execution of 72 generations using populations of 25 blueprints and 45 modules to gener-

[1] https://towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-54c35b77a38d

Figure 4.5: Samples from the CIFAR-10 dataset, a collection of different classes of images in small scale.



Source: (KRIZHEVSKY; NAIR; HINTON, 2009)

ate 100 individuals (CNNs) per generation. The evaluation of these individuals is done through the test scores of their respective CNNs after 8 training epochs using 50000 images divided into a training set of 42500 samples and a validation set of 7500 samples. Since training convolutional neural networks takes a long time, the reduced training epochs are necessary to achieve approximations of adequate topologies in a viable time. Still, the processing required to train all the individuals every generation for multiple generations is considerable.

After the evolution process of CoDeepNEAT's original CIFAR-10 experiment (MIIKKULAINEN et al., 2017) was complete, the resulting best network was trained on all the 50000 training images for 300 epochs. The classification error obtained was 7.3%, taking 12 epochs to reach 20% test error and around 120 epochs to converge.

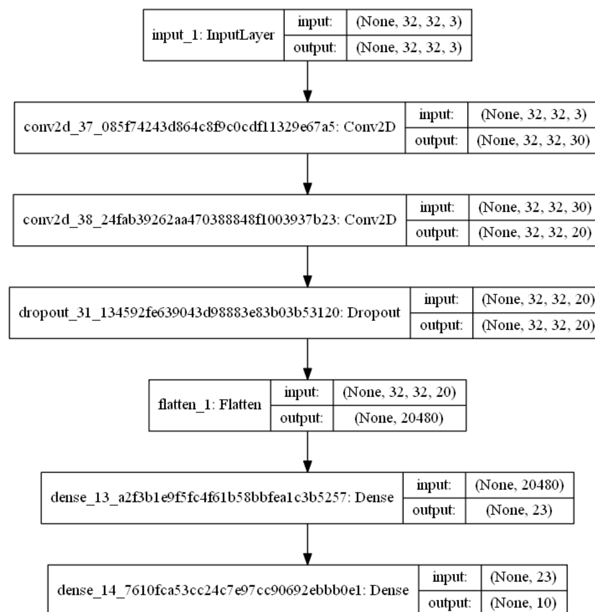
Reproducing such experiment requires a considerable amount of hardware. Training 100 CNNs for 72 generations and 8 training epochs each generation, supposing 30 seconds for each training epoch, would require 1728000 seconds or 480 hours to complete evolution, not considering parallelization efforts. As one of the purposes of this work is to evaluate the usage of CoDeepNEAT in practical use cases for users that might not have access to incredibly potent hardware, the experiments for this work were executed in a smaller scale, similar to the MNIST experiment.

The runs iterated over 40 generations for 6 hours, with populations of 30 modules, 10 blueprints, 10 individuals and starting with 3 species, running in a setup of 4 cores, 30.5GB memory, no GPU included. Following the same steps as in the MNIST experiment, elitism rate is set to 20%, crossover rate is set to 30% and mutation rate is set to

50%. Training epochs are limited to 4 and the original datasets are downsampled to 20000 training images and 2000 validation images for early evaluations.

As expected through the configuration of table 4.1, initial network structures are small and simple, as the network shown in Figure 4.6, created at the first generation of the experiment. As generations pass, the amount of nodes and connections increases or decreases as scores are evaluated. In this specific case, most initial graphs are small structures, thus they naturally increase over generations. This can be visualized in Figure 4.7, where the average count of connections, nodes, and the sizes of blueprints increase over generations.

Figure 4.6: Best scoring network for CIFAR-10 at generation 1. Smaller network topologies are expected to predominate in early generations.

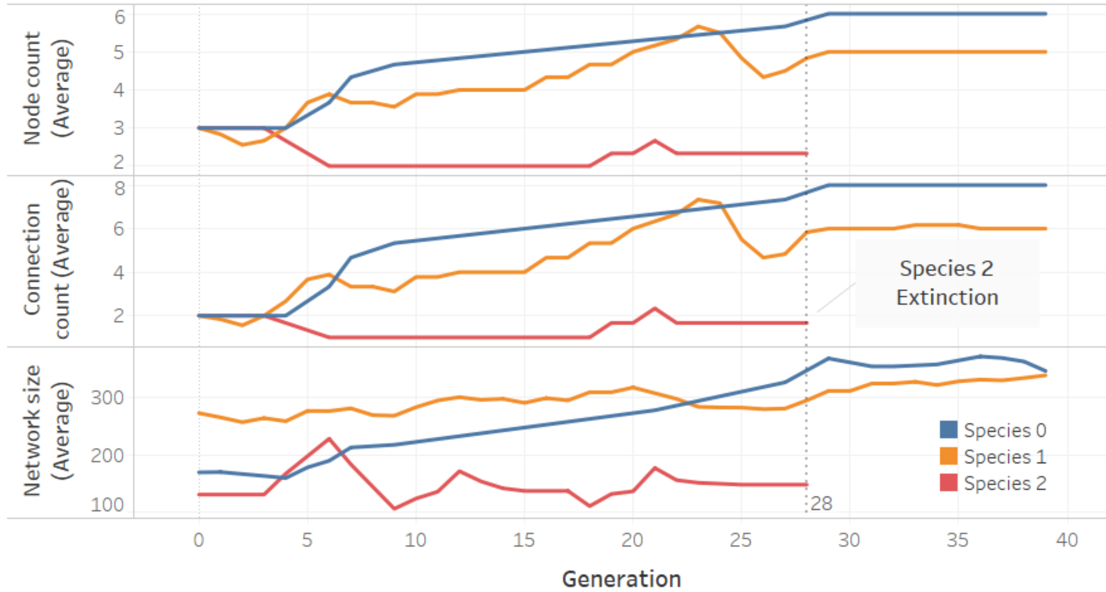


Source: The Author

The increase in network sizes is expected due to CIFAR-10 being slightly more complex then the use case explored with MNIST, requiring larger structures to correctly differentiate the dataset's classes. Figure 4.8 shows the small increase in the accuracy and loss metrics over time as features increase in Figure 4.7. The improvements in the metrics are small due to the few training epochs used, but this early result has the tendency to impact in greater changes in full training sessions using the complete CIFAR-10 dataset.

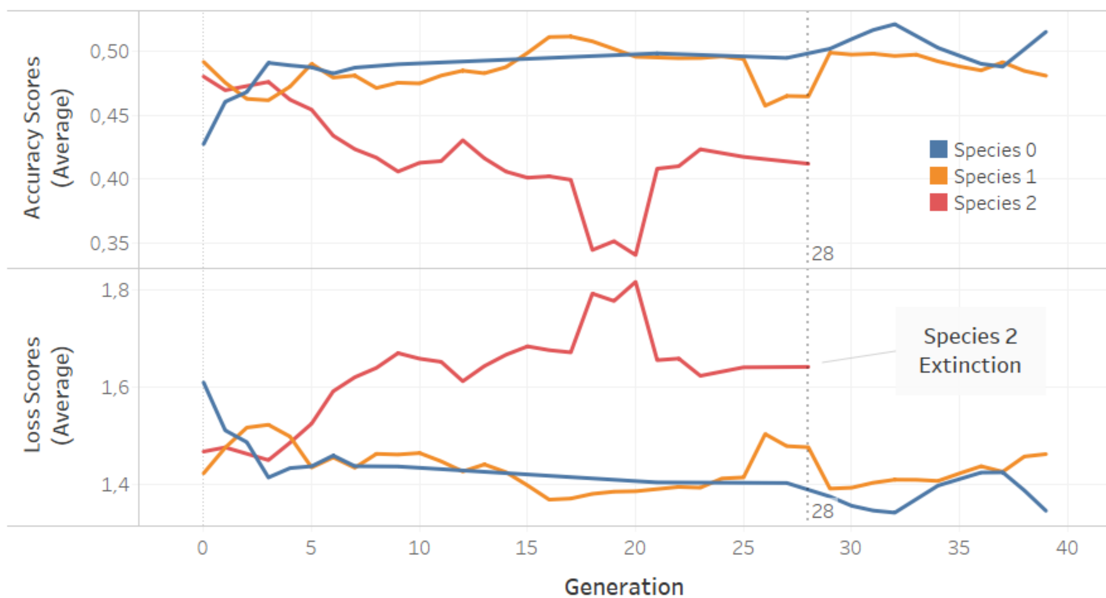
The best resulting network from the experiment after 40 generations was obtained in about 10 hours and can be seen in Figure 4.10. This network was then trained for 130 epochs, but reached a plateau in the validation accuracy metric around the 90th epoch.

Figure 4.7: Progress of the features of the three species of networks generated for CIFAR-10 over generations. The representation shows the average of each feature for each species. The different line colors represent different species.



Source: The Author

Figure 4.8: Progress of the average accuracy and loss scores of the species over generations for CIFAR-10 dataset. The different colors represent different species.



Source: The Author

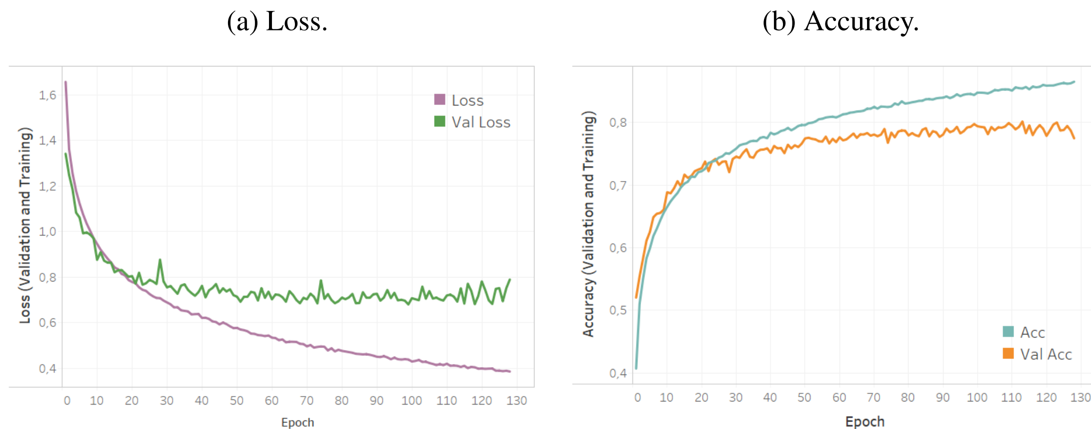
The achieved training accuracy was of 86.5% and 79.5% validation accuracy. Training history for this network can be seen in Figure 4.9, depicting loss and accuracy metrics for

training and validation datasets.

In comparison to the original CIFAR-10 experiment, the network performed slightly worse, presenting a test accuracy of 77% (or test error rate of 23%) as opposed to the 7.3% error presented by (MIIKKULAINEN et al., 2017). The convergence of the validation accuracy happened around training epoch 90, converging faster in comparison to the original CIFAR-10 , where the convergence occured around epoch 120. The faster convergence (and subsequent smaller accuracy) are probably associated with the fact that while (MIIKKULAINEN et al., 2017) executed training sessions during evolution using the full CIFAR-10 dataset, this experiment used downsampled versions to reduce the execution time of each generation. At the same time, this experiment considered only 4 training epochs, while the original used 8 training epochs. Evaluating networks with few training epochs usually favours smaller networks that converge faster, but don't necessarily achieve optimum accuracy.

Figure 4.9: Training and validation metrics for the best network generated for CIFAR-10 after 40 generations. The network achieved 86.5% training accuracy and 79.5% validation accuracy.

(a) Loss. (b) Accuracy.

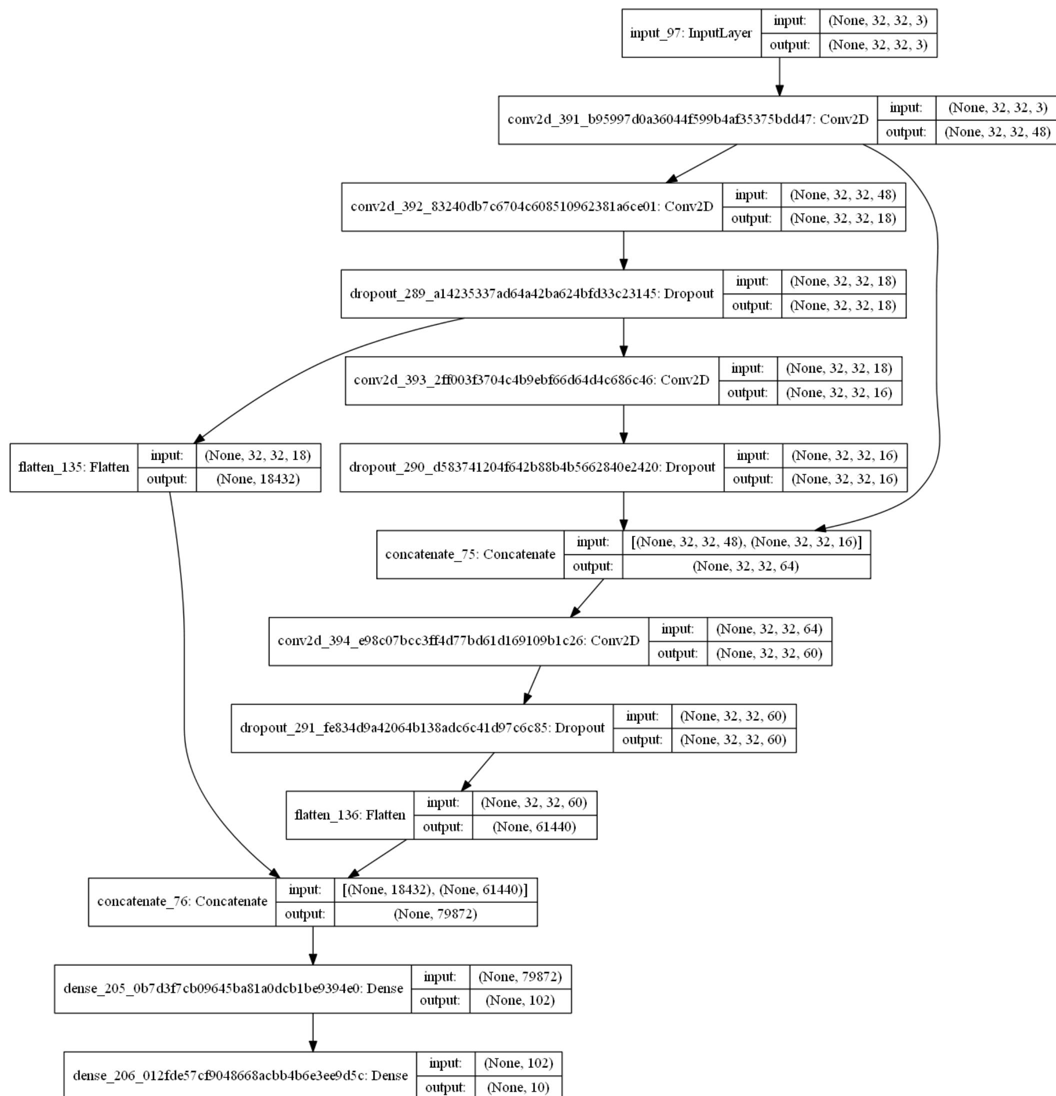


Source: The Author

4.4 Discussion

The results obtained from the experiment show once again that GAs - and specifically CoDeepNEAT - pose as viable solutions to the problems of topology and hyperparameter selection to generate good scoring networks in practical scenarios. Two widely used datasets for machine learning benchmarking and testing, MNIST and CIFAR-10, were used to validate the results and visualize the evolution of solutions over generations.

Figure 4.10: Best scoring network for CIFAR-10 at generation 40.



Source: The Author

The proposed implementation returned adequate solutions even with few generations and small population sizes. Still, as research indicates (VRAJITORU, 1999), bigger populations would probably benefit more from the heuristics used in the genetic algorithm, such as the crossover operator or speciation mechanisms. This is also backed by the original results from (MIIKKULAINEN et al., 2017), which evolve much larger populations of solutions over the course of almost double the generations and finally generate better scoring networks, with the downside of the larger execution time required.

Another point is that studies (RAZALI; GERAGHTY, 2011) indicate that traditional GA operators thrive specially in simple problems where generations can be iterated many times, which is not the case of CoDeepNEAT. The time and hardware requirements

for experimentation with large populations and many generations are relatively demanding even for simple use cases like MNIST or CIFAR-10, and are not explored in detail by (MIIKKULAINEN et al., 2017), making the usage of this type of algorithm very limited to the type of network to be trained and the size of the target dataset.

Deep and complex networks that target large problems, like high-resolution image recognition (DENG et al., 2009) or video recognition (RUSSAKOVSKY et al., 2015), for example, may pose as challenging use cases due to the time required for them to execute training sessions even for few epochs. Traditional efforts to improve the efficiency of GAs (CANTú-PAZ; GOLDBERG, 2000) may generate minor improvements to execution times, but the current algorithm would benefit mostly from approaches that evaluate generated networks using alternative methods rather than exclusively running training sessions for all of them. An example would be methods that generate huge populations but evaluate only certain representatives of each species to elaborate a shared score (Hee-Su Kim; Sung-Bae Cho, 2001), reducing the amount of evaluations performed each generation.

Other ideas that could generate benefits to the algorithm are approaches that narrow the search space to more specific topologies, reducing the need to train so many different networks. One recent example would be (XU et al., 2019), where good results are achieved in reduced GPU time for an object detection use case using the MSCOCO dataset (LIN et al., 2014) by reducing the search space of the exploration algorithm using specialized topological knowledge on the object detection domain.

In scenarios where computing power is not a problem, CoDeepNEAT is proven to achieve good solutions, as demonstrated by (Liang et al., 2019). Then again, the computing power to train thousands of networks by "brute force" is extremelly high and is not commonly accessible for standard users.

5 CONCLUSION

In this work was proposed an open implementation of the CoDeepNEAT algorithm using the popular and highly supported Keras framework. CoDeepNEAT is a powerfull neural network topology generation approach based on neuroevolution of augmenting topologies (NEAT) and co-evolution of modules. It profits from evolutionary techniques and heuristics to explore the immense search space of possible topological configurations for neural networks, employing specialized genetic algorithm aspects to generate and evaluate solutions to the problems of topology and hyperparameter selection. Even though the algorithm is a known approach, no other accessible and public implementations were available to the general academic community as of the conception of this work.

The implementation was detailed on how every aspect was designed to fit together in the final version, based on the original algorithm. It was then tested on popular image datasets and compared to results from the original version considering the differences in environments and experimentation parameters. The results obtained show that acceptable network topologies can be achieve with small population sizes and few generations running in limited hardware environments, even though large runs and big populations generate better results.

With this implementation complete, possible changes can be proposed to improve the base algorithm such as different crossover operations, domain-specialized generation rules for topologies, methods for speciation and classification of individuals and overall population management strategies. The results specially highlight the need to provide better evaluation techniques for the generated neural networks, as it is the most time consuming activity in the algorithm. Another possibility is improving the network generation procedures to explore specialized topologies with previous domain knowledge so the necessary network evaluations are narrowed to smaller search spaces.

The implementation is available at Github [1] with documentation and examples to reproduce the experiments performed for this work.

[1] https://github.com/sbcblab/Keras-CoDeepNEAT

REFERENCES

AALTONEN, T. e. a. Measurement of the top-quark mass with dilepton events selected using neuroevolution at cdf. **Phys. Rev. Lett.**, American Physical Society, v. 102, p. 152001, Apr 2009. Available from Internet: <https://link.aps.org/doi/10.1103/PhysRevLett.102.152001>.

ABADI, M. et al. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Software available from tensorflow.org. Available from Internet: <http://tensorflow.org/>.

AL-SAHAF, H. et al. A survey on evolutionary machine learning. **Journal of the Royal Society of New Zealand**, Taylor Francis, v. 49, n. 2, p. 205–228, 2019.

AMODEI, D. et al. Deep speech 2 : End-to-end speech recognition in english and mandarin. In: BALCAN, M. F.; WEINBERGER, K. Q. (Ed.). **Proceedings of The 33rd International Conference on Machine Learning**. New York, New York, USA: PMLR, 2016. (Proceedings of Machine Learning Research, v. 48), p. 173–182. Available from Internet: <http://proceedings.mlr.press/v48/amodei16.html>.

BäCK, T. **Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms**. New York, NY, USA: Oxford University Press, Inc., 1996. ISBN 0-19-509971-0.

BEASLEY, D.; BULL, D. R.; MARTIN, R. R. **An Overview of Genetic Algorithms : Part 1, Fundamentals**. 1993.

BORTFELDT, A.; GEHRING, H. A hybrid genetic algorithm for the container loading problem. **European Journal of Operational Research**, v. 131, n. 1, p. 143 – 161, 2001. ISSN 0377-2217. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0377221700000552>.

CANTú-PAZ, E.; GOLDBERG, D. E. Efficient parallel genetic algorithms: theory and practice. **Computer Methods in Applied Mechanics and Engineering**, v. 186, n. 2, p. 221 – 238, 2000. ISSN 0045-7825. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0045782599003850>.

CHEN, X. et al. Microsoft COCO captions: Data collection and evaluation server. **CoRR**, abs/1504.00325, 2015. Available from Internet: <http://arxiv.org/abs/1504.00325>.

CHOLLET, F. et al. **Keras**. 2015. <https://keras.io>.

Ciregan, D.; Meier, U.; Schmidhuber, J. Multi-column deep neural networks for image classification. In: **2012 IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2012. p. 3642–3649. ISSN 1063-6919.

Clune, J. et al. Evolving coordinated quadruped gaits with the hyperneat generative encoding. In: **2009 IEEE Congress on Evolutionary Computation**. [S.l.: s.n.], 2009. p. 2764–2771.

CUBUK, E. D. et al. **AutoAugment: Learning Augmentation Policies from Data**. 2018.

DAVIS, L. (Ed.). **Handbook of Genetic Algorithms**. [S.l.]: Van Nostrand Reinhold, 1991.

DENG, J. et al. ImageNet: A Large-Scale Hierarchical Image Database. In: **CVPR09**. [S.l.: s.n.], 2009.

DENG, L.; YU, D. **Deep Learning: Methods and Applications**. [S.l.], 2014. Available from Internet: <https://www.microsoft.com/en-us/research/publication/deep-learning-methods-and-applications/>.

D'SILVA, T. W. **Evolving Robot Arm Controllers Using the NEAT Neuroevolution Method**. Dissertation (Master) — Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, 2006. Available from Internet: <http://nn.cs.utexas.edu/?dsilva:ms06>.

FERNANDO, C. et al. Convolution by evolution: Differentiable pattern producing networks. In: **Proceedings of the Genetic and Evolutionary Computation Conference 2016**. New York, NY, USA: ACM, 2016. (GECCO '16), p. 109–116. ISBN 978-1-4503-4206-3. Available from Internet: <http://doi.acm.org/10.1145/2908812.2908890>.

FLOREANO, D.; DÜRR, P.; MATTIUSSI, C. Neuroevolution: from architectures to learning. **Evolutionary Intelligence**, v. 1, n. 1, p. 47–62, Mar 2008. ISSN 1864-5917. Available from Internet: <https://doi.org/10.1007/s12065-007-0002-4>.

GOLDBERG, Y.; HIRST, G. **Neural Network Methods in Natural Language Processing**. [S.l.]: Morgan & Claypool Publishers, 2017. ISBN 1627052984, 9781627052986.

GOMEZ, F.; SCHMIDHUBER, J.; MIIKKULAINEN, R. Accelerated neural evolution through cooperatively coevolved synapses. **Journal of Machine Learning Research**, p. 937–965, 2008. Available from Internet: <http://nn.cs.utexas.edu/?gomez:jmlr08>.

GOMEZ, F. J.; MIIKKULAINEN, R. Solving non-markovian control tasks with neuroevolution. In: **Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. (IJCAI'99), p. 1356–1361. Available from Internet: <http://dl.acm.org/citation.cfm?id=1624312.1624411>.

GRISCI, B. I.; FELTES, B. C.; DORN, M. Neuroevolution as a tool for microarray gene expression pattern identification in cancer research. **J Biomed Inform**, v. 89, p. 122–133, 01 2019.

HAGBERG, A. A.; SCHULT, D. A.; SWART, P. J. Exploring network structure, dynamics, and function using networkx. In: VAROQUAUX, G.; VAUGHT, T.; MILLMAN, J. (Ed.). **Proceedings of the 7th Python in Science Conference**. Pasadena, CA USA: [s.n.], 2008. p. 11 – 15.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. **The elements of statistical learning: data mining, inference and prediction**. 2. ed. Springer, 2009. Available from Internet: <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>.

HASTINGS, E. J.; STANLEY, K. O. Galactic arms race: An experiment in evolving video game content. **SIGEVOlution**, ACM, New York, NY, USA, v. 4, n. 4, p. 2–10, mar. 2010. ISSN 1931-8499. Available from Internet: <http://doi.acm.org/10.1145/1810136.1810137>.

HAUSKNECHT, M. et al. A neuroevolution approach to general atari game playing. **IEEE Transactions on Computational Intelligence and AI in Games**, 2013. Available from Internet: <http://nn.cs.utexas.edu/?hausknecht:tciaig14>.

HAYKIN, S. S. **Neural networks and learning machines**. Third. Upper Saddle River, NJ: Pearson Education, 2009.

Hee-Su Kim; Sung-Bae Cho. An efficient genetic algorithm with less fitness evaluation by clustering. In: **Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)**. [S.l.: s.n.], 2001. v. 2, p. 887–894 vol. 2. ISSN null.

HOLLAND, J. H. **Adaptation in Natural and Artificial Systems**. Ann Arbor, MI: University of Michigan Press, 1975. Second edition, 1992.

HOLLAND, J. H. **Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence**. Cambridge, MA, USA: MIT Press, 1992. ISBN 0262082136.

JONG, K. A. D. **Evolutionary Computation: A Unified Approach**. Cambridge, MA, USA: MIT Press, 2016. ISBN 0262529602, 9780262529600.

KRIZHEVSKY, A.; NAIR, V.; HINTON, G. Cifar-10 (canadian institute for advanced research). 2009. Available from Internet: <http://www.cs.toronto.edu/~kriz/cifar.html>.

Lecun, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, v. 86, n. 11, p. 2278–2324, Nov 1998. ISSN 1558-2256.

Liang, J. et al. Evolutionary Neural AutoML for Deep Learning. **arXiv e-prints**, p. arXiv:1902.06827, Feb 2019.

LIN, T. et al. Microsoft COCO: common objects in context. **CoRR**, abs/1405.0312, 2014. Available from Internet: <http://arxiv.org/abs/1405.0312>.

LITJENS, G. J. S. et al. A survey on deep learning in medical image analysis. **Medical Image Analysis**, v. 42, p. 60–88, 2017. Available from Internet: <http://dblp.uni-trier.de/db/journals/mia/mia42.html#LitjensKBSCGLGS17>.

LLOYD, S. Least squares quantization in pcm. **IEEE Trans. Inf. Theor.**, IEEE Press, Piscataway, NJ, USA, v. 28, n. 2, p. 129–137, sep. 2006. ISSN 0018-9448. Available from Internet: <https://doi.org/10.1109/TIT.1982.1056489>.

MADHULATHA, T. S. An overview on clustering methods. **CoRR**, abs/1205.1117, 2012. Available from Internet: <http://arxiv.org/abs/1205.1117>.

Mason, S. J. Feedback theory-some properties of signal flow graphs. **Proceedings of the IRE**, v. 41, n. 9, p. 1144–1156, Sep. 1953. ISSN 2162-6634.

MATTIOLI, F. et al. An experiment on the use of genetic algorithms for topology selection in deep learning. **J. Electrical and Computer Engineering**, v. 2019, p. 3217542:1–3217542:12, 2019. Available from Internet: <https://doi.org/10.1155/2019/3217542>.

METAWA, N.; HASSAN, M. K.; ELHOSENY, M. Genetic algorithm based model for optimizing bank lending decisions. **Expert Systems with Applications**, v. 80, p. 75 – 82, 2017. ISSN 0957-4174. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0957417417301677>.

MIIKKULAINEN, R. et al. Evolving deep neural networks. **CoRR**, abs/1703.00548, 2017. Available from Internet: <http://arxiv.org/abs/1703.00548>.

MORIARTY, D. E.; MIIKKULAINEN, R. Forming neural networks through efficient and adaptive coevolution. **Evolutionary Computation**, v. 5, p. 373–399, 1997.

PAPPA, G. L. et al. Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms. **Genetic Programming and Evolvable Machines**, v. 15, n. 1, p. 3–35, Mar 2014. ISSN 1573-7632. Available from Internet: <https://doi.org/10.1007/s10710-013-9186-9>.

PASZKE, A. et al. Automatic differentiation in PyTorch. In: **NeurIPS Autodiff Workshop**. [S.l.: s.n.], 2017.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011.

RAJPURKAR, P. et al. **CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning**. 2017.

RAWAT, W.; WANG, Z. Deep convolutional neural networks for image classification: A comprehensive review. **Neural Comput.**, MIT Press, Cambridge, MA, USA, v. 29, n. 9, p. 2352–2449, sep. 2017. ISSN 0899-7667. Available from Internet: <https://doi.org/10.1162/neco_a_00990>.

RAZALI, N. M.; GERAGHTY, J. Genetic algorithm performance with different selection strategies in solving tsp. In: . [S.l.: s.n.], 2011.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **Nature**, v. 323, n. 6088, p. 533–536, 1986. ISSN 1476-4687. Available from Internet: <https://doi.org/10.1038/323533a0>.

RUSSAKOVSKY, O. et al. ImageNet Large Scale Visual Recognition Challenge. **International Journal of Computer Vision (IJCV)**, v. 115, n. 3, p. 211–252, 2015.

SALIMANS, T. et al. **Evolution Strategies as a Scalable Alternative to Reinforcement Learning**. 2017.

SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. **CoRR**, abs/1409.1556, 2014. Available from Internet: <http://arxiv.org/abs/1409.1556>.

STANLEY, K.; D'AMBROSIO, D.; GAUCI, J. A hypercube-based encoding for evolving large-scale neural networks. **Artificial life**, v. 15, p. 185–212, 02 2009.

STANLEY, K. O. et al. Designing neural networks through neuroevolution. **Nature Machine Intelligence**, v. 1, n. 1, p. 24–35, 2019. ISSN 2522-5839. Available from Internet: <https://doi.org/10.1038/s42256-018-0006-z>.

STANLEY, K. O.; MIIKKULAINEN, R. Evolving neural networks through augmenting topologies. **Evolutionary Computation**, v. 10, p. 99–127, 2001.

SUCH, F. et al. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. 12 2017.

UNGER, R.; MOULT, J. Genetic algorithm for 3d protein folding simulations. In: **Proceedings of the 5th International Conference on Genetic Algorithms**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. p. 581–588. ISBN 1-55860-299-2. Available from Internet: <http://dl.acm.org/citation.cfm?id=645513.657747>.

VERBANCSICS, P.; STANLEY, K. O. Constraining connectivity to encourage modularity in hyperneat. In: **Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation**. New York, NY, USA: ACM, 2011. (GECCO '11), p. 1483–1490. ISBN 978-1-4503-0557-0. Available from Internet: <http://doi.acm.org/10.1145/2001576.2001776>.

VRAJITORU, D. Large population or many generations for genetic algorithms? implications in information retrieval. 12 1999.

Watts, T.; Xue, B.; Zhang, M. Blocky net: A new neuroevolution method. In: **2019 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.: s.n.], 2019. p. 586–593. ISSN null.

WITTEN, I. H.; FRANK, E.; HALL, M. A. **Data Mining: Practical Machine Learning Tools and Techniques**. 3. ed. Amsterdam: Morgan Kaufmann, 2011. (Morgan Kaufmann Series in Data Management Systems). ISBN 978-0-12-374856-0. Available from Internet: <http://www.sciencedirect.com/science/book/9780123748560>.

XU, H. et al. Auto-fpn: Automatic network architecture adaptation for object detection beyond classification. In: **The IEEE International Conference on Computer Vision (ICCV)**. [S.l.: s.n.], 2019.

YANG, J.; HONAVAR, V. Feature subset selection using a genetic algorithm. In: ____. **Feature Extraction, Construction and Selection: A Data Mining Perspective**. Boston, MA: Springer US, 1998. p. 117–136. ISBN 978-1-4615-5725-8. Available from Internet: <https://doi.org/10.1007/978-1-4615-5725-8_8>.

YUAN, X. et al. A genetic algorithm-based, dynamic clustering method towards improved wsn longevity. **Journal of Network and Systems Management**, v. 25, n. 1, p. 21–46, Jan 2017. ISSN 1573-7705. Available from Internet: <https://doi.org/10.1007/s10922-016-9379-7>.

ZOPH, B.; LE, Q. V. Neural architecture search with reinforcement learning. **CoRR**, abs/1611.01578, 2016. Available from Internet: <http://arxiv.org/abs/1611.01578>.