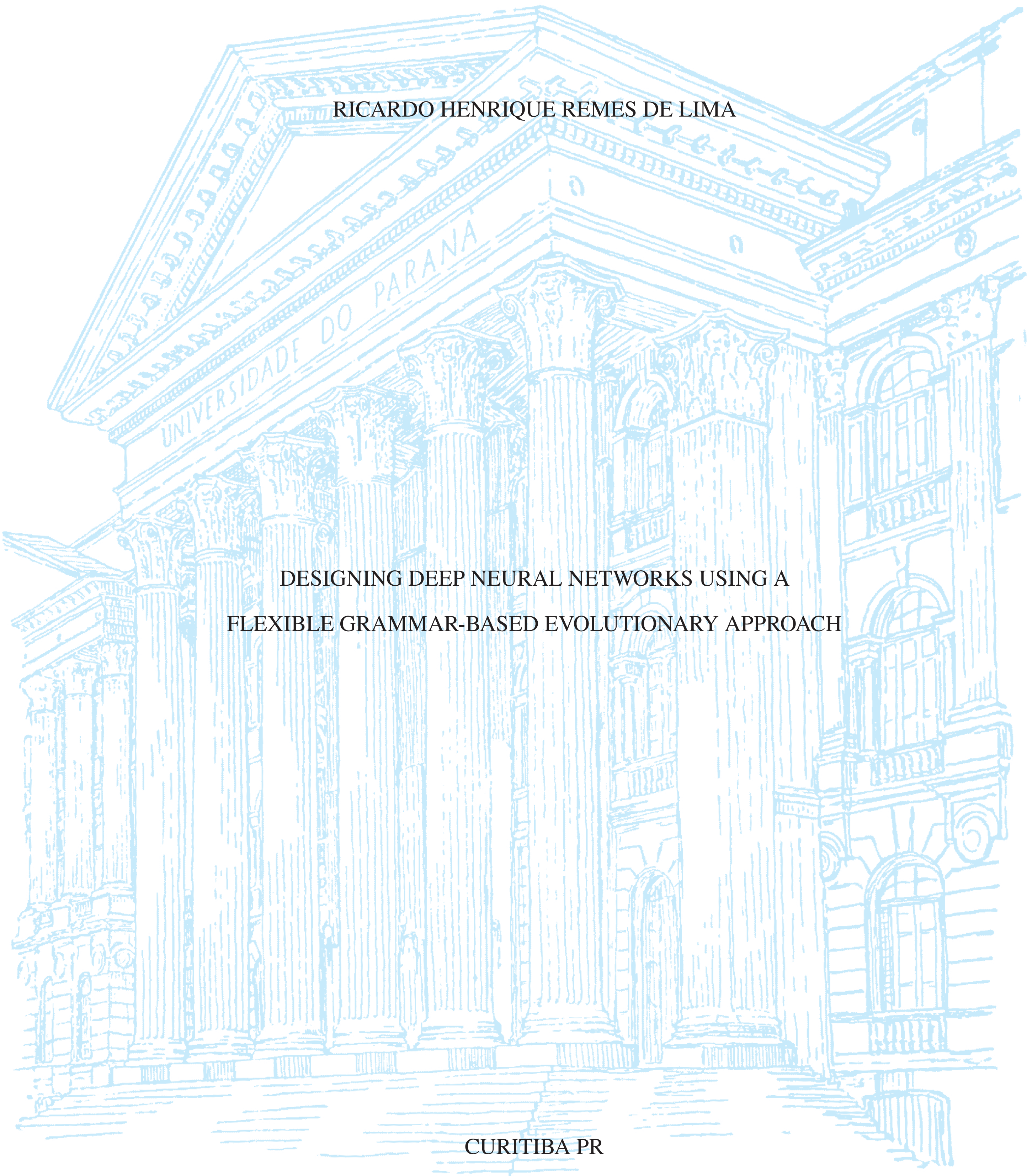
UNIVERSIDADE FEDERAL DO PARANÁ



RICARDO HENRIQUE REMES DE LIMA

DESIGNING DEEP NEURAL NETWORKS USING A FLEXIBLE GRAMMAR-BASED EVOLUTIONARY APPROACH

CURITIBA PR

2018

RICARDO HENRIQUE REMES DE LIMA

DESIGNING DEEP NEURAL NETWORKS USING A FLEXIBLE GRAMMAR-BASED EVOLUTIONARY APPROACH

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Aurora Pozo.

Coorientador: Roberto Santana.

CURITIBA PR

2018

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Lima, Ricardo Henrique Remes de
Designing deep neural networks using a flexible grammar-based evolutionary approach / Ricardo Henrique Remes de Lima. – Curitiba, 2022.
1 recurso on-line : PDF.

Tese (Doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Aurora Trinidad Ramirez Pozo
Coorientador: Roberto Santana

1. Redes neurais (Computação). 2. Sistemas de reconhecimento de padrões. 3. Algoritmos. 4. Programação genética (Computação). 5. Gramática. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Pozo, Aurora Trinidad Ramirez. IV. Santana, Roberto. V. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **RICARDO HENRIQUE REMES DE LIMA** intitulada: **Designing deep neural networks using a flexible grammar-based evolutionary approach**, sob orientação da Profa. Dra. AURORA TRINIDAD RAMIREZ POZO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.
A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 14 de Junho de 2022.

Assinatura Eletrônica
15/06/2022 15:14:13.0
AURORA TRINIDAD RAMIREZ POZO
Presidente da Banca Examinadora

Assinatura Eletrônica
15/06/2022 11:46:53.0
GISELE PAPPA
Avaliador Externo (UNIVERSIDADE FEDERAL DE MINAS GERAIS - UFMG)

Assinatura Eletrônica
15/06/2022 11:49:39.0
EDUARDO JAQUES SPINOSA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
15/06/2022 09:29:30.0
RICHARD ADERBAL GONÇALVES
Avaliador Externo (UNIVERSIDADE ESTADUAL DO CENTRO-OESTE)

Rua Cel. Francisco H. dos Santos, 100 - Centro Politécnico da UFPR - CURITIBA - Paraná - Brasil
CEP 81531-980 - Tel: (41) 3361-3101 - E-mail: ppginf@inf.ufpr.br
Documento assinado eletronicamente de acordo com o disposto na legislação federal Decreto 8539 de 08 de outubro de 2015.
Gerado e autenticado pelo SIGA-UFPR, com a seguinte identificação única: 196916
Para autenticar este documento/assinatura, acesse https://www.prppg.ufpr.br/siga/visitante/autenticacaoassinaturas.jsp e insira o codigo 196916

A minha mãe, a pessoa mais honesta, esforçada, gentil, e amorosa que eu conheci.

ACKNOWLEDGEMENTS

São tantas as pessoas que contribuíram para essa jornada, que de alguma forma, direta ou indiretamente, me ajudaram em momentos difíceis, dando suporte, conselhos, e até me repreendendo quando necessário.

Agradeço primeiramente a Deus, pela minha vida, e por me ajudar a enfrentar todos os obstáculos ao longo dessa caminhada.

A minha família, em especial minha mãe Rosilda, que infelizmente partiu cedo demais, que junto de meu pai Rubens, sempre prezaram pelo meu estudo, dando todo o suporte, mesmo em frente a tantas dificuldades. Meu irmão Rafael e minha irmã Rafaela, que também sempre estiveram ali presentes, cada um do seu jeito, me lembrando do por que eu deveria me esforçar, e por quem eu queria me esforçar. Meus demais familiares, tios, tias, primos, primas, avôs, e avós, que também estiveram ali, torcendo por mim, do início ao fim.

Agradeço a minha namorada Alane, que além de ser uma grande amiga, é minha companheira, que compartilhou e compartilha vários momentos comigo, sejam eles felizes ou tristes. Que me aceitou, conhecendo meus pontos positivos e negativos, e que me ajuda a tentar ser uma pessoa melhor. Sua família também. A Raquel, o Luiz e o Pedro Paulo, que me acolheram e me deram suporte.

Agradeço aos meus amigos, todos eles, desde a o pessoal que eu conheci na escola, na faculdade, no mestrado e agora no doutorado. Há vários que eu talvez já não mantenha tanto contato, mas que também fazem parte da minha formação. O pessoal da graduação, onde a maioria acabou seguindo caminhos diferentes, mas sempre mantendo contato. O pessoal que joga "lolzinho" comigo, e aguenta as reclamações. O pessoal dos laboratórios C-Bio, GrES, e também do Teoria, que me apresentaram pessoas incríveis, com quem passei vários anos, trabalhando, conversando, tomando café, esperando nas filas do RU, e enfrentando os altos e baixos da pós-graduação.

E finalmente, agradeço aos meus professores, tanto da graduação, quanto da pós-graduação. Aqueles que se dedicam a ensinar outras pessoas, preparando aula, procurando melhores exemplos, e tirando as nossas dúvidas. Minha orientadora, Aurora Pozo, que me guiou no mestrado e no doutorado, sendo um grande exemplo sobre como fazer ciência, aceitar críticas, buscar se aperfeiçoar, e saber colher os frutos do trabalho duro.

RESUMO

Na área da Computação Bio-Inspirada, Redes Neurais Artificiais (ANNs) e Algoritmos Evolutivos (EAs) são dois tópicos muito populares, frequentemente aplicados a uma variedade de problemas. Aprendizagem Profunda (DL) tem tido sucesso no processo de automação da engenharia de características, sendo amplamente aplicada para diversas tarefas, como, por exemplo, reconhecimento de fala, classificação, segmentação de imagens, predição de séries temporais, dentre outras. Redes Neurais Profundas (DNNs) incorporam o poder de aprender padrões através dos dados, seguindo um esquema ponta-a-ponta, e expandindo a aplicabilidade em problemas do mundo real, sendo que menos pré-processamento é necessário. EAs podem ser utilizados no projeto de soluções para problemas complexos como, por exemplo, robôs, placas de circuitos, e outros algoritmos, através do uso da Programação Genética (GP). GP utiliza dos mesmos princípios que outros algoritmos evolutivos, evoluindo uma população de soluções, e modificando elas através de operadores de recombinação, buscando pelas melhores soluções, definidas por uma métrica de qualidade. Com o rápido crescimento em ambos escala e complexidade, novos desafios surgiram com relação ao projeto manual e configuração de DNNs. Abordagens evolucionárias têm crescido em popularidade para esse assunto, com mais estudos na área de Neuro-evolução, várias técnicas baseadas em GP tem sido aplicadas de diversas maneiras. Neste trabalho, uma abordagem evolucionária baseada em gramática, chamada GE/DNN, está sendo proposta para o projeto de DNNs. Ela consiste de uma versão modificada da Evolução Gramatical (GE), também buscando inspiração em outros trabalhos relacionados ao projeto de algoritmos, sendo composta por três componentes principais: a gramática, o mapeamento, e o motor de busca. A abordagem é validada em três diferentes aplicações: no projeto de Redes Neurais Convolucionais (CNNs) para a classificação de imagens, U-Nets para segmentação de imagens, e Redes Neurais de Grafos (GNNs) para classificação de textos. Os resultados mostram que a GE/DNN pode gerar diferentes arquiteturas de DNNs de forma eficiente, sendo adaptada para cada problema, e aplicando escolhas que diferem do que é visto usualmente em redes projetadas manualmente. A abordagem tem mostrado grande potencial em relação ao projeto de arquiteturas, atingindo resultados competitivos comparado as demais abordagens comparadas.

Palavras-chave: Algoritmos Evolutivos Programação Genética Projeto Automático Evolução Gramatical Redes Neurais Profundas

ABSTRACT

In the field of Bio-Inspired Computation, Artificial Neural Networks (ANNs) and Evolutionary Algorithms (EAs) are two popular topics, often applied to a variety of problems. Deep Learning (DL) has been very successful in automating the feature engineering process, widely applied for various tasks, such as speech recognition, classification, segmentation of images, time-series forecasting, among others. Deep neural networks (DNNs) incorporate the power to learn patterns through data, following an end-to-end fashion, and expand the applicability in real world problems, since less pre-processing is necessary. EAs can be used to design solutions for complex problems, like robots, circuit boards, and designing other algorithms, through the use of Genetic Programming (GP). GP uses the same principles as other evolutionary algorithms, evolving a population of solutions, and modifying them through the use of recombination operators, searching for the fittest solutions, defined by a quality measure. With the fast growth in both scale and complexity, new challenges have emerged regarding the manual design and configuration of DNNs. Evolutionary approaches have been growing in popularity for this subject, as Neuroevolution is studied more, with many GP-based techniques being applied in different ways. In this work, an evolutionary grammar-based GP approach, called GE/DNN, is being proposed for the design of DNNs. It consists of a modified version of Grammatical Evolution (GE), also drawing inspiration from other works related to the design of algorithms, and it is composed of three main components: the grammar, mapping, and search engine. The approach is validated in three different applications: the design of Convolutional Neural Networks (CNNs) for image classification, U-Nets for image segmentation, and Graph Neural Networks (GNNs) for text classification. The results show that GE/DNN can efficiently generate different DNN architectures, being adapted to each problem, and employing choices that differ from what is usually seen in networks designed by hand. This approach has shown a lot of promise regarding the design of architectures, reaching competitive results with the compared approaches.

Keywords: Evolutionary algorithms Genetic Programming Automatic Design Grammatical Evolution Deep Neural Networks

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ANN	Artificial Neural Network
AutoML	Automated Machine Learning
BNF	Backus Naur Form
BSDS500	Berkeley Segmentation Data Set 500
CNN	Convolutional Neural Network
CGP	Cartesian Genetic Programming
DINF	Departamento de Informática
DL	Deep Learning
DSGE	Dynamic Structured Grammatical Evolution
DNN	Deep Neural Network
EA	Evolutionary Algorithm
EC	Evolutionary Computating
ES	Evolution Strategy
GA	Genetic Algorithm
GE	Grammatical Evolution
GE/DNN	Grammatical Evolution for Deep Neural Networks
GNN	Graph Neural Network
GP	Genetic Programming
ISBI	International Symposium on Biomedical Imaging
LGP	Linear Genetic Programming
LSTM	Long Short-term Memory
ML	Machine Learning
NN	Neural Network
NAS	Neuro Architecture Search
NEAT	Neuro Evolution for Augmenting Topologies
PPGINF	Programa de Pós-Graduação em Informática
RL	Reinforcement Learning
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
SL	Supervised Learning
SGD	Stochastic Gradient Descent
SSTEM	Serial Section Transmission Electron Microscopy
UFPR	Universidade Federal do Paraná

CONTENTS

1 INTRODUCTION

Bio-Inspired Computing is the field of study which seeks to solve computer science problems using models based or inspired in the nature [21]. Bio-Inspired methods focuses on understanding existing patterns, to then apply them in the problem solution, developing new technologies and tuning existing systems. Among the main areas of research for bio-inspired computing, we have:

- Artificial Neural Networks: inspired by the human brain [78];
- Evolutionary Computing: based on the evolution theory [5];
- Swam intelligence: inspired by the social behavior of birds, bees, etc [27];
- Artificial Immunologic Systems: based on the immune system of human beings [16].

The field of Evolutionary Computation (EC) is well known by the constant efforts of researchers into mimicking the mechanisms of biological evolution in order to develop robust algorithms for problems of adaptation and optimization [5]. EC mainly studies population-based trial-and-error problem solvers, called Evolutionary Algorithms (EA), that employ metaheuristics and stochastic strategies. EAs maintain a set of candidate solutions that are generated and iteratively updated. Each new iteration is produced by stochastically removing less desired solutions, and introducing small random changes. This results in a set of solutions that will gradually “evolve”, increasing the quality of its solutions.

A very common example of EA is the Genetic Algorithm (GA) [70], which was inspired by the theory of Darwin on the evolution of species, where only the fittest individuals survive through a number of generations [28, 40]. It also maintains a population of candidate solutions, from where individuals are selected and recombined, using procedures analogous to crossover and mutation from biology, in order to produce new solutions, which can replace the parents if their fitness is better than the the previous ones.

EAs can also used to design solutions of more complex problems, like robots [50], circuit boards [49], and other algorithms [57], through the use of Genetic Programming. Genetic Programming (GP) is a variation of evolutionary algorithms, proposed for the design of computer programs [84]. GP also follows the same principles of other EAs, having a population of computer programs that iteratively evolves, applying recombination operators in order to produce better and improved solutions. Traditional GP uses syntax trees to represent how the programs are encoded. The nodes are used to represent the modules or procedures of a program, and the connections between them are used as inputs and outputs. Grammatical Evolution (GE) [90] is another popular option, among the GP variations, that offers great flexibility to work with the design of computer programs. GE introduces a different way to work with solutions using an indirect encoding, which are encoded as an integer vector of variable-length, and a context-free grammar is used to describe the pieces of the program and the rules that express how these pieces can be combined in order to generate new programs. Other GP approaches include a graph-based variation, called Cartesian GP [74], and also Linear GP [45], to cite the more common.

Artificial Neural Networks (ANNs) is another popular topic in the bio-inspired computing and Machine Learning (ML) fields. ANNs have been developed for many years, trying to use the available computational power to design intelligent systems, following the natural intelligence displayed by humans or animals [89]. ANNs are powerful mathematical models that learn from data, following a similar trial-and-error scheme seen in evolutionary algorithms [78], to be able to make decisions without being explicitly programmed to [49]. ML algorithms are used in a wide variety of applications, such as in medicine [31], email filtering [79], speech recognition [22], and computer vision [18], where it is difficult or unfeasible to develop conventional algorithms to perform the needed tasks [41].

Deep learning (DL) techniques have obtained remarkable achievements on various tasks, such as image recognition, object detection, and language modeling. However, building a high-quality DL system for a specific task highly relies on human expertise, hindering its wide application.

Traditional ML methods require careful engineering and considerable domain expertise to design a feature extractor that could transform raw data into suitable representations learned by the system, to classify or detect patterns in the input [53]. Deep Learning (DL) was proposed to overcome such challenges, by including the steps related to figuring out the best representation for the data, as part of the network. DL models include multiple levels of representation, where each level learns characteristics from the data, starting simple, and building into more complex characteristics, feeding the systems with raw data, and learning the representation from data, automatically. However, building a high-quality DL system for a specific task highly relies on human expertise, hindering its wide application. Deep Neural Network (DNN) architectures are growing fast, in both scale and complexity, presenting new challenges regarding the configuration of these systems. Experts on the subject know how to design and optimize a handful of configuration parameters through experimentation, however, DNNs have complex topologies and hundreds of hyperparameters, which require more time and resources to be tuned [72].

Neural Architecture Search (NAS) can be defined as the process of automating architecture engineering [30, 32, 115]. NAS techniques are mainly categorized into three groups: The Search Space, which defines which architectures can be represented; The Search Strategy, which details how the space is explored; And the Performance estimation strategy, which defines how to measure the quality of architectures. Neuroevolution [34, 72, 87, 93, 97] can be considered as a sub-field on NAS techniques, focusing on evolutionary-based techniques applied to NAS. A major inspiration of neuroevolution comes from the evolution of brains in nature, asking whether these rough abstractions of brains might be evolved artificially through EAs [97].

Recently, Dynamic Structured Grammatical Evolution (DSGE) has been proposed for the design of DNNs [3, 57, 58], as a variation of the traditional Grammatical Evolution (GE) [90]. DSGE was proposed offering more flexibility to the designed networks, and overcoming known issues faced by the traditional approach. As a grammar-based technique, the space of possible designs is defined through a grammar that contains the building blocks for constructing the networks. According to the defined rules, these building blocks describe the many different components of a solution (DNN) that can be combined in many different ways, to construct an architecture offering flexibility and reaching a wider amount of designs with simpler rules.

The automatic design of DNNs is getting more attention [6, 30, 105, 113]. As the performance of top human-designed networks gets refined, the more difficult it becomes to design even better networks. At the same time it creates obstacles, it offers the perfect opportunity to try approaches that deal with the automatic design of algorithms. There are works that successfully applied automatic configuration techniques, which outperformed their hand-designed counterparts [65].

In this work, an evolutionary grammar-based approach for the design of DNNs is proposed. The approach is called GE/DNN, which stands for Grammatical Evolution for Deep Neural Networks. It consists of a modified version of GE, also drawing inspiration from other works related to the design of algorithms, and it is composed of three main components: the grammar, mapping function, and search engine. Firstly, the structure of the grammar is based on the work of [4], expanding it to describe both the layers and hyperparameters of a neural network, also including rules that modify how the architecture grows. Secondly, the improved encoding and mapping processes from DSGE [3] were incorporated along with the GA as the search engine. Furthermore, new genetic operators were proposed to take full advantage of the new encoding and grammar. Thirdly, the evaluation of the solutions was based on the works of [4, 100], where the network first is trained following the traditional weight-optimization scheme, calculating the fitness using a validation set. After the evolution is over, the best designs are then re-trained, and then have their final fitness calculated using the test set.

One of the strengths of GE/DNN is the flexibility it offers, and how easy it is to be adapted to different problems. To demonstrate this, three different applications are presented, including problems of different domains, addressing a different type of network, employing an individual grammar, mapping process, and evaluation strategy, designed specifically for that domain.

In the first application, we deal with the design of CNNs for image classification tasks. Image classification is the process of categorizing and labeling groups of pixels or vectors within an image based on specific rules [96]. A grammar that includes the most common layers used for image tasks is introduced. Furthermore, the grammar also includes the representation of residual blocks [37], a custom layer composed of a well defined group of simpler layers. The use of custom blocks works as a shortcut for designing more elaborated architectures, as it addresses complex structures built with fewer blocks. An empirical study was made to evaluate the approach on the CIFAR-10 [51] dataset, comparing the results with other state-of-the-art evolutionary approaches.

In the second application, we investigate the design of U-Nets for image segmentation tasks. Segmenting images consists of assigning a label to each pixel of an image in order to simplify the information, modifying it into a representation which is easier to analyze [95]. We use a symmetric grammar, which offers an easy and flexible way to generate various U-shaped networks. Two image segmentation datasets are used, the BSDS500 [2] and ISBI12 [15], searching for a network design that best performs in each dataset, and then the results are compared with the original U-Net using 4 image similarity metrics.

The third application covers the design of DNNs for text classification. This study was performed as part of another work, developed in conjunction with a colleague, and apply the proposed approach to problems not related to images, verifying the adaptability of the method. The proposed grammar is based on different Graph Neural Network (GNNs) approaches [11, 36, 47, 48, 106], to design networks that will be used along with a robust representation generator for texts, the Sentence-Bert [86]. Experiments then compared the designed networks, combining these GNN approaches with the individual approaches.

The major contribution of this study is to explore the use of a grammar-based EA in the design of DNNs, and propose a flexible tool that is extendable to a variety of neuroevolution problems. To achieve this, the grammar structure must allow the use of any type of layer, including their hyperparameters. Furthermore, propose ways to manipulate the architecture, doing more than just selecting which pieces will be used, but also how they are going to be used. In addition to that, minor goals can be described as follows, which contribute to achieve the major goal:

- Study of different neuroevolution techniques;
- Explore the variety of neural networks and applications;
- Propose variations of the DSGE approach to design DNNs;
- Design a flexible grammar to handle different networks.

The remainder of this work is structured as follows. In Chapter 2, we present an introduction to subjects, such as Evolutionary Computation, Genetic Programming, and Neural Networks, subjects that are heavily addressed in the proposed approach. In Chapter 3, we present a collection of previous studies on NAS and neuroevolution techniques, covering some different approaches used over time, and focusing on GP-based approaches. Next, Chapter 4 describes our grammar-based approach based on DSGE, providing more details regarding the grammar, mapping, and search process, adapted for the design of DNNs. Moreover, in Chapter 5, we split our experiments into different sections, each covering the different applications we tested using our approach, along with the experiments and evaluation of each scenario. In Section 5.2, we cover our first application, on image classification; Section 5.4, describes the second application, on image segmentation. Section 5.3, presents the third application, on text classification; and Finally, in Chapter 6 we discuss the conclusions and propose future ideas related to this work.

2 BACKGROUND

This chapter covers the main subjects that are relevant for this study, presenting an introduction to each of them. First, a brief explanation about evolutionary computation is given, introducing two population algorithms, evolution strategy, and genetic algorithm. Then, GP is presented, which is a commonly used technique for designing computer programs, inspired by evolutionary algorithms. GP-based variants are also presented, including the tree-based and grammar-based, focusing on a grammar-based variant, the DSGE. Moreover, an introduction to DNNs is given, presenting an overview about the different layers, parameters, and architectures used for most networks. It is also described how these networks are trained, and tested, with different optimizers, and how different metrics are employed for the quality assessment. Furthermore, DL is introduced, including the differences between deep and shallow networks, and also describing two architectures that are commonly used for some DL problems.

2.1 EVOLUTIONARY COMPUTATION

In computer science, researchers put lot of effort into mimicking mechanism of biological evolution in order to develop powerful algorithms for problems of adaptation and optimization, creating what nowadays is called Evolutionary Computation (EC) [5]. In technical terms, EC is a sub-field of artificial intelligence, that studies population-based trial-and-error problem solvers with a metaheuristic or stochastic optimization character.

Typically, an optimization application requires finding a setting $\vec{x} = (x_1, x_2, ..., x_n) \in M$ of free parameters of the system under consideration, such that a certain quality criterion $f : M \longrightarrow I\!R$ (typically called the objective function) is maximized (or, equivalently, minimized).

In Evolutionary Algorithms (EAs), an initial set of candidate solutions is generated and iteratively updated. Each new generation is produced by stochastically removing less desired solutions, and introducing small random changes. In biological terminology, a population of solutions is subjected to natural selection (or artificial selection) and mutation. As a result, the population will gradually evolve to increase in fitness, in this case the chosen fitness function of the algorithm.

EC techniques can produce highly optimized solutions in a wide range of problem settings, making them popular in computer science. Many variants and extensions exist, suited to more specific families of problems and data structures. Most EAs may be divided into generational algorithms, which update the entire sample once per iteration, and steady-state algorithms, which update the sample a few candidate solutions at a time. Common EAs include the Genetic Algorithm (GA) and Evolution Strategies (ES); and there are both generational and steady-state versions of each [70].

2.1.1 Evolution Strategy

The Evolution Strategy (ES) algorithm was developed by Ingo Rechenberg and Hans-Paul Schwefel in 1960 [43]. ES is defined by a simple procedure for selecting individuals, and using only mutation as the operators for introducing small changes. Among the simplest ES algorithms is the $(\mu + \lambda)$ algorithm.

Algorithm 1 presents the basic pseudocode for ES. It starts with a population of μ individuals, generated randomly. Then, the following steps are repeated. First, the fitness for each solution is computed, keeping track of the best individual found so far, and deleting all bu the μ fittest ones. Each of the μ fittest individuals gets to produce λ/μ children through mutation. Then, all children replace the parents. These steps are repeated until the ideal solution is found, or another stop criteria is met, for instance, execution time. Notice that λ should be a multiple of μ. For example, if $\mu = 5$ and $\lambda = 20$, then we have a "(5, 20) Evolution Strategy".

Algorithm 1 The $(\mu + \lambda)$ Evolution Strategy

```
1:  $\mu \leftarrow$ number of parents selected
2:  $\lambda \leftarrow$ number of children generanted by the parents
3:  $P \leftarrow \{\}$
4:  for $\lambda$ times do
5:      $P \leftarrow P \bigcup \{$new random solution$\}$
6:  $Best \leftarrow \Box$
7:  repeat
8:      for each individual $P_i \in P$ do
9:          AssessFitness($P_i$)
10:         if $Best = \Box$ or Fitness($P_i$) > Fitness($Best$) then
11:             $Best \leftarrow P_i$
12:     $Q \leftarrow$ the $\mu$ individuals in $P$ whose fitness are the greatest
13:     $P \leftarrow Q$
14:     for each individual $Q_j \in Q$ do
15:         for $\lambda/\mu$ times do
16:             $P \leftarrow P \bigcup \{$Mutate(Copy($Q_j$))$\}$
17:     $P \leftarrow Q$
18: until $Best$ is the ideal solution or we have run out of time
        return $Best$
```

2.1.2 Genetic Algorithm

The Genetic Algorithm (GA) was invented by John Holland, in 1970, at the University of Michigan [70]. It is a population-based evolutionary strategy, similar to the ES $\mu + \lambda$. It iterates through quality assessments (fitness), selecting and recombining individuals (solutions), from a population of candidates.

The primary difference appears in how selection and recombination take place. While ES selects all parents (μ) and create all the children (λ), the GA selects a few parents and generates a few children, until enough children have been created. Recombination starts with an empty population of children, where a selection operator will pick two parents from the original population. Then, these parents are crossed through a Crossover operator, creating two new children. Later, a Mutation operator is applied, provoking little changes to these children. This process is repeated until the child population is filled, as shown in Algorithm 2. Since the child population has the same number of solutions as the original population, the steps for creating children are repeated for half the size (line 12: $popsize/2$), as each iteration creates two new individuals. The algorithm will repeat the steps of creating, recombining and replacing the solutions in the population for a number of iterations, called generations, and saving the best solution found so far, according to the fitness. The algorithm finally stops when reaching a stopping criteria, for example, by finding the ideal solution, running out of time, or running for number of generations.

Algorithm 2 The Genetic Algorithm

```
1: $popsize \leftarrow$ desired population size
2: $P \leftarrow \{\}$
3: for $popsize$ times do
4:     $P \leftarrow P \bigcup \{$new random solution$\}$
5: $Best \leftarrow \Box$
6: repeat
7:     for each individual $P_i \in P$ do
8:         AssessFitness($P_i$)
9:         if $Best = \Box$ or Fitness($P_i$) > Fitness($Best$) then
10:            $Best \leftarrow P_i$
11:    $Q \leftarrow \{\}$
12:    for $popsize/2$ times do
13:        Parent $P_a \leftarrow$ SelectWithReplacement($P$)
14:        Parent $P_b \leftarrow$ SelectWithReplacement($P$)
15:        Children $C_a, C_b \leftarrow$ Crossover(Copy($P_a$), Copy($P_b$))
16:        $Q \leftarrow Q \bigcup \{$Mutate($C_a$), Mutate($C_b$)$\}$
17:    $P \leftarrow Q$
18: until $Best$ is the ideal solution or we have run out of time
        return $Best$
```

2.1.2.1 Encoding

There is no specific representation for the solutions, though it has classically been operated over fixed-length vectors of boolean values. Representation will often vary according to the problem, or in some cases, according to the algorithm. For example, most common representations are lists of integers and/or floats, and in some cases, trees or any other data structure.

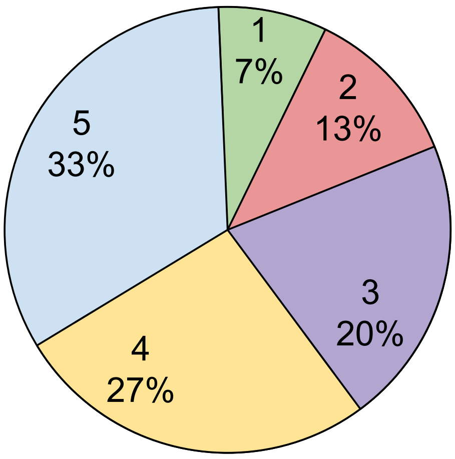
Some of the important aspects that are desirable in the encoding selected for an algorithm or problem, includes the representation power. A binary encoding will have a limited range of values (0 and 1), which is easy to generate new solutions, and recombine using crossover and mutation. However, it might also require a very specific setup, or way to translate a binary string into the actual solution.

In other cases, using solutions encoded as tree will have more representation power, regarding all the characteristics of tree that can be used in your favor. Moreover, the down side of it might be the difficulty of designing recombination operators for it, also generating new solutions, and maybe requiring a repair mechanism for such cases.

2.1.2.2 Selection

The Selection procedure is used to select which solutions will be used for crossover and mutation. While in ES, all μ best solutions are used, the Genetic Algorithm offers more options, and usually not all solutions are used. This way, in GA a solution can be a parent to a undefined number of child solutions, while in ES it is the opposite.

For example, the procedure SelectWithReplacement, mentioned in Algorithm 2, will pick solutions randomly. Solutions are selected proportionally to their fitness values, so the best solutions are selected more often, but also offering chance to low-fitness ones. This procedure is also known as Roulette Selection, with a graphical representation in Figure 2.1.



Individual	Fitness
1	1.0
2	2.0
3	3.0
4	4.0
5	5.0

Figure 2.1: Roulette Selection example.

2.1.2.3 Crossover

Basically, the Crossover operator involves mixing and matching parts of two parents to form a child. This mixing and matching will depend on the representation of the solutions. One classic way of doing crossover in vectors is called One-Point Crossover.

In One-Point crossover, for a vector of length n, a random cut c is selected between 0 and n. Then, all values which index is $< c$ are swapped, producing two children, as shown in Figure 2.2. Some applications might decide to use only one of the two produced child solutions. In such cases, any of the possible combinations can be used.

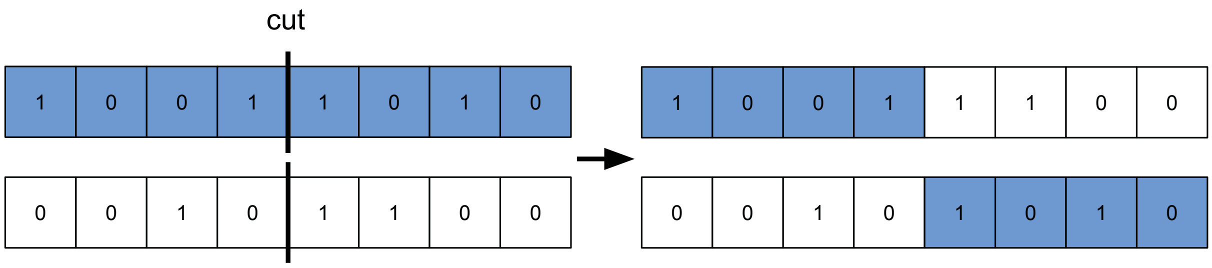


Figure 2.2: One-Point Crossover example.

The One-point crossover can be generalized, generating what is called a N-point crossover. Figure 2.3 shows an example of a two-point crossover. It follows the same principles from the one-point, generating N cuts (2 in this case), and combining one part from each parent, producing two new children solutions.

2.1.2.4 Mutation

The mutation is an operator that modifies a children solution and provides the chance of generating new solutions that would not be possible through traditional Crossover. One simple way to mutate a boolean vector is through the Bit-flip Mutation.

The Bit-flip Mutation iterates through the vector, generating a random value of a certain probability (usually 1 / length of the vector). If the generated value is greater than a predefined threshold, flip the bit in that index. Figure 2.4 shows an example of this procedure.

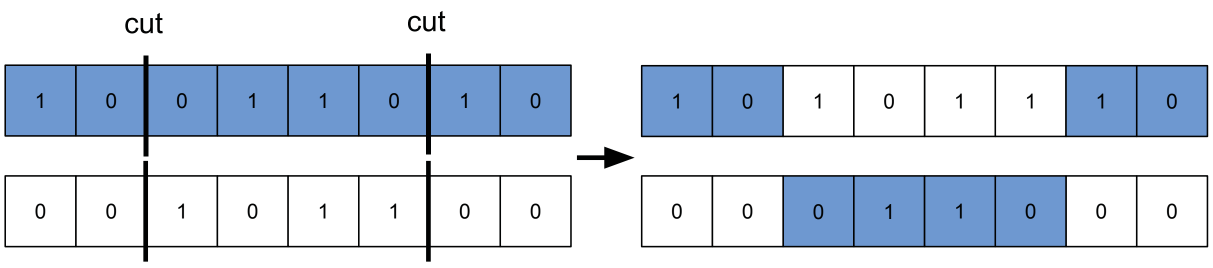


Figure 2.3: Two-Point Crossover example.

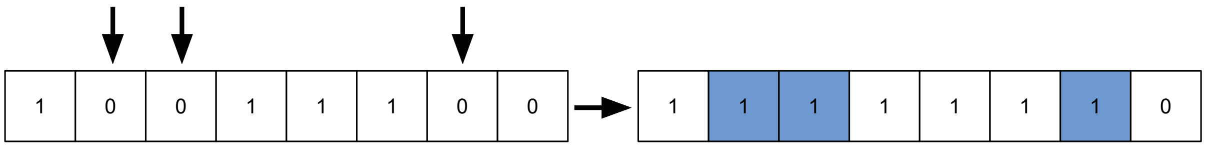


Figure 2.4: Bit-flip Mutation example.

2.2 GENETIC PROGRAMMING

Genetic Programming (GP) is a domain-independent method for solving problems, by designing computer programs, starting from a high-level statement of what needs to be done [84]. It commonly follows the same structure of other evolutionary algorithms, breeding a population of computer programs, and iteratively transforming these programs into a new generation by applying recombination operators. The high-level statement is used to define the requirements of a problem, and attempts of producing computer programs for solving it. A set of well-defined preparatory steps can be used to describe the basic version of GP:

1. The set of terminals (e.g. the independent variables of the problem, zero-argument functions, and random constants) for each branch of the to-be-evolved program,
2. The set of primitive functions for each branch of the to-be-evolved program,
3. The fitness measure (for explicitly or implicitly measuring the quality of individuals in the population),
4. Certain parameters for controlling the run, and
5. The termination criterion and method for designating the result of the run.

The set of terminals and primitives specify the ingredients that are available to create the computer programs. For some problems, the function set may consist of merely the arithmetic functions of addition, subtraction, multiplication and division as well as a conditional branching operator. The terminal set may consist of the external inputs (independent variables) and numerical constants. For many other problems, the ingredients include specialized functions and terminals. For example, if the goal is to automatically program a robot to clean the floor in a room with obstacles, the user must define what the robot is capable of doing, including moving, turning, and activate the mop or vacuum. Regarding the fitness measure, it is necessary to specify how GP evaluates what needs to be done. It can be faced as a mechanism for communicating the high-level requirements of the problem to the GP system. For example, if the goal is to design a robot that collects trash, the fitness function can be defined as the amount of trash collected. The first two preparatory steps define the search space whereas the fitness measure implicitly specifies the s desired goal. The fourth step is related to specifying the control parameters for the run, including population size, probabilities for performing recombination, maximum size of programs, and other details of the run. The fifth preparatory step consists of specifying the termination criterion and the method of designating the result of the run. For the termination criterion, we can define a maximum number of generations to be run, while a single best-so-far strategy can be used to designate the result of the run.

GP has many variations, with different approaches for how the programs are encoded. The traditional GP, which is also known as tree-based GP, the computer programs are expressed as syntax trees rather than as lines of code. For example, the simple expression $max(x * x, x + 3 * y)$ is represented as shown in Figure 2.5. The tree includes nodes and links, where nodes are the instructions to execute, and the links the arguments passed to the instructions. The execution of a tree starts from the leaves, called terminals, passing the arguments to the instructions, called functions, and producing outputs that are passed as arguments to other functions, repeating this process until it reaches the root.

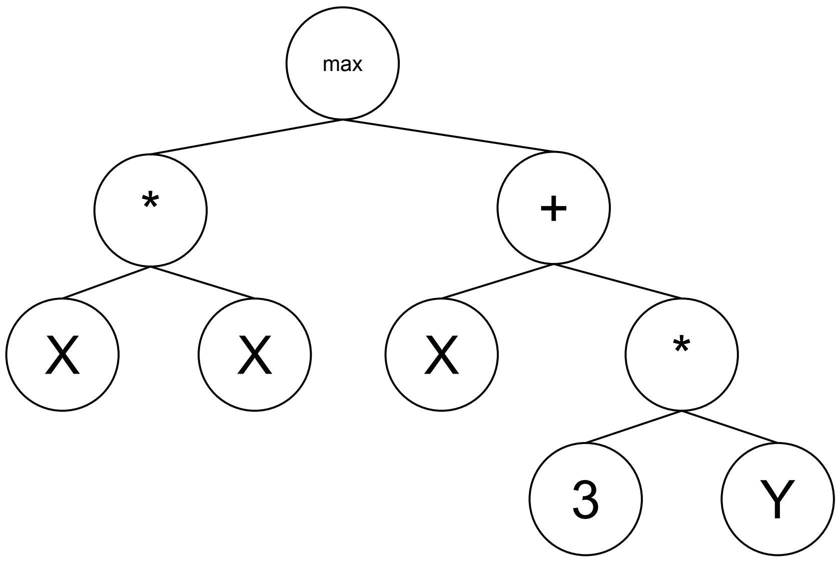


Figure 2.5: Example of a tree-like program.

Some other well-known GP approaches include: Stack-based GP [82], Linear GP (LGP) [45], Cartesian GP (CGP) [74], and Grammatical Evolution (GE) [90]. Each of these approaches have their own characteristics that makes them more suitable in different situations, leaving the decision to the user on which one to use.

2.2.1 Grammatical Evolution

Among the variety of different GP techniques, the grammar-based one is called Grammatical Evolution (GE) [81]. GE introduced a different way to work with solutions using an indirect encoding, where they are encoded as an integer vector of variable-length, and a context-free grammar is used to describe the pieces of the program and the rules that express how these pieces can be combined in order to generate new programs.

GE can be defined by three main components: a) a grammar; b) the genotype-phenotype mapping; and c) the search engine. The grammar is responsible for defining the components used to built the programs, followed by the mapping, which translates one encoded solution into an actual executable program. Moreover, the search engine is the mechanism for searching improved solutions.

2.2.1.1 Grammar

The GE grammar is usually an external file that contains the definition of the components that can be used to build the computer programs, as well as the definition of structures that allows the creation of different variations of that program. These components are usually put in form of rules and productions following the Backus Naur Form (BNF) format.

Formally speaking, a grammar is defined by a tuple $G = (N, T, S, P)$, where N is a non-empty set of non-terminal symbols, T is a non-empty set of terminal symbols, S is an element of N called axiom and used as start rule, and P is a set of productions of the form $A ::= \alpha$, with $A \in N$ and $\alpha \in (N \cup T)^*$, N and T are disjoint [68]. Figure 2.6 shows an example of a simple grammar. The set N of non-terminals contains $\{start, expr, op, term\}$, the set of terminals T contains $\{+, -, *, /, x_1, 0.5\}$, and the axiom S is given by the non-terminal $start$.

$\langle start \rangle$ ::= $\langle expr \rangle\langle op \rangle\langle expr \rangle$ | $\langle expr \rangle$

$\langle expr \rangle$ ::= $\langle term \rangle\langle op \rangle\langle term \rangle$ | ($\langle term \rangle\langle op \rangle\langle term \rangle$)

$\langle op \rangle$ ::= + | - | / | *

$\langle term \rangle$ ::= x_1 | 0.5

Figure 2.6: Example of a grammar for mathematical expressions.

2.2.1.2 Mapping

After the solutions have been created, by random sampling or resulting from the combinations of others solutions, the mapping is responsible for translating a solution into an executable program. The mapping process is basically a grammar expansion, that starting from the axiom, replaces all non-terminals (always picking the left-most non-terminal) by one of their corresponding productions. The way that productions are selected to replace a given nonterminal is by applying the mathematical *mod* operator as:

$$GR = sv\ MOD\ np \tag{2.1}$$

where GR is the chosen grammar rule resulted from the MOD operation between solution value (sv) and the number of possible productions for that rule (np). Figure 2.7 shows a example of a possible expansion using the vector $[22, 7, 55, 22, 3, 4, 30, 16, 203, 24]$, using the Grammar 2.6.

2.2.1.3 Search Engine

As mentioned earlier, the search engine is responsible for searching for new solutions with better fitness values through the application of genetic operators. Traditionally, GE uses a GA as the search mechanism [81], besides some smaller changes, the algorithm is practically the same. The first change is the addition of the mapping process just before the evaluation step, as we first need to generate the executable program from the encoded solutions. Moreover, in addition to the traditional operators of selection, recombination and replacement, GE also introduces the use of two different operators, created to deal with specific situations generated by the use of variable-length solutions, to increase diversity. The first operator is called Prune, which is used

Derivation step	Integers left
<start>	[22, 7, 55, 22, 3, 4, 30, 16, 203, 24]
<expr><op><expr>	[7, 55, 22, 3, 4, 30, 16, 203, 24]
(<term><op><term>)<op><expr>	[55, 22, 3, 4, 30, 16, 203, 24]
(0.5 <op><term>)<op><expr>	[22, 3, 4, 30, 16, 203, 24]
(0.5 / <term>)<op><expr>	[3, 4, 30, 16, 203, 24]
(0.5 / 0.5)<op><expr>	[4, 30, 16, 203, 24]
(0.5 / 0.5) + <expr>	[30, 16, 203, 24]
(0.5 / 0.5) + <term><op><term>	[16, 203, 24]
(0.5 / 0.5) + x1 <op><term>	[203, 24]
(0.5 / 0.5) + x1 * <term>	[24]
(0.5 / 0.5) + x1 * x1	[ ]

Figure 2.7: Example of the mapping steps for a random solution.

to cut solutions in a given random point, and the second operator is called Duplicate, used to copy a portion of a solution and pasting it at the end [90].

2.2.2 Dynamic Structured Grammatical Evolution

GE is considered one of the most relevant variants of GP approaches [69]. However, it has two major issues related to locality and redundancy. The first regards the situation where small modifications to the genotype can lead to big modifications in the phenotype. The latter leads to situations where different genotypes can map to the same phenotype. Both situations can affect the evolutionary process negatively, by wasting time and resources with solutions of poor quality, resembling random search.

Dynamic Structured Grammatical Evolution (DSGE) [4, 67] is an improved version of GE, that was proposed to address the known problems with low locality and redundancy present in the original approach. Moreover, it proposes a different encoding for solutions, which affects how the grammar and mapping function work, being more efficient during the search process.

2.2.2.1 Encoding

The solutions in DSGE are encoded as a list of lists, rather than a single list from classical GE, where each internal list is directly connected to one non-terminal symbol from the grammar (see Figure 2.8). The length of a solution is defined by the number of non-terminals in the grammar. Moreover, the size of each internal list will depend on the number of values needed to perform a complete map, and the values will depend on the number of options for each rule.

For example, in Figure 2.8, we have a solution built based on Grammar 2.6. The first sub-list points to the non-terminal <start>, and it can include values ranging from 0 to 1 (two options). The second sub-list points to the rule <expr>, also ranging from 0 to 1. Moreover, the third sub-list points to rule <op>, having four possible values, from 0 to 3. Finally, the last sub-list points to rule <term>, also with two options.

2.2.2.2 Initialization

Algorithm 3 presents the procedure that is used to create new random solutions using a grammar. It takes five parameters: the grammar, a maximum depth, the genotype (initially empty), a non-terminal symbol (initially the axiom symbol), and the current depth (initialized as 0). The

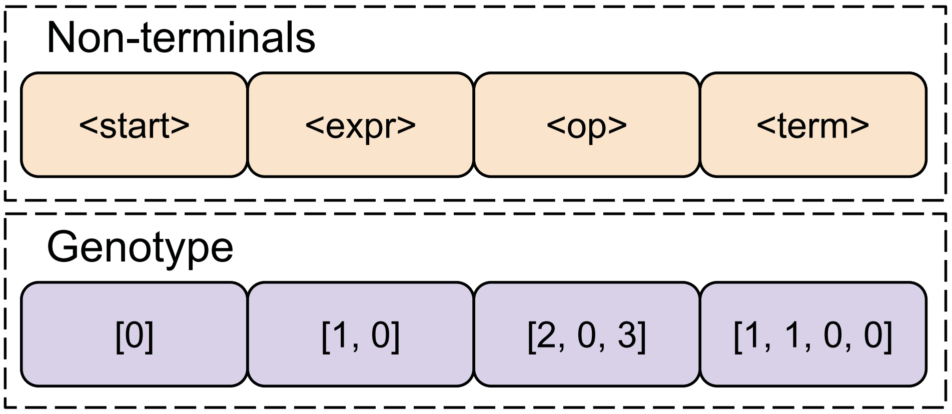


Figure 2.8: Sample solution using the DSGE encoding.

algorithm takes the current symbol and selects one of the possible expansion (production) for it, in this initialization context, it means the integer value that would represent that expansion (line 2). Then, it checks whether this symbol is recursive, and if current procedure state is respecting the max depth, otherwise it selects a new expansion with no recursive outcome (lines 3-7). Following, the expansion is added to the genotype in the sub-list that represents the current symbol (lines 8-11). Then, it takes the symbols produced by this expansion and calls this procedure recursively for each non-terminal symbol (lines 12-15).

Algorithm 3 DSGE Recursive solution generation

```
 1: procedure CREATE_SOLUTION(grammar, max_depth, genotype, symbol, depth)
 2:     expansion = randint(0, len(grammar[symbol]-1)
 3:     if is_recursive(symbol) then
 4:         if expansion in grammar.recursive(symbol) then
 5:             if depth ≥ max_depth then
 6:                 non_rec = grammar.non_recursive(symbol)
 7:                 expansion = choice(non_rec)
 8:     if symbol in genotype then
 9:         genotype[symbol].append(expansion)
10:     else
11:         genotype[symbol] = [expansion]
12:     expansion_symbols = grammar[symbol][expansion]
13:     for symb in expansion_symbols do
14:         if not is_terminal(symb) then
15:             create_solution(grammar, max_depth, genotype, symb, depth+1)
```

2.2.2.3 Mapping

The mapping process for DSGE is very similar to the initialization, however, as presented in Algorithm 4, instead of randomly selecting the expansions according to the symbol, it uses the value presented in the encoded solution as the expansion choice.

In other words, the mapping performs a grammar expansion, starting from the axiom, always replacing the leftmost non-terminal by one of its productions, which is analogous to a depth-first search. For example, if the left-most terminal is <expr>, the values will be extracted

Algorithm 4 DSGE Recursive solution mapping

```
1: procedure MAPPING(genotype, grammar, max_depth, read_integers, symbol, depth)
2:     phenotype = “ ”
3:     if symbol not in read_integers then
4:         read_integers[symbol] = 0
5:     if symbol not in genotype then
6:         genotype[symbol] = []
7:     if read_integers[symbol] $\geq$ len(genotype[symbol)] then
8:         if depth $\geq$ max_depth then
9:             generate_terminal_expansion(genotype, symbol)
10:        else
11:            generate_expansion(genotype, symbol)
12:    gen_int = genotype[symbol][read_integer[symbol]
13:    expansion = grammar[symbol][gen_int]
14:    read_integer[symbol] += 1
15:    for symb in expansion do
16:        if is_terminal(symb) then
17:            phenotype += symb
18:        else
19:            phenotype += mapping(genotype, grammar, max_depth, read_integers, symb, depth+1)
    return phenotype
```

from the list $\{1, 0\}$ as in Figure 2.8. Moreover, the values for each non-terminal are sampled respecting the number of options, which removes the need for using the mod operator as in classical GE.

Derivation step	Integers left
<start>	[[0], [1, 0], [2, 0, 3], [1, 1, 0, 0]]
<expr><op><expr>	[[], [1, 0], [2, 0, 3], [1, 1, 0, 0]]
(<term><op><term>)<op><expr>	[[], [0], [2, 0, 3], [1, 1, 0, 0]]
(0.5 <op><expr>)<op><expr>	[[], [0], [2, 0, 3], [1, 0, 0]]
(0.5 / <term>)<op><expr>	[[], [0], [0, 3], [1, 0, 0]]
(0.5 / 0.5)<op><expr>	[[], [0], [0, 3], [0, 0]]
(0.5 / 0.5) + <expr>	[[], [0], [3], [0, 0]]
(0.5 / 0.5) + <expr><op><term>	[[], [], [3], [0, 0]]
(0.5 / 0.5) + x1 <op><term>	[[], [], [3], [0]]
(0.5 / 0.5) + x1 * <term>	[[], [], [], [0]]
(0.5 / 0.5) + x1 * x1	[[], [], [], []]

Figure 2.9: Mapping process using the DSGE encoding.

Considering the solution present in Figure 2.8, the mapping would happen as in Figure 2.9. Starting from the axiom <start>, the first value from the first list is consumed, which is related to the non-terminal <start>. The value is 0, which means that the non-terminal will be replaced by the production (considering we are using Grammar 2.6) <expr><op><exp>. To follow, the next non-terminal is <expr>, and the next value related to this non-terminal is 1, which will replace <expr> by the production (<term><op><term>). The mapping procedure repeats these steps until all the values are used, and there are no non-terminals left.

During the evolution, there might be cases when a solution resulting from a recombination, might have either extra or not enough values to perform a full mapping. In such cases, the mapping function is also responsible for adding new random value when needed, and just ignore unused values, removing them at the end.

2.3 NEURAL NETWORKS

Machine learning (ML) is the study of computer algorithms that can improve automatically through experience, and by the use of data [78]. The main tool in ML are the Neural Networks (NNs), which are pattern recognition systems that require careful engineering and considerable domain expertise to design a feature extractor that transforms raw data, such as the pixel values of an image, into a suitable internal representation or feature vector from which the learning subsystem, often a classifier, could detect or classify patterns in the input.

Basically, neural networks are built from simple units, usually called neurons by analogy. These are interlinked by a set of weighted connections. Learning is accomplished by modifying these connection weights. Each unit corresponds to a feature of characteristic we want to analyze, and are organized in layers. Networks have multiple layers, where we have an input layer, the output layer, and a set of what are called hidden layers (see Figure 2.10). The information to be analyzed in fed to the neurons in the input layer, and then propagated to the neurons of the second layer for further processing. The result of this processing is then propagated to the third layer and so on until the output layer, which defines what is known as a feed-forward network. The goal of the network is to learn, or to discover, some association between input the output patterns. This learning is achieved through the modification of the connection weights between units. In statistical terms, this is equivalent to interpreting the value of the connections between units as parameters [12].

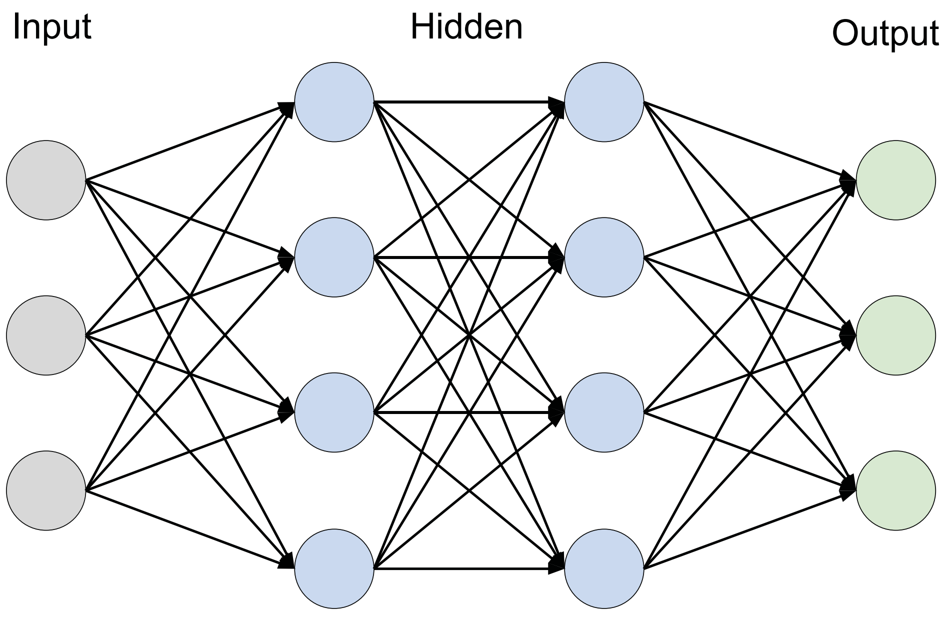


Figure 2.10: Example of a feed-forward network.

Furthermore, NNs can also be categorized according to the number of hidden layers it contains. Shallow networks will have a small number of hidden layers, while deep networks will be the opposite. There are other characteristics that make deep networks more useful for many situations, in special, the fact that deep networks require less pre-processing of the data.

2.3.1 Layers

The neurons in a NN are also known as layers, where they act as the basic building blocks for constructing an architecture. These building blocks consist of basically a set of weights and an activation function. In general, a layer works by feeding data to the inputs, which are then multiplied by the weights and then added together. Then, it applies the activation function, which is usually a non-linear transformation, generating the outputs.

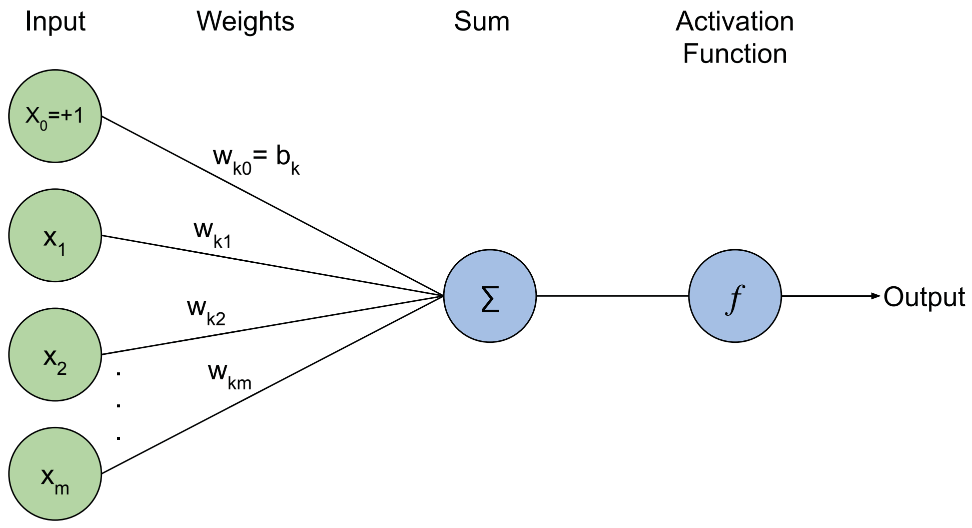


Figure 2.11: Representation of a artificial neuron used in neural networks.

Figure 2.11 shows an example of a basic neuron. For a given artificial neuron k, let there be $m + 1$ inputs with signals x_0 through x_m and weights w_{k0} through w_{km}. Usually, the x_0 input is assigned the value +1, which makes it a bias input with $w_{k0} = b_k$. This leaves only m actual inputs to the neuron: from x_1 to x_m. The output of the k_{th} neuron is:

$$v_k = f(\sum_{j=0}^{m} w_{kj}x_j) \tag{2.2}$$

There are a great variety of layers, for instance, convolutional, pooling, dropouts, and dense layers, are some of the most common layers that are applied to a variety of architectures, and for many different tasks. Each of these performs specific transformations to the inputs in addition to the basic behavior, bringing more options when designing the networks.

2.3.2 Training and Testing

The most common form of ML, deep or not, is Supervised Learning (SL). In SL, the objective is to learn weights of the network that maps an input to an output based on example input-output pairs [53]. For example, in convolutional layers, we want to learn the filters, which will extract the characteristics from the input. In dense layers, we learn the weights that multiply the inputs, which will give more importance to some features over others. In any case, the weights of each layers are fine tuned in order to produce an output that is as close to the correct answer. The learning process is done on a training set, from which we know the input and expected output.

Then, there is also the test set, which is a set of unseen data, used to evaluate the performance of the network at the end of the training. Additionally, a validation set can be used to check the learning process during the training, being applied in regular intervals.

An optimizer is the responsible for adjusting the weights. The network will process each input and compare the produced output by the network, with the expected output, adjusting the weights of the layers through backpropagation, guiding the networks towards producing outputs that are closer to the analyzed data. By repeating this process, the objective is to produce a network which weights are adjusted to the point where it is able to process unseen data, and produce the correct output, or something as close to it as possible. There are many different optimizers that can be used, employing different strategies. For instance, common examples of optimizers include the Stochastic Gradient Descent (SGD) [13], Adam [46], and RMSprop [52]. Moreover, the optimization also requires the definition of a set of parameters, like batch size, number of epochs, momentum, and most important, learning rate, which determines how the weight adjustments are calculated, heavily impacting the quality of the learning process of a network. Furthermore, the "quality" of a network is computed using the test set, where the network is executed over a portion of unseen data, and evaluating how many samples were processed correctly, compared to the expected output.

Depending on the problem, different metrics can be used to measure the quality of a network. For instance, for classification problems, with balanced datasets, using the accuracy might be enough. Then, for segmentation problems, image similarity metrics, such as DICE or Jaccard, might be more suitable. For other problems, we have root mean squared error (RMSE), precision, recall, and others.

2.3.3 Deep Neural Networks

Traditional NNs were limited in their ability to process natural data in their raw form. Usually, pre-processing techniques were applied in order to generate a set of relevant features, that describe the data being fed into the networks, attaching the quality of a network to the quality of this feature set. Deep Neural Networks (DNNs) were proposed with the objective of making the architecture also learn what would be the best representation for the data, through the layers. Deep-learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple non-linear modules, where each transforms the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level.

For instance, convolutional neural networks (CNNs) [54] and recurrent neural networks (RNNs, in particular long short-term memory, or LSTM [39]), which have existed since the 1990s, have improved State-of-the-art significantly in computer vision, speech, language processing, and many other areas [19, 35, 104].

2.3.4 Convolutional Neural Network

Convolutional Neural Networks, are much easier to train and test to achieve good performances when compared to standard feed-forward neural networks [51]. To design a CNN architecture, it is required to define the number layers, as well as their types. Usually, convolutional, pooling, dropout, and dense layers are used. Moreover, each type of layer has different parameters, for instance, convolutional layers, have the number of feature maps, kernel size, stride size, among others, while pooling layers have filter size, stride size, number of neurons.

The architecture of a CNN is basically divided into two parts (see Figure 2.12). The first is responsible for the feature extraction, starting from the inputs, it usually uses convolutional and pooling layers, to learn features from the data. These features start as simple shapes ans forms in early layers, becoming more complex as the architecture goes deeper. The second part is responsible for the classification, where dense and dropout layers are used to classify the data, using the features learned previously, to make decisions.

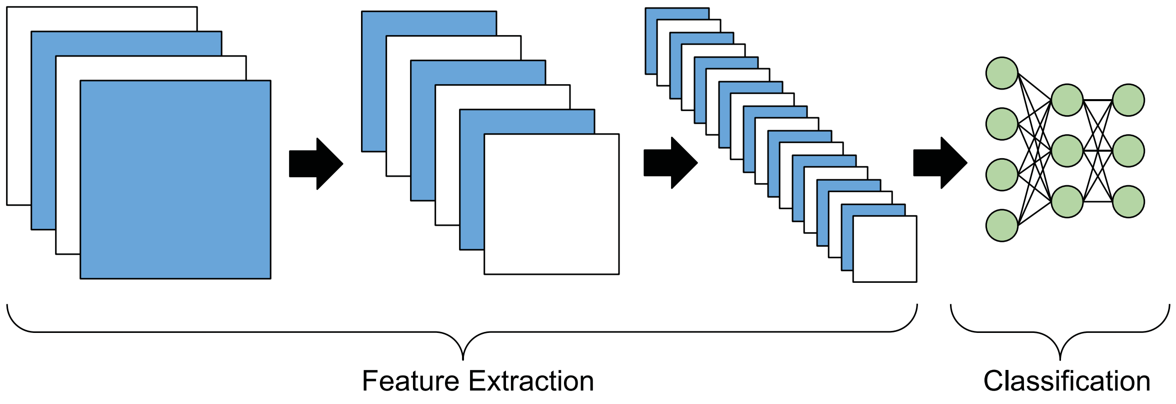


Figure 2.12: CNN architecture example.

Convolutional Layers apply what is called a 2D convolution. The equation for a 2D convolution is presented in Equation 2.3. Where F is the filter, I the matrix (image), M and N are the number of lines and columns respectively.

$$F[m,n] \oplus I[m,n] = \sum_{j=-N}^{N} \sum_{i=-M}^{M} F[i,j] * I[m-i, n-j] \tag{2.3}$$

In convolutional layers, given an input image I with size of M x N, a feature map is generated by performing a convolution operation $F \oplus I$. It uses a filter F, also known as kernel, that is commonly represented as a matrix of fixed size and randomly initialized. The filter slides through the image at a specified step, called “stride”, from left to right, top to bottom. Each element in the feature map is the sum of the products of each element from the filter and the corresponding elements this filter overlaps on the image [101]. A graphic representation of a convolution is represented in Figure 2.13.

Pooling layers are very similar to convolutional. It also has a kernel with a pre-defined size, that slides through the image, calculating the average value (average pooling), or the maximal value (max pooling) of the elements present at the kernel region [101]. The kernel moves at a certain pace defined by the stride value. This operation is represented in Figure 2.14.

Finally, a fully connected layer works as regular neural network with one hidden layer (see Figure 2.15). First the user defines the number of neurons and the number of outputs. Then, it works by feeding data as inputs, multiplying them by the weights, adding everything together, and applying the activation function, to generate the outputs.

Regarding the activation function, the rectified linear unit, commonly known as ReLU, is the most commonly used activation function in neural networks, especially in CNNs, in convolutional and dense layers. It is an activation function that applies an element-wise function such as sigmoid to the output of the activation produced by the previous layer. The ReLU activation is given by the equation $y = max(0, x)$. ReLU is linear (identity) for all positive values, and zero for all negative values, which means that, it is cheap to compute as there is no complicated math, and also that it helps converging faster, as linearity means the output does not saturate with larger values of x.

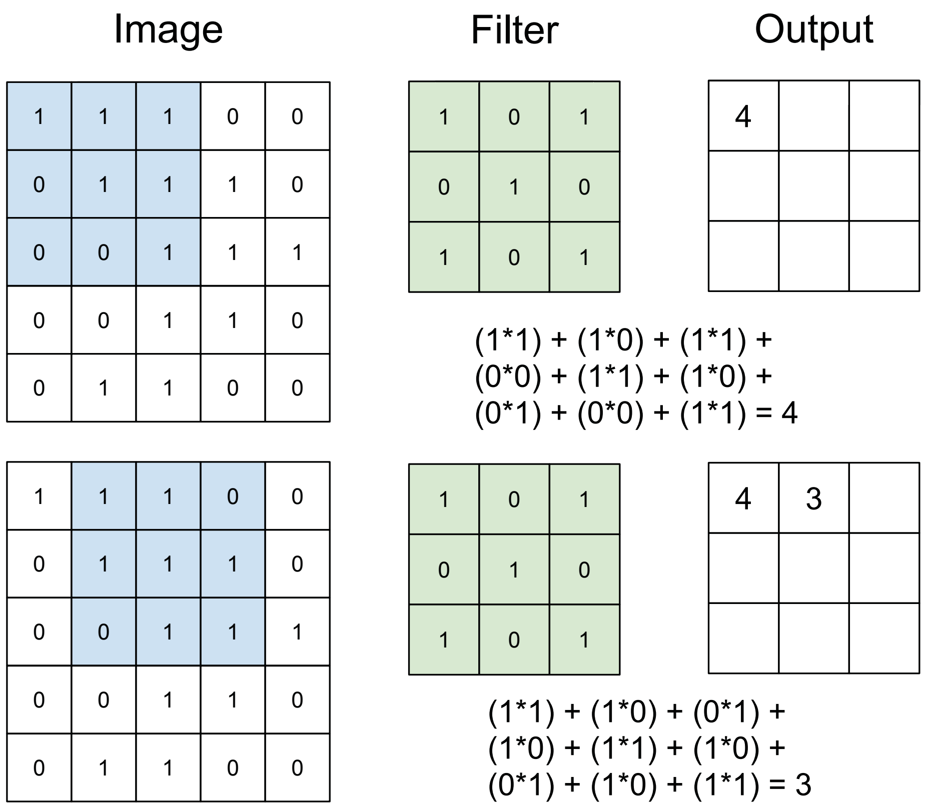


Figure 2.13: Example of a convolution operation using a 3x3 kernel.

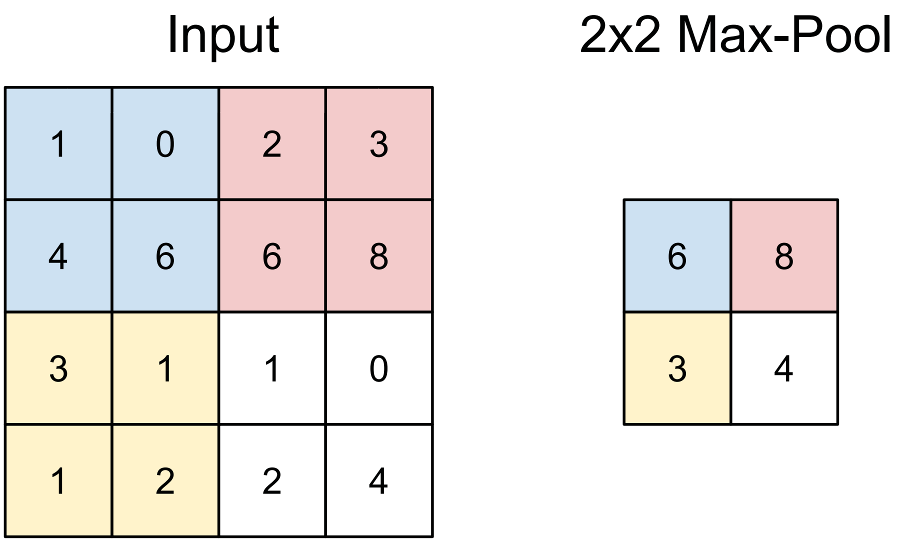


Figure 2.14: Example of a max-pooling operation with a kernel size of 2x2.

2.3.5 U-Net

The U-Net was proposed by Ronneberger, Fischer and Brox [88], with the objective of overcoming the limitations of CNNs when dealing with image segmentation tasks. Classical CNNs have had limited success due to the size of available datasets and the size of the networks.

The U-Net was proposed as an extension the architecture presented in [64], in which the main idea was to supplement the contracting part, replacing pooling operators by upsampling

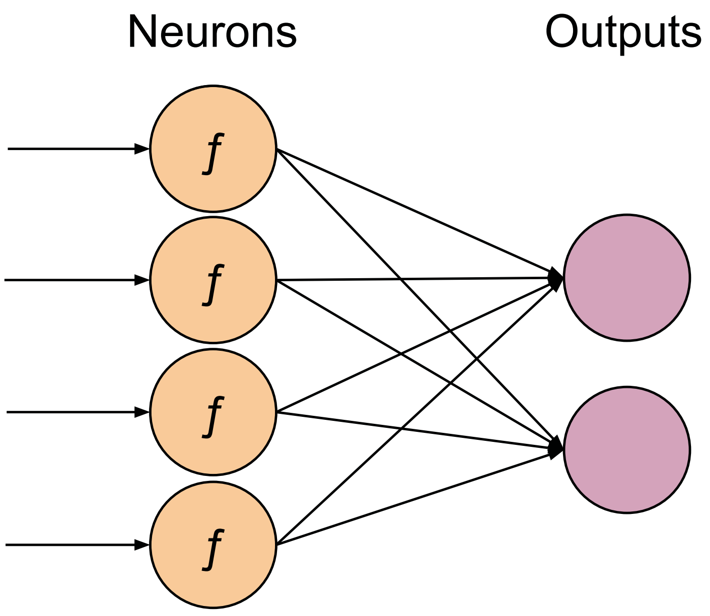


Figure 2.15: Example of a dense layer with 4 neurons and output size of 2.

operators. It combines the the high resolution information, resultando from the upsampling, with the high resolution feature from the contracting part, and apply convolutional layers to assemble more precise output based on this information.

Figure 2.16 presents a representation of the U-Net arhictecture. As it is mainly composed of convolutional and pooling layers, the U-Net is called a fully convolutional network [64]. The architecture is divided into three main parts: the contracting path, the expansive path, and middle. It follows a unique U-shaped structure, in which some connections between layers are used to retrieve information from previous layers, to reconstruct the output image. The contracting path is responsible for learning basic shapes while down-sampling the image, following a typical convolutional network scheme. It does this by applying repetitions of 3x3 convolutional layers, each followed by a ReLU activation and a 2x2 max-pooling with stride 2 for down-sampling. The expansive path is then responsible for reconstructing the image, learning more complex features while also using information from previous layers. Each step consists of an up-sampling followed by a 2x2 convolution, a concatenation with the corresponding cropped image from the contracting path, and two 3x3 convolutions followed by a ReLU activation. Usually considered part of the contracting part, the middle part is composed of a few convolutional layers and dropout layers. The output of the network is given by a 1x1 convolutional layer, which performs the classification for each pixel through a Sigmoid activation function. The networks does not have any dense layers and only uses the valid part of each convolutional, meaning the segmentation map only contains the pixels, for which the full context is available in the input image.

The U-Net was presented in [88] for tasks that have very little training data available, and thus, they use excessive data augmentation, proposed by Dosovitskiy in [25], by applying elastic deformations to the available training images. This allowed the network to learn invariance to such deformations, without having to them in the annotated image corpus.

2.4 CONCLUDING REMARKS

Population-based algorithms are often used for combinatorial optimization problems, where is infeasible to test all possible combinations. Moreover, GP techniques expands the application

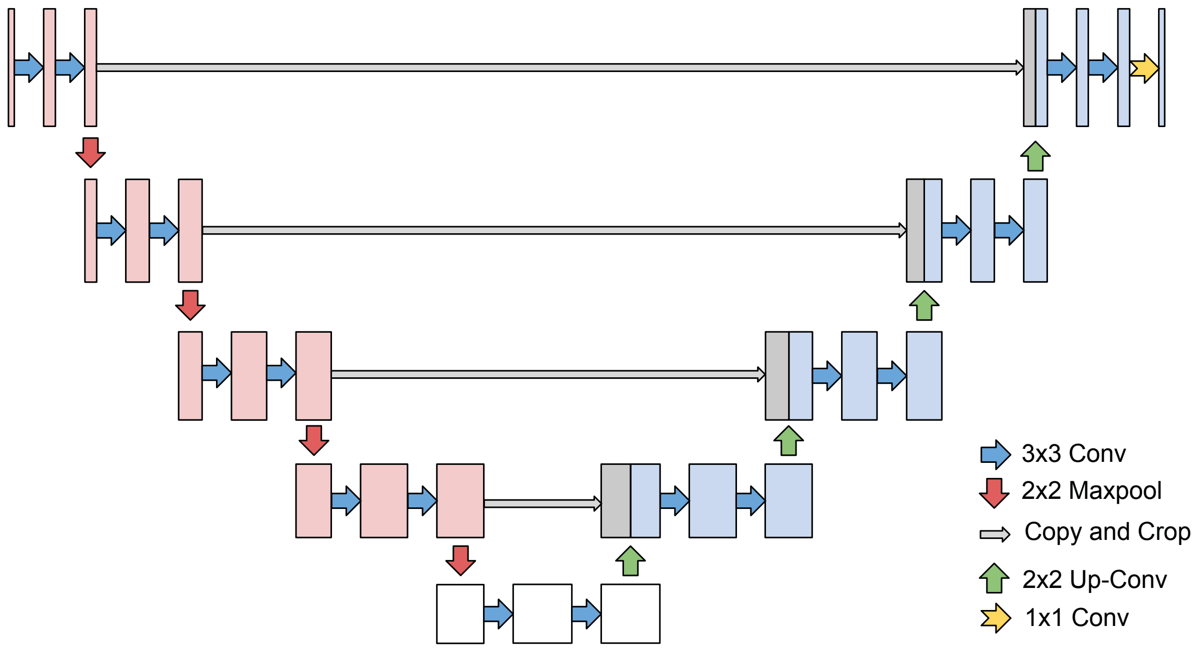


Figure 2.16: Vanilla U-Net architecture.

of evolutionary algorithms, to the design of computer programs, offering a more dynamic way to deal with the search space. Then, by analyzing how neural network architectures are often designed, as composed of a number of different layers, and presenting different architectures. For example, CNNs and U-Nets are two types of networks that present similar structures, composed of many convolutional, pooling, and dense layers. On the other hand, the way each architecture is designed, allows the use of the same components for totally different objectives. DNNs have not yet discovered the best way to combine layers and hyperparameters, and can be interpreted also as a combinatorial problem, where each piece of the puzzle has to be carefully placed, in order to produce good results.

3 RELATED WORK

This chapter presents a number of works related to the design of DNNs, starting from a more broad definition given by automated machine learning (AutoML), and narrowing down to specific topics, such as neural architecture search (NAS), which is a sub-field from AutoML, and even further with evolutionary-based design of neural networks covered in Neuroevolution. We highlight evolutionary-based methods, as they are closely related to the approach being proposed, more specific GP-based techniques, showing the different techniques and perspectives, regarding how networks are represented, encoded, built, and modified, as the search for improved designs unfold.

3.1 AUTOML AND NEURAL ARCHITECTURE SEARCH

According to [38], automating the entire pipeline of ML has emerged mainly as a way to reduce the costs related to the development and tuning of well-performing models, and creating what is known as AutoML. Among the many definitions for AutoML, [114] states that AutoML was designed to reduce the demand for data scientists and enable domain experts to automatically build ML applications without much requirement for statistical and ML knowledge. Moreover, [111] defines AutoML as a combination of automation and ML, involving the automated construction of an ML pipeline considering a limited computational budget. AutoML will include works that deal with the automation of pipelines of ML, including data preparation, feature engineering, model generation, and model estimation [37].

NAS is the term used to categorize techniques focused on automating the design of ANNs, a trendy topic in the field of automatic ML [30]. Methods for NAS can be categorized according to the search space, search strategy, and performance estimation strategy used:

- The search space defines the type(s) of ANN that can be designed and optimized;
- The search strategy defines the approach used to explore the search space;
- The performance estimation strategy evaluates the performance of a possible ANN from its design (without constructing and training it).

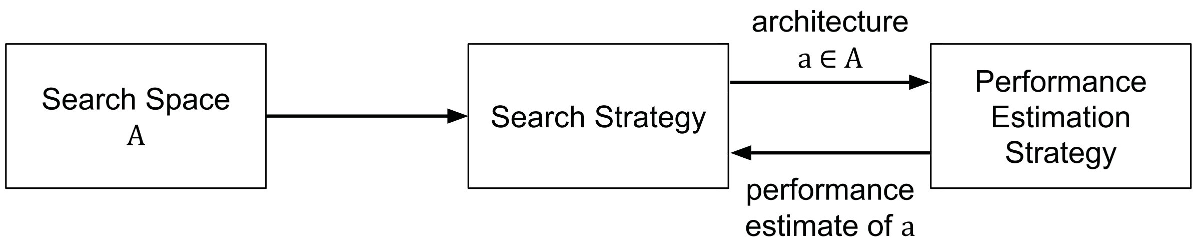


Figure 3.1: Abstract illustration of Neural Architecture Search methods. Adapted from [30].

A search strategy selects an architecture a from a predefined search space A. The architecture is passed to a performance estimation strategy, which returns the estimated performance of a to the search strategy. The described process of NAS methods is presented in Figure 3.1.

Recent work on NAS [14, 29, 112, 116] incorporates modern design elements known from hand-crafted architectures, such as skip connections, which allow to build complex, multi-branch networks, which results in significantly more degrees of freedom.

In the Survey of Elskem et al. [30], the authors cite works, such as [116], which optimizes two different kind of cells: a normal cell that preserves the dimensionality of the input and a reduction cell which reduces the spatial dimension. The final architecture is then built by stacking these cells in a predefined manner. Consequently, this cell-based search space is also successfully employed by many recent works [61, 85]. Moreover, the work analyzes three major advantages emerged from this strategy of designing networks from building blocks:

1. Reduction of the search space due to the use of less layers overall.

2. Architectures built from building blocks can be easily transferred or adapted to other data.

3. Creating architectures by repeating building blocks has proven a useful design principle in general, such as repeating an LSTM block in RNNs or stacking a residual block.

3.2 NEUROEVOLUTION

Neuroevolution can be seen as a sub-field of NAS that covers the techniques based on EAs to generate ANNs, by learning neural network building blocks (for example, activation functions), hyperparameters, architectures, and methods for learning weights [97]. Early studies [73, 109] already mentioned the use of evolutionary algorithms, a genetic algorithm, in this case, to evolve the connections of the networks. Regarding recent works, there exist a variety of studies and reviews [97, 30, 105] that cover the different approaches to the subject. The automatic design of neural networks has been studied for a long time, with different approaches ranging from evolving the weights, topology, architectures, parameters, and combinations.

Evolving the weights was the first attempt to combine evolutionary techniques and neural networks. Most works on this subject [24, 33, 83] follow the same idea of applying evolutionary algorithms as the mechanism that will learn and adjust the weights of the networks. This approach is reported as a faster, more efficient, yet robust, training procedure. However, for DNNs that comprise tens of thousands of parameters, the evolution of weights rapidly becomes infeasible. In recent works, [99] states that ES can rival backpropagation-based algorithms, such as Q-learning and policy gradients, on challenging deep reinforcement learning (RL) problems. They evolve the weights of DNNs with a simple GA, resulting in good performance for hard deep RL problems.

As the next step, there are many works [44, 55, 66, 72, 98] that explore the design of the topology of the networks. Neuroevolution of Augmenting Topologies (NEAT) is one of the first well-known methods that was proposed in [98], by Stanley and Miikkulainen. It was presented as a method that can outperform the best fixed-topology methods on a challenging benchmark reinforcement learning task. Later, Miikkulainen et al. in [72] expanded NEAT, proposing the CoDeepNEAT for optimizing DL architectures through evolution. CoDeepNEAT uses a coevolutionary strategy to evolve population of chromossomes, encoded as graphs, which starts with minimal complexity, and adds new nodes incrementally through mutation. By extending existing neuroevolution methods to topology, components, and hyperparameters, CoDeepNEAT achieves results comparable to the best human designs in standard benchmarks in object recognition and language modeling.

In a similar work from Liang, J et al. [55], the author propose LEAF, which can be used to optimize both the topology, hyperparameters, and the size of the networks. LEAF uses CoDeepNEAT in its core, and its composed of three main components: algorithm layer, system layer, and problem-domain layer. The algorithm layer is responsible for evolving the DNN hyperparameters and architectures. The system layer parallelizes the training of DNNs on the cloud. Finally, the problem-domain layer is where LEAF solves problems such as hyperparameter tuning, architecture search, and complexity minimization.

Furthermore, the use of GP to evolve DNNs became a more interesting option, as they offer flexible tools to design the networks. There are approaches that use the traditional Tree-GP [63], Cartesian GP [100], and Grammar-based GP [4].

In [63], Londt, Gao, and Andreae introduce GP-Dense, a novel GP-based algorithm coupled with an indirect-encoding scheme that facilitates the evolution of performing char-DenseNet architectures. During the evolution, the networks are evaluated using 25% of the dataset, and the best-evolved network is then evaluated using 100% of the dataset. Reported results indicate that even with the reduced dataset, the algorithm produced networks with relevant performance in terms of accuracy and number of parameters.

The work of Suganuma, Shirakawa, and Nagao [100] proposed the automatic design of Convolutional Neural Networks (CNNs) using Cartesian GP (CGP). They adopt highly functional modules, such as convolutional blocks and tensor concatenation, as the node functions. The use of graph-based encoding allows the creation of non-linear models, expanding the possibilities in the space of models.

Regarding grammar-based approaches, Assunção et al. present in [3] a grammar-based algorithm that is an improved version of the traditional GE. In this first work, they propose DSGE to evolve ANNs with more than one hidden layer. This work was later expanded in [4], introducing the evolution of DNNs. Moreover, a previous work (Lima and Pozo [58]) studied the design of CNN using the traditional GE, conducting experiments using the CIFAR10 and MNIST datasets. In other works, Lima et al. [59, 56], adapted the approach to employ DSGE, proposing a symmetric grammar that is particularly suited for the characteristics of the U-Net architecture.

Table 3.1 presents the summary of the analyzed works, categorizing them according to the strategy, and giving a brief description regarding the strategy.

Table 3.1: Summary of related work papers

Strategy	Papers	Description
Weights	[24, 33, 83, 99]	Evolves just the weights of the networks. First attemps on combining evolutionary techniques and NNs.
Topology and Hyperparameters	[44, 55, 66, 72, 98, 14, 29, 112, 116]	Extended existing neuroevolution methods to topology, components, and hyperparameters.
GP-based	[63, 100]	Explore the use of GP-based methods for the design of DNNs.
Grammar-based	[3, 4, 58, 59, 56]	Works focused on grammar-based techniques, with application to different types of networks.

3.3 CONCLUDING REMARKS

Each GP approach offers different perspectives on how to build the networks. GP-based approaches can use either a direct or indirect encoding for the solutions. A direct encoding can offer more control over the solution through adapted genetic operators. However, it increases the difficulties of representing the different architectures. On the other hand, an indirect encoding will require little or no modification to the genetic operators, as it is easier to use and modify a simplified representation, requiring only a mapping function to build the models. Furthermore, some approaches cover the evolution of both architectures and parameters, limited only by how flexible the building blocks are and how it impacts the generation of valid models.

4 PROPOSED APPROACH

In this chapter the concepts introduced previously, regarding evolutionary algorithms, genetic programming, and deep neural networks, are used to describe the proposed approach. First, an overview is presented, describing the components included in the approach, and how they interact. Following, a section about how to structure grammars for the design of DNNs is presented. It proposes some guidelines on how to define and represent layers and hyperparameters from neural networks as components of the grammar, including special cases, such as the use of custom layers, and the design of non-sequential architectures, like the U-Net. The next section describes the encoding used in the approach, combining what was already described in previous chapters regarding GE and DSGE, with the newly proposed grammar. The encoding and grammars are directly related to the mapping procedure, and the building process, which are responsible for translating an encoded solution into an executable DNN. In this section we describe how this translation works, explaining how the networks are built, covering simple networks like CNNs, and also more advanced cases like the U-Net. Moreover, we cover how we evaluate these networks, listing common approaches used in different cases. Finally, we present the custom genetic operators proposed for our approach, designed to extract max potential from both the grammar and encoding applied to the design of DNNs.

4.1 OVERVIEW

The proposed approach is called GE/DNN, and it is composed basically by the definition of a grammar, a mapping process, and configuring the search engine (see Figure 4.1).

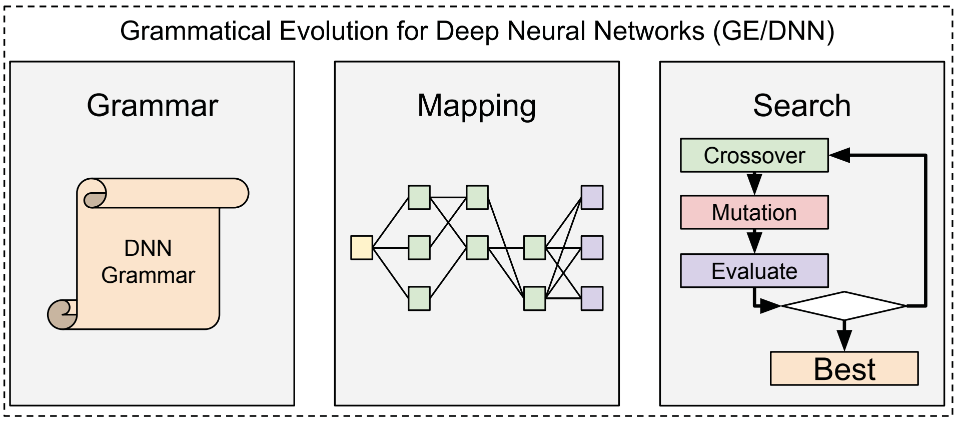


Figure 4.1: Main components of GE/DNN.

First, in the grammar, we define the layers and parameters used to describe the space of possible solutions. For instance, we might include Convolutional, Pooling, Dense, and Dropout layers for image tasks. While Long Short-term Memory (LSTM) [102], Recurrent [35], and Bidirectional [94] layers might be more suitable for text tasks. Furthermore, we propose to use the grammar to also include rules that influence the initial structure of the network, or rules that goes beyond simply selecting which layer to add, which later can be complemented by the mapping process.

Regarding the mapping process, building the networks requires more than just stacking up layers, as each type of network might employ a variety of connections between the layers and create architectures with multiple paths. In this case, prior knowledge about the problem and networks being explored, are valuable information that can be added to the grammar and mapping process, in the form of common blocks or set of blocks that are usually seen in the best architectures, or even adding to the grammar characteristics of the architecture itself, such as a specific shape or ramification strategy, as the U-Net for example.

Finally, the search engine is one of the most important pieces in this puzzle, since it will be the responsible for creating and combining the networks. A good search engine must balance the ability to maintain enough diversity, while also avoiding falling into a random search. This approach follows the traditional GE and uses a GA as the search engine. The combination of its populational nature, with the recombination operators, along with an indirect encoding, makes it very easy to adapt do many different situations.

Figure 4.2 illustrates the general scheme of GE/DNN. The execution follows the usual evolutionary cycle; solutions are randomly generated, selected, and recombined (through crossover and mutation) to produce new solutions. After reaching a stop criterion, the best solution found is returned. The grammar is the component that defines how solutions are created and evaluated, along with the dataset, that is directly related to the problem we want to solve.

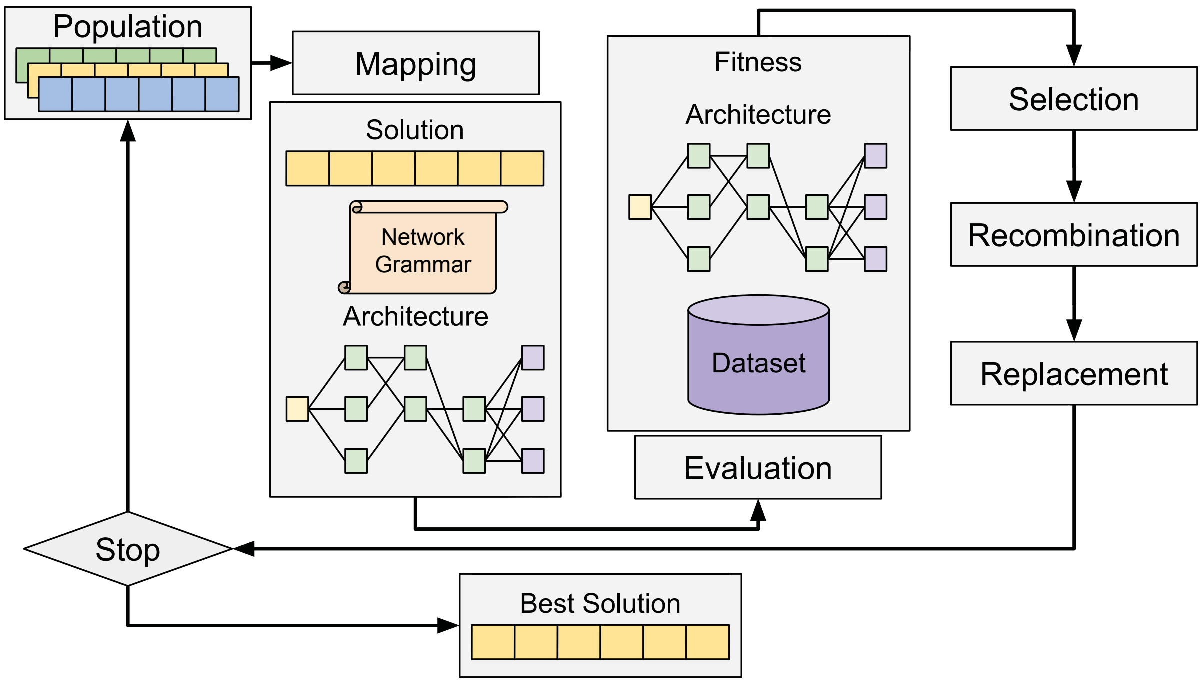


Figure 4.2: General scheme of the proposed approach.

An important aspect of making science is to make sure the work is reliable and reproducible. For this reason, the code for this approach is publicly available in Github at `https://github.com/rhrlima/cbioge`. It was implemented in Python3 mainly, using two well-known libraries for machine learning, Tensorflow[1] and Keras.

4.2 GRAMMARS FOR DNNS

As defined previously in the Section 2.2.1.1, a grammar is basically composed of a tuple $G = (N, T, S, P)$. In the context of designing DNNs, we define these elements in the same manner. Taking Grammar 4.3 as reference. The group of non-terminals N will be <start>,

[1]https://www.tensorflow.org

<conv>, and <dense>. The terminals T will be <pool>, <features>, <kernel>, and <units>. Notice that <conv> and <dense> were not included as terminals, even through they are layers, because their productions contain other rules. The rule <start> is used as the start symbol S. And finally, P denotes the relation between rules and the productions, presented in the form of A ::= B, where A is the rule, and B one or more options separated by the pipe symbol "|".

$\langle start \rangle$ ::= $\langle conv \rangle$ $\langle start \rangle$ | $\langle pool \rangle$ $\langle start \rangle$ | $\langle dense \rangle$ $\langle start \rangle$ | &

$\langle conv \rangle$ ::= Convolutional $\langle features \rangle$ $\langle kernel \rangle$

$\langle pool \rangle$ ::= Maxpool | Avgpool

$\langle dense \rangle$ ::= Dense $\langle units \rangle$

$\langle features \rangle$ ::= 16 | 32 | 64 | 128 | 256 | 512

$\langle kernel \rangle$ ::= 2 | 3 | 5 | 7

$\langle units \rangle$ ::= 16 | 32 | 64 | 128 | 256 | 512 | 1024

Figure 4.3: Example of a simplified grammar for designing DNNs.

The grammar does not follow any specific structure rather than the pair "rule" and "productions". For instance, the <conv> rule, that represents a convolutional layers, includes the <features> (number of feature maps) and <kernel> (kernel size) hyperparameters, each with their possible values, defined in their respective rules. Moreover we have the flexibility to add any combination of layers, hyperparameters, numeric values, intervals, custom layers, "special types", and any other element that might be useful when generating the architectures. At the end, the mapping process will be the responsible to parse the result of a grammar expansion and then build the network. However, there are some decisions regarding how the rules are designed in the grammar, which might affect the generated solutions and the probabilities for selecting some production over others. Two examples are presented bellow, showing some of the possibilities regarding these decisions.

$\langle rule\text{-}a \rangle$ ::= $\langle layer \rangle$ $\langle rule\text{-}a \rangle$
  | &

$\langle layer \rangle$ ::= $\langle conv \rangle$ | $\langle pool \rangle$ | $\langle dense \rangle$

$\langle rule\text{-}b \rangle$ ::= $\langle conv \rangle$ $\langle rule\text{-}b \rangle$
  | $\langle pool \rangle$ $\langle rule\text{-}b \rangle$
  | $\langle dense \rangle$ $\langle rule\text{-}b \rangle$
  | &

Figure 4.4: Example of different structures for grammar rules.

For example, in Figure 4.4 we have two different pieces of grammars, that provide the same representation power. The difference is that the grammar on the left has two productions for <rule-a>, <layer> and "&", and then <layer> contains the productions including <conv>, <pool>, and <dense>. In this setup, both <layer> and the empty production will have 50% chance to be picked, and once <layer> is selected, each production of it will have 33% chance to be selected. On the other hand, the grammar on the right contains all four productions included in <rule-b>. This setup gives 25% chance for each production, which includes the empty production. In summary, the second option has a smaller chance to select the empty production, compared to the first one, which might be desirable depending on the context.

$\langle rule \rangle$ ::= $\langle conv \rangle$ $\langle rule \rangle$
| $\langle resblock \rangle$ $\langle rule \rangle$

$\langle conv \rangle$ ::= Convolutional $\langle features \rangle$ $\langle kernel \rangle$

$\langle resblock \rangle$ ::= ResBlock $\langle features \rangle$ $\langle kernel \rangle$

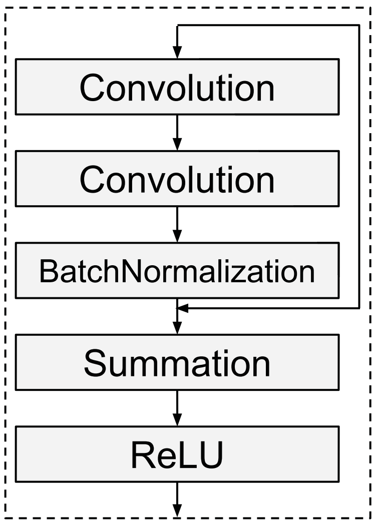


Figure 4.5: Example of grammar that includes a custom layer, identified as the <resblock> here.

A second example is presented in Figure 4.5, where a custom layer is added to the grammar as a way to define a group of layers, of a special type created for the problem. In this example, the rule <rule> contains the productions <conv>, <resblock>, which add a convolutional or a residual layer, respectively. In this context, the residual layer is a custom layer that is composed of two convolutional layers, one batch normalization, one sum operation, which adds the output from the previous layer and the original input, finalizing with a ReLU activation. For custom layers, specially groups of layers, to keep track of the input shape, and produced output shape, as they might interfere with the architecture, producing invalid solutions if not configured with caution.

4.2.1 Mirror Grammar

When designing more complex networks, such as a U-Net (see Section 2.3.5), a basic grammar like the example in Figure 2.6 will not be enough, requiring something that allows recreating the characteristics of U-shaped networks.

Figure 4.6 presents our proposal for a "mirror grammar". This grammar is able to generate architectures that respect the structural characteristics of U-Nets, finding a balance between the original aspects of the network and the possibility of exploring new designs. By assuming the U-Net as a symmetric structure, it be used to design only the contracting and middle parts of the network, and then build the expanding path by exploiting the symmetry. The solutions would be representing only half of the network, decreasing the representation complexity and also decreasing the chances of producing invalid connections between layers, and thus, having better control of the parameters.

Forcing a symmetric makes it easier to control how the architecture will grow, considering that all the steps done on the left side of the networks will be "undone" on the right side. And by undone, it means the process that reconstructs the image, including all the learning. On the other hand, this approach also restricts the grammar, once it assumes and pre-defines almost 50% of the architecture based on the first half.

In the presented example, the grammar "forces" the architecture to start with a convolutional layer, followed by a rule called <next>. Then <next> can be used to produce all the options usually seen in a U-Net, including convolutional layers, dropouts, and poolings. In addition to these, it is also including a production "bridge" and a rule <nextp>. In this context

$\langle unet \rangle$ ::= $\langle conv \rangle$ $\langle next \rangle$

$\langle next \rangle$ ::= $\langle conv \rangle$ $\langle next \rangle$ | $\langle dropout \rangle$ $\langle next \rangle$ | $\langle pool \rangle$ $\langle nextp \rangle$ | bridge $\langle pool \rangle$ $\langle nextp \rangle$

$\langle nextp \rangle$ ::= $\langle conv \rangle$ $\langle next \rangle$ | $\langle middle \rangle$

$\langle middle \rangle$ ::= $\langle middle \rangle$ $\langle middle \rangle$ | $\langle conv \rangle$ | $\langle dropout \rangle$

Figure 4.6: Example of a grammar designed for U-Nets.

"bridge" is being used to indicate where a concatenation layer will be added during the mapping and building process, which is the layers that connects both sides of the networks, creating the U-shape architecture. Moreover, the <nextp> rule stands for "protected", and it is used to prevent the addition of two sequential bridges to the architecture. This protected rule will only allow the addition of a new <conv> <next>, which restarts the cycle, or a transition to the middle part with the rule <middle>.

4.3 ENCODING

Following the encoding already defined for DSGE in Section 2.2.2.1, we use a list of lists, that differently from a usual matrix, not all rows will have the same length, yet they depend on the number of times that rule appeared during the mapping.

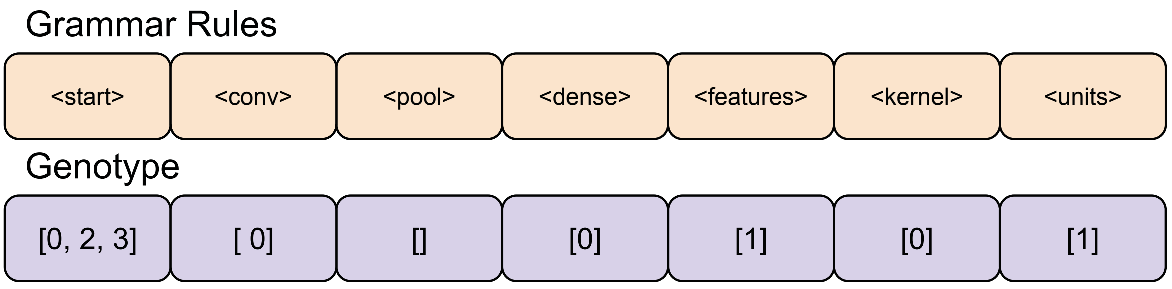


Figure 4.7: Example of genotype in GE/DNN using the DSGE encoding.

Figure 4.7 shows a sample solution that was created based on Grammar 4.3. For instance, rule <start> has three values in its list, meaning that this rule appeared three times during the creation/recombination. Then, the values 0, 2 and 3, are related to the first, third, and fourth productions. The length of a solution is defined by the number of non-terminals in the grammar. Moreover, each internal list will have only values used during the mapping, and the values will range from 0 to the number of options for each rule.

There is no strict pattern for how the grammar defines the sequence of the rules. As mentioned earlier, an encoded solution will have as many sub-lists as the number of rules in the grammar, and these sub-lists will follow the same sequence as the rules. According to Grammar 4.3 and Figure 4.7, both start <start> as the first rule/sub-list, followed by <conv>, and so on. This is an interesting property that we explore in the proposed genetic operators. By dividing the encoded solution, grouping all non-terminals in one side, and the terminals in the opposite side, we can have operators that affect the solutions differently, with bigger or smaller changes.

4.4 MAPPING

The mapping process for DNNs is the same as the original DSGE (see Section 2.2.2.3). For example, considering the solution presented in Figure 4.7, if the left-most terminal is <features>, according to the grammar in Figure 4.3, we have 16, 32, 64, 128, 256, and 512 as possible productions, and thus, the possibilities will range from 0 to 5.

Derivation step	Genotype
<start>	[[0, 2, 3] [0] [ ] [0] [1] [0] [1]]
<conv><start>	[[2, 3] [0] [ ] [0] [1] [0] [1]]
Conv <features><kernel><start>	[[2, 3] [ ] [ ] [0] [1] [0] [1]]
Conv 32 <kernel><start>	[[2, 3] [ ] [ ] [0] [ ] [0] [1]]
Conv 32 2 <start>	[[2, 3] [ ] [ ] [0] [ ] [ ] [1]]
Conv 32 2 <dense><start>	[[3] [ ] [ ] [0] [ ] [ ] [1]]
Conv 32 2 Dense <units><start>	[[3] [ ] [ ] [ ] [ ] [ ] [1]]
Conv 32 2 Dense 32 <start>	[[3] [ ] [ ] [ ] [ ] [ ] [ ]]
Conv 32 2 Dense 32 &	[[ ] [ ] [ ] [ ] [ ] [ ] [ ]]

Figure 4.8: DSGE mapping process for the solution presented in Figure 2.8.

Figure 4.8 presents the mapping of one possible solution generated from the grammar presented previously. Following the mapping steps, the axiom is represented by the rule <start>. The sub-list related to this rule is [0, 2, 3], and the first value consumed is 0, replacing <start> by <conv><start>. This process is repeated until terminal values replace all non-terminals, consuming all values from the solution. After mapping a solution, we have a list containing the sequence of layers and their parameters, that can be used to build the network. In the given example, the produced output was [Conv, 32, 2, Dense, 32] (ignoring the empty production), which translates to a network with one convolutional layer, with 32 filters, and a 2x2 kernel, followed by a dense layer with 32 units.

4.5 BUILDING

Sequential architectures are the most common ones, where the layers are sequentially connected, with no skip connections, or multiple inputs/outputs. Moreover, we can simplify the mapping and building process, by identifying pieces that will always be present in the architectures, such as the input and classifier layers. Figure 4.9 shows an example for a linear network, where the intput and classifiers are added during the building, and thus, those layers can be removed from the grammar.

However, in cases where we want to build more complex architectures, for example a U-Net, presented in Section 2.3.5, one of the possible approaches is to initially build the network as a sequential one, and later add the skip connections between the left and right side of the network. For example, by using the mirror structure proposed in section 4.2.1, U-shaped networks can be build following three steps. First, starting from the encoded solution, the mapping will occur as normal, generating a sequential list of layers (see Figure 4.10). In the example in Figure 4.10, we have a sequence of convolutional (cv), pooling (pl), some bridges along the way, and the transition to the middle part with dropout (dr) and more convolutional layers.

The second step consists of adding layers to the expanding part according the contracting part. Following some simple rules we are able to ensure that a symmetric architecture is created. For example, for each pooling layer, a upsampling layers is added to the expanding part. As

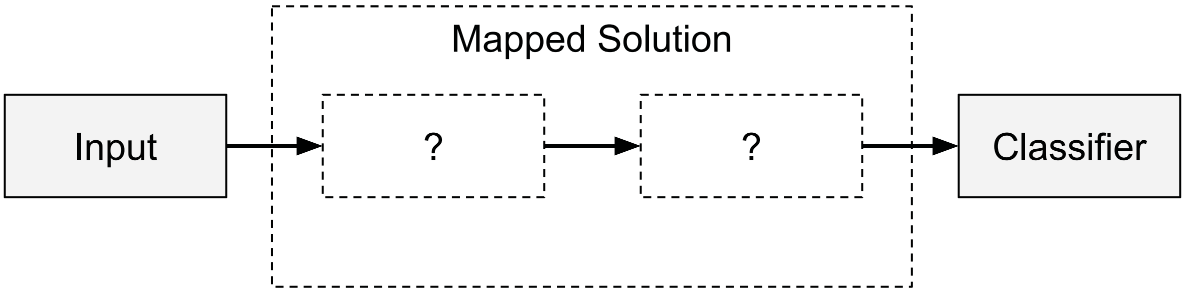


Figure 4.9: Sequential build example.

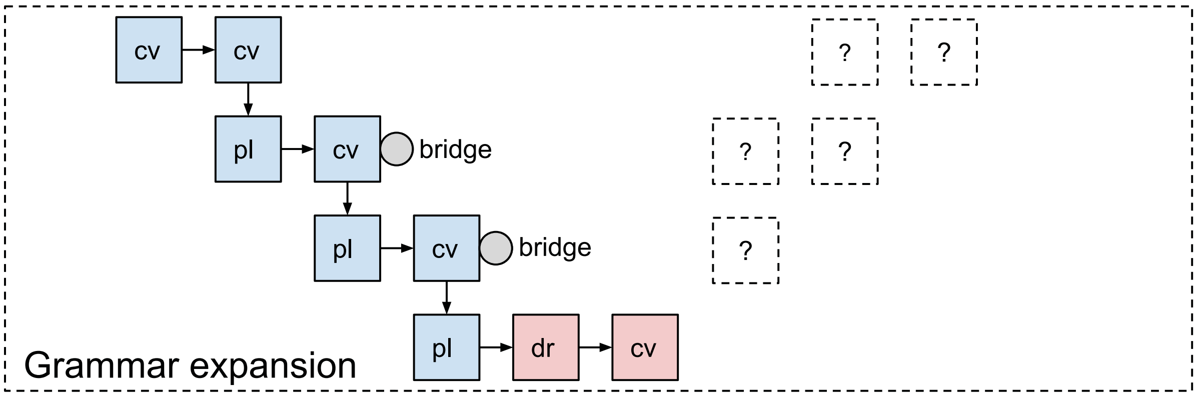


Figure 4.10: Mirror build step 1: performing the grammar expansion.

the pooling is responsible for decreasing the shape of the data flowing through the network, the upsampling will do the opposite. Moreover, a bridge connection can be added before a pooling, indicating that the previous layer will be concatenated with the output from the upsampling added to the expanding part. Also, a convolutional layer is added after every upsampling, responsible for learning features from the data resulted from the concatenation.

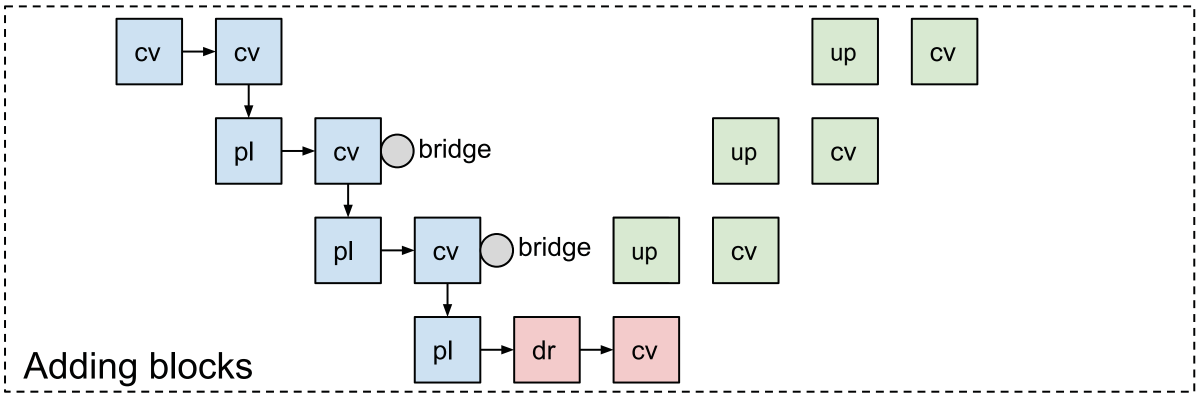


Figure 4.11: Mirror build step 2: adding the layer for the opposite side.

The third and final step is responsible for two things. First is adding the remaining layers that are common to every designed network, such as the input and output layers, as well as the classifier, which in U-Nets is represented by a 1x1 convolutional layer with sigmoid activation. The second is to connect all the layers, making sure they are valid connections regarding the input and output shapes, and also applying some repair procedure when necessary.

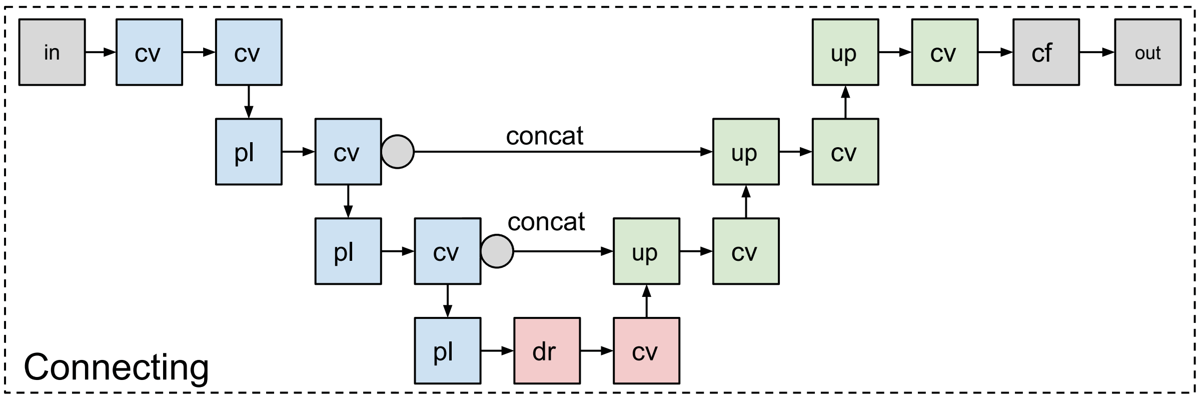


Figure 4.12: Mirror build step 3: connecting the layers.

4.6 EVALUATION

Evaluating a solution will first require the solution to be mapped, and then the network can be built and executed. For classification problems, the most common way is to use the accuracy achieved by the network on the dataset as the quality of that network. Depending on the problem, different metrics can be used, for instance, the loss function, similarity metrics, root mean squared error (RMSE), precision, and recall, among others. Executing the network means performing a full training using the training set, adjusting the weights according to the selected optimizer, batch size, epochs, and others parameters. Then, the accuracy of the network is computed by executing the trained network on the validation set. Furthermore, the best-evolved network is re-trained, calculating the final accuracy using the test set after the evolution ends.

Other strategies were tested, for instance, evaluating networks with no training phase (random weights). The hypothesis behind this option was to verify if "good" networks, i.e. models with good fitness, would achieve even better performance when re-evaluated after the evolution with a proper training phase. After some empirical experimentation, it was concluded what a good performance with random weights do not guarantee that the model will perform well after trained.

4.7 GENETIC OPERATORS

There are four basic operators in an evolutionary algorithm: selection, crossover, mutation, and replacement. In GE/DNN we adapted these operators to work with the new encoding, and extract maximum potential from the characteristics of each type of network. Selection and replacement follow the same behavior seen in the literature; for instance, we used tournament selection and replacement with elitism. Crossover and Mutation were also adapted, including the creation of two mechanisms to help controlling diversity in the population. They are not actually operators, however, are used as such as they are applied during the recombination phase.

4.7.1 Crossover

Our adaptation for the crossover follows a similar behavior of a simple one-point crossover, from where two variations were created. The first, named after the original One-Point Crossover, starts with two parents, where a random "cut" point is selected, and then creates the new solution by combining the first part from one parent and the second part from the other parent, respecting the blocks (sub-lists). The second variant is called Gene Crossover, which similarly to the previous one, will select a “cut” point, this time for each individual sub-list, and then create the new solution from the combination of the parents. A graphical representation of the two crossovers is shown in Figure 4.13.

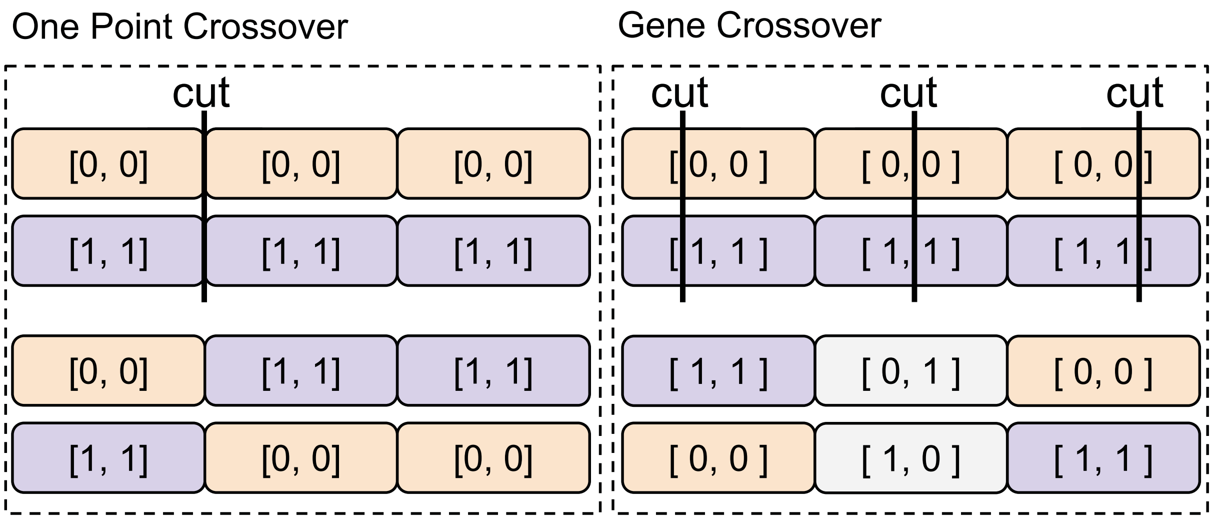


Figure 4.13: DSGE One-Point and Gene crossover examples.

4.7.2 Mutation

The adaptation of the mutation also follows the original point-mutation behavior. A random gene in the solution is selected to be replaced by a new valid one. The difference here is that this process can be applied to the whole genotype and produce a greater variety of changes, or be restricted to parts of the genotype related to either non-terminals or terminals. When restricted, the operator may produce changes with a bigger impact when changing non-terminal values or fine-tuning by changing terminal values. GE/DNN proposes two new mutation operators, Terminal Mutation, and Nonterminal Mutation, which are described below.

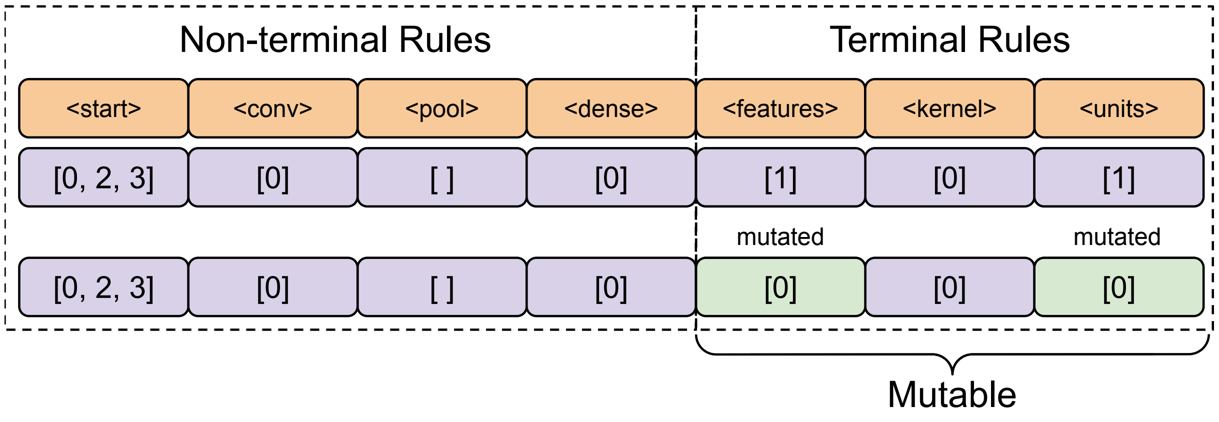


Figure 4.14: Terminal Mutation example. The replaced values will produce smaller changes to the network, acting like fine tuning.

In the Terminal Mutation (see Figure 4.14), what happens is that each sub-list has a chance to have one of their values replaced. Replacing a terminal value will, for example, change the number of filters on one convolutional layer from 256 to 64.

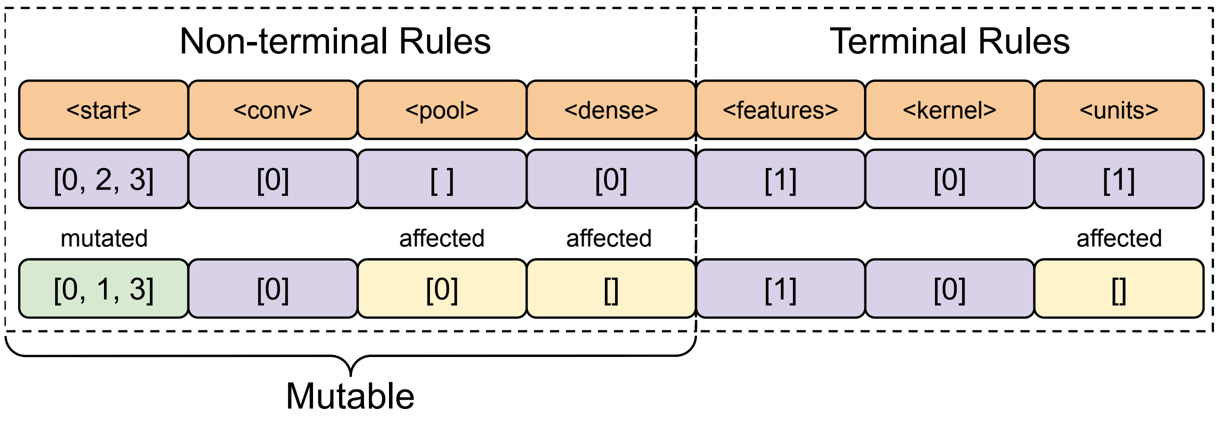


Figure 4.15: Nonterminal Mutation example. Replaced values might impact other parts of the solution, causing big changes to the network, including the necessity of adding more values, or remove some.

In our Nonterminal Mutation (see Figure 4.15), similar to the previous operator, each sub-list will have a chance to replace of the of values inside it. When a non-terminal value is replaced, it might produce a structural change that, for example, adds three new layers to the architecture. And thus, when such chances happen, other sub-lists are affected, as now we have three convolutional layers, instead of one, or in other cases, less layers than before. For both cases, the mapping process "repairs" the solution, adding values as needed, or removing unused values.

4.7.3 Diversity Control

The first proposed mechanism is called Half and Half, which combines crossover and mutation. It works by applying either crossover or mutation with a given probability. For example if crossover is defined with a probability of $p = 0.6$, then the mutation will have $1.0 - p$, which is 0.4. Half and Half is more like a meta-operator, as crossover and mutation must be defined previously.

A second diversity mechanism was proposed with the objective of maintaining high diversity in the population, working as "movement acceptance". The way it works is by checking for every new solution, created or recombined, if it was already known, and replacing it by another solution if so. In other words, every solution ever created during the evolution is registered in a list, so no duplicates are accepted. The impact on the evolution time is very low, considering how diverse a simple grammar can be in generating different networks. This mechanism is not always necessary, being dependent on the context of the application.

4.8 REPAIR SOLUTION

Invalid solutions are defined by solutions which do not produce a executable architecture. As mentioned earlier, there are some cases where invalid solutions are generated as product of recombination. The impact of having too many invalid solutions might affect the evolutionary process. Combining two valid solutions can produce a invalid one, and it is more likely to also produce invalid solutions when combining other invalid ones. In the end the population can have a premature convergence full of invalid architectures.

This approach has two repair mechanisms, one generic that is applied when necessary, and an optional one that is domain dependent, and must be adapted to the problem. The generic repair system is applied to the encoded solutions, on the search engine level, by adding new values when necessary, and removing unused values. The optional repair system can be applied to the intermediate representation, generated during the mapping procedure, where domain-specific changes will be applied in order to repair the architecture. It cover cases such as invalid connections between layers, modifying hyperparameters to ensure that the connection won't lead to invalid outputs, such as decreasing an image below 1x1.

4.9 CONCLUDING REMARKS

GE/DNN can be defined through its three main components: the grammar, mapping, and the search engine. Grammars can be easily adapted to the design of DNNs, but also include more elaborated rules, and strategic decisions, which affects how architectures are built. The mapping is then responsible for expanding and applying what was defined in the grammar, building a variety of networks, from linear CNNs, to U-shaped networks. How the solution was encoded, and how the grammar was structure, will affect the mapping process, into building the networks. Moreover, the search engine explores the use of a GA, proposing new genetic operators, which were adapted to the design of DNNs, including modified crossover, mutation, and other operators, to generate, combine, and evaluate the solutions.

5 APPLICATIONS

In this chapter, we present three scenarios in which GE/DNN is applied, as a proof of concept on designing DNNs. Each application addresses a different type of network, and also a different problem. The first application focuses on the design of CNNs for image classification, a classic problem, with a variety of architectures. The second application regards image segmentation, covering a different perspective on how the networks can be built, and also the metrics used to evaluate them. The third application is a study performed by a colleague, in collaboration to expand the approach and test it on different domains, in this case, text classification.

5.1 RESEARCH QUESTIONS

The main objective of this research is proposing an approach that is able to produce high quality networks, that are adapted to each problem, and reducing the need for expert knowledge and resources, usually seen on manual designs. The following research questions are proposed to discuss the individual aspects that helps addressing the main objective.

- RQ1: Is the approach easily adapted for different types of networks?
- RQ2: Are the networks generated by this approach competitive with other evolutionary design methods?
- RQ3: Can automatically designed networks perform better or similarly to manually created designs?
- RQ4: What novelty does this approach brings, compared to other neuroevolution techniques?

5.2 DESIGN OF NETWORKS FOR IMAGE CLASSIFICATION

CNNs are one of the reasons why research on DL is a hot topic in AI [72, 101], and presented as a powerful approach when applied to computer vision tasks, delivering state-of-the-art architectures. Recent studies on the manual design of novel CNN architectures [37, 42, 103] reported high accuracy, showing the potential of using such techniques.

CNNs are a popular option for classification tasks, where the network is used to learn the features from data and then classify the data into classes. In this application, we propose a grammar that includes the most common layers found in state-of-the-art CNN architectures, searching for the best performing design on the CIFAR-10 [51] dataset, a well-known dataset for image classification tasks. We compare our GE/DNN architectures with the designs found by two GP-based approaches, DENSER [4] and CGP [100, 108].

5.2.1 CNN grammar

For the CNN grammar (Figure 5.1), we use five types of layers, commonly found in CNNs architectures. The layers are convolutional, residual, pooling, dropout, and dense; each layer includes some of the hyperparameters related to it, such as number of filter and kernel size for convolutional layers and residual layers, number of units for the dense layers, and the rate for dropout layers. In our context, the residual layer [37] was defined as a group of two convolutional layers, batch normalization, summation, and a ReLU activation, with a skip connection from the input to the summation layer.

$\langle cnn \rangle$ ::= $\langle layer \rangle$ $\langle layer \rangle$ $\langle layer \rangle$ $\langle layer \rangle$ $\langle layer \rangle$ | $\langle layer \rangle$ $\langle layer \rangle$ $\langle layer \rangle$ | $\langle layer \rangle$

$\langle layer \rangle$ ::= $\langle l\_type \rangle$ $\langle layer \rangle$ | $\langle l\_type \rangle$

$\langle l\_type \rangle$ ::= $\langle conv \rangle$ | $\langle resblock \rangle$ | $\langle pool \rangle$ | $\langle dropout \rangle$ | $\langle dense \rangle$ | &

$\langle conv \rangle$ ::= conv $\langle filters \rangle$ $\langle ksize \rangle$ $\langle strides \rangle$ $\langle padding \rangle$ $\langle activation \rangle$

$\langle resblock \rangle$ ::= resblock $\langle filters \rangle$ $\langle ksize \rangle$

$\langle dense \rangle$ ::= dense $\langle units \rangle$ $\langle activation \rangle$

$\langle dropout \rangle$ ::= dropout $\langle rate \rangle$

$\langle pool \rangle$ ::= $\langle pool\_type \rangle$ $\langle ksize \rangle$ $\langle strides \rangle$ $\langle padding \rangle$

$\langle pool\_type \rangle$ ::= maxpool | avgpool

$\langle activation \rangle$ ::= relu | selu | elu | tanh | sigmoid | linear

$\langle padding \rangle$ ::= valid | same

$\langle filters \rangle$ ::= 16 | 32 | 64 | 128 | 256 | 512

$\langle strides \rangle$ ::= 1 | 2

$\langle ksize \rangle$ ::= 1 | 2 | 3 | 4 | 5

$\langle units \rangle$ ::= 32 | 64 | 128 | 256 | 512 | 1024

$\langle rate \rangle$ ::= [0.0,0.5]

Figure 5.1: CNN grammar. It includes structure rules, layers and hyperparameters. The most common layers for CNN were used. For hyperparameters, we also included special types, for example, the <rate> rule, which samples a random value within the defined range when selected.

In order to support the creation of larger networks, the first rule <cnn> offers three start points, with one, three, or five <layer> non-terminals. The redundancy helps creating different start points for the networks, rather than creating all designs starting with one single layer, we provide them the chance of producing bigger networks, without having to heavily depend on randomness to grow. Rule <layer> can also be used to grow the structure by adding one of the five layers we defined through the <l-type> non-terminal, followed by a new occurrence of <layer>, or not, giving the grammar a recursive tool. The remaining non-terminals are used to describe the layer types and their hyperparameters.

Building the networks designed by this grammar is very simple. During the mapping and building processes, two extra layers are added to every design, the input layer, already configured with the shape from the data that will be fed, a flatten, which transforms the 2d matrix input into a 1d vector, and finally a dense layer, also configured with the number of units as the number of classes, which will be used as the classification layer of the network. The input and classification layers are the same regardless of the architecture, meaning that there is no need to add them to the grammar.

5.2.2 Experiments

Table 5.1 presents the configuration of the evolutionary search. We perform 10 independent runs, with a population of 20 solutions, for 25 generations. These values are small, mainly due to the computational cost involved in training many networks. Moreover, genetic operators include tournament selection, our Half and Half recombination (described in Section 4.7), with one-point crossover, non-terminal mutation, and replacement with elitism. During the evolution, each model is trained using the Adam optimizer [46], with default parameters, for 50 epochs, and the fitness is calculated using a validation set.

Table 5.1: Parameters of the evolutionary search of CNNs.

Parameter	Value	Parameter	Value
Population size	20	Generations	25
One-point Crossover	60%	Non-Term. Mutation	40%
Tournament size	5	# of Elites	25%
Model Epochs	50/500	# of Runs	10

In the end, the best design found is re-trained 10 times, using the same configuration proposed in [4], with stochastic gradient descent (SGD) optimizer, with a learning rate of 0.01 and momentum of 0.9, for 500 epochs. Basically, Adam is faster, compared to SGD, regarding the optimization of the loss function, however, for a higher number of epochs, it has difficulties for improving the weights of the architectures. Then, the SGD optimizer, despite being slower, is more reliable when used for longer training, decreasing the risks of producing overfitting.

The dataset CIFAR-10 [51], consists of 60.000 32x32 colored images (Figure 5.2), divided into 10 classes, with 6.000 images per class. The dataset is divided into five training batches and one test batch, each with 10.000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Following the methodology described in [100], for this experimentation, we used only 10% of the training set, divided into 4500 images for training, 500 for validation, and the test set (10.000 images), for testing the best network. Moreover, the training set also employs the following data augmentation techniques: padding 4 pixels on each side followed by a random 32×32 crop from the padded image and random horizontal flips on the cropped 32×32 image. The main reason for this is to provide variety for the data, considering it uses only 10% of the dataset. The final network quality is calculated using the test set.

Table 5.2: Classification accuracy and loss for the CIFAR-10 dataset, with average accuracy and standard deviation.

Model	Accuracy	Loss	Mean (StdDev)
CGP-CNN (ResSet1)	74,16	1,70	73,42 ± 0,89
CGP-CNN (ResSet2)	73,22	1,93	71,22 ± 1,76
DENSER	72,93	1,62	67,68 ± 5,60
GE/DNN	74,28	1,79	72,42 ± 1,65

Table 5.2 presents the accuracy, loss, and number of parameters for CGP [100], DENSER [4], and GE/DNN, and also the mean accuracy across all runs including the standard deviation.

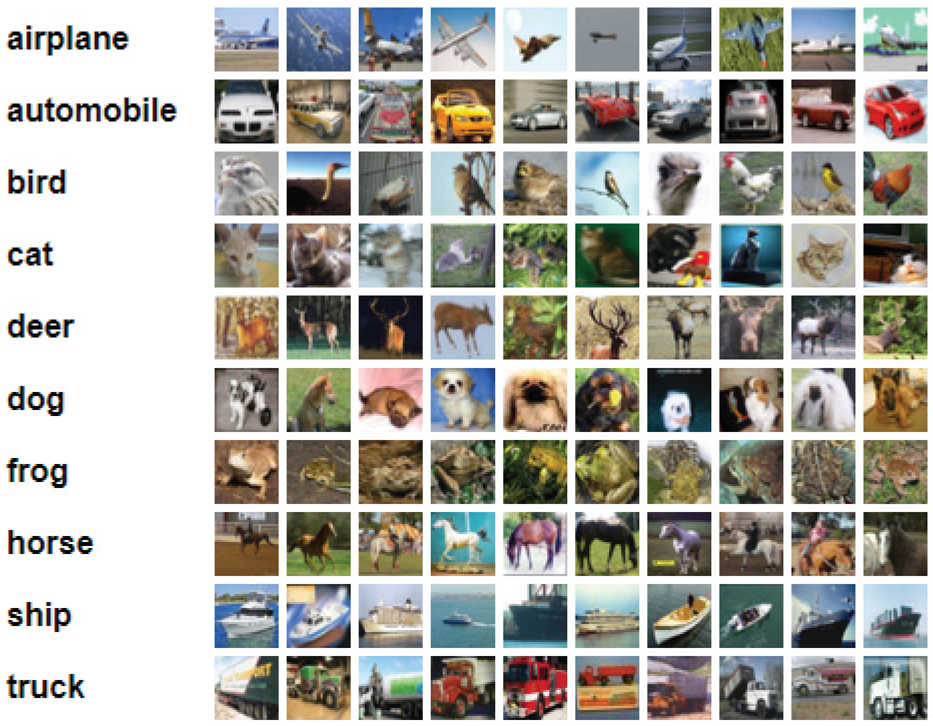


Figure 5.2: Image examples from the CIFAR-10 dataset. The dataset includes 10 arbitrary classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

Our approach took took, on average, 38 hours per run, running on a i3-9100f CPU, with 16gb RAM, and a GTX 1660, 6gb VRAM. After training all best models for 500 epochs and evaluating them on the test set, the GE/DNN networks achieved higher accuracy values, with 74,28% against 74,16% for CGP, and 72,93% for DENSER, on the best models. For the loss, GE/DNN achieved 1,79 on the best model, compared to 1,7 for CGP, and 1,62 for DENSER. On the other hand, CGP was able to generate less complex networks with 1.64 million parameters, compared to 3.55, and 10.08 million for GE/DNN and DENSER respectively. We should add that the DENSER model was designed using the full data from CIFAR-10, and thus, this probably affected the performance in our reduced scenario.

The architecture of our best CNN is represented in Figure 5.3. It combines convolutional, residual, pooling, and dense layers, in uncommon sequences, with dense layers appearing in early layers, rather than at the end, and also the preference for the Elu activation instead of ReLU. It is noteworthy that the representation of the network does not show the input and classification layers, which is composed of a flatten layer followed by a dense layer with the number of units equal to the number of classes of the dataset.

5.3 DESIGNING NETWORKS FOR IMAGE SEGMENTATION

Segmenting an image consists of dividing it in multiple segments of pixels to describe objects. The objective is to simplify the information, modifying its representation into some semantic representation which is easier to analyze [95]. Image segmentation tasks are classification problems where a label is assigned for every pixel individually, thus creating a new image as output. This kind of approach is commonly used to identify objects and boundaries (lines, curves, shapes, etc.), in images, and often applied to Medical [26, 75, 80, 107], Security [1, 10], and other modern systems, such as self-driving cars [92].

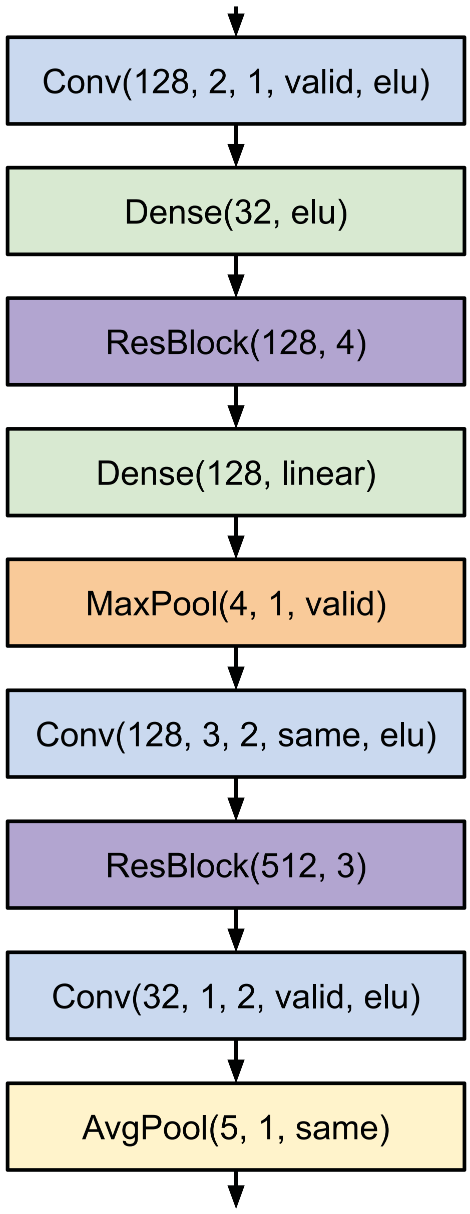


Figure 5.3: Best CNN architecture designed by the GE/DNN approach for the CIFAR-10 dataset.

The U-net [88] is the most popular deep learning model used for image segmentation for medical applications [77, 80, 91, 107]. As described in Section 2.3.5, its architecture is composed basically of convolutional and pooling layers. The non-sequential organization of its layers is one of the most relevant characteristics, which gives the network the ability to re-use information from previous layers in order to learn high-resolution features. These characteristics add new restrictions, which make the design of U-nets harder, compared to classical CNNs.

5.3.1 U-Net grammar

For designing U-shape architectures, we introduce a mirror-grammar that explores the characteristics of the U-Nets. It allowed us to design U-shaped networks with broader variety, regarding the overall structure and components, while also achieving a low probability of producing invalid models during generation.

The proposed grammar is shown in Figure 5.4. This grammar assumes the U-Net has a symmetric structure; thus, we used it to design only the contracting (left-side) and middle parts of the network and then build the expanding path (right-side) by exploiting the symmetry. By representing only half of the network in each solution, the representation complexity is reduced, and we also decrease the chances of producing invalid connections between layers, having better control of the parameters while guaranteeing the generation of 100% valid solutions.

⟨*unet*⟩ ::= ⟨*conv*⟩ ⟨*next*⟩

⟨*next*⟩ ::= ⟨*conv*⟩ ⟨*next*⟩ | ⟨*dropout*⟩ ⟨*next*⟩ | ⟨*pool*⟩ ⟨*nextp*⟩ | bridge ⟨*pool*⟩ ⟨*nextp*⟩

⟨*nextp*⟩ ::= ⟨*conv*⟩ ⟨*next*⟩ | ⟨*middle*⟩

⟨*middle*⟩ ::= ⟨*middle*⟩ ⟨*middle*⟩ | ⟨*conv*⟩ | ⟨*dropout*⟩

⟨*conv*⟩ ::= conv ⟨*filters*⟩ ⟨*k_size*⟩ ⟨*strides*⟩ ⟨*padding*⟩ ⟨*activ*⟩

⟨*pool*⟩ ::= ⟨*p_type*⟩ ⟨*strides*⟩ ⟨*padding*⟩ ⟨*padding*⟩

⟨*dropout*⟩ ::= dropout ⟨*rate*⟩

⟨*p_type*⟩ ::= maxpool | avgpool

⟨*filters*⟩ ::= 16 | 32 | 64 | 128 | 256

⟨*k_size*⟩ ::= 1 | 2 | 3 | 4

⟨*strides*⟩ ::= 1 | 2

⟨*padding*⟩ ::= valid | same

⟨*activ*⟩ ::= relu | sigmoid | linear

⟨*rate*⟩ ::= [0.0, 0.5]

Figure 5.4: U-Net mirror grammar. The structural rules focus on building the left side of the network.

The mapping and building processes had to be adapted. During the mapping process, the layers and hyperparameters for the expanding path are selected according to the result obtained by the mapping of the contracting part. Later, building the network will apply the configuration, repairing the network when needed. More details about the how the U-shaped networks are built, are given in Section 4.5.

5.3.2 Experiments

GE/DNN is applied for two image segmentation datasets, the BSDS500 [2] and ISBI12 [15], with the objective of searching for a network design that best performs in each dataset. Then, we compare the best designed networks with the original U-Net using four image similarity metrics. Since there is a variety of metrics, each one focuses on different characteristics of the image to calculate how similar two images are.

Here, four metrics are introduced: Jaccard, Dice, Sensitivity, and Specificity. The Jaccard coefficient (Jacc, Eq. 5.1), also known as Intersection over Union, is a statistic used for gauging the similarity and diversity of sample sets. The Dice coefficient (DCS, Eq. 5.2) is similar to the Jaccard but more popular in image segmentation tasks. We also use the well-known Sensitivity and Specificity metrics. Sensitivity, or true positive rate (TPR, Eq. 5.3), measures the proportion of actual positives that are correctly identified as such (e.g., the percentage of foreground pixels correctly identified as foreground). Specificity, or true negative rate (TNR, Eq. 5.4), measures the proportion of actual negatives that are correctly identified as such (e.g., the percentage of background pixels correctly identified as background). The fitness we proposed is then calculated as a weighted average (Eq. 5.5) of the presented metrics. The weights can be adjusted according to either how easy or hard is to satisfy each metric.

$$Jacc = \frac{|A \cap B|}{|A \cup B|} \quad (5.1) \qquad DCS = \frac{2|A \cap B|}{|A| + |B|} \quad (5.2)$$

$$TPR = \frac{TP}{TP + FN} \quad (5.3) \qquad TNR = \frac{TN}{TN + FP} \quad (5.4)$$

$$fitness = \frac{a * Jacc + b * DCS + c * TPR + d * TNR}{a + b + c + d} \quad (5.5)$$

Table 5.3 present the parameters used for the experiments. We performed 10 runs, generating 10 different U-shaped networks for each dataset. The population size was set to 20, evolved for 25 generations, resulting in about 500 evaluated networks per run. For recombination, we used our adapted crossover and mutation, tournament selection, and replacement with elitism. During the evolutionary process, the models were trained for 10 epochs and with a time limit of 60 minutes, with the quality of the network being calculated using the validation set. In the end, the best-performing model is delivered and re-trained for 500 epochs, now having its performance calculated on the test data. Adam optimizer, with default parameters, was used for both the evolution and the best network.

The weighted mean used as the fitness was configured using 0.4 to Dice and Sensitivity and 0.1 to Jaccard and Specificity. The two metrics with smaller weights were not able to give much feedback to the evolution, as they were either getting high values too easily or becoming too difficult to improve. It was concluded that such metrics were not able to track fine tuning on the segmentation produced by the models, and thus, not being a suitable fitness function for this application.

Table 5.3: Parameters of the evolutionary search of U-Nets.

Parameter	Value	Parameter	Value
Population size	20	Generations	25
Crossover	60%	Mutation	40%
Tournament size	3	# of Elites	10%
# Epochs	10/500	Model time limit	60 min

The Berkeley Segmentation Dataset and Benchmark[1] (BSDS) [2] is a large dataset of natural images that have been segmented by human observers (Figure 5.5). The human annotations serve as ground truth for learning grouping cues as well as a benchmark for comparing different segmentation and boundary detection algorithms. The dataset is divided into the three

[1]https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/

sets usually used for learning approaches: training, validation, and test sets, with 200, 100, and 200 images. As the images do not share the same width and height across the dataset, we resized both the image and its corresponding contour image to the size of 256x256 pixels.

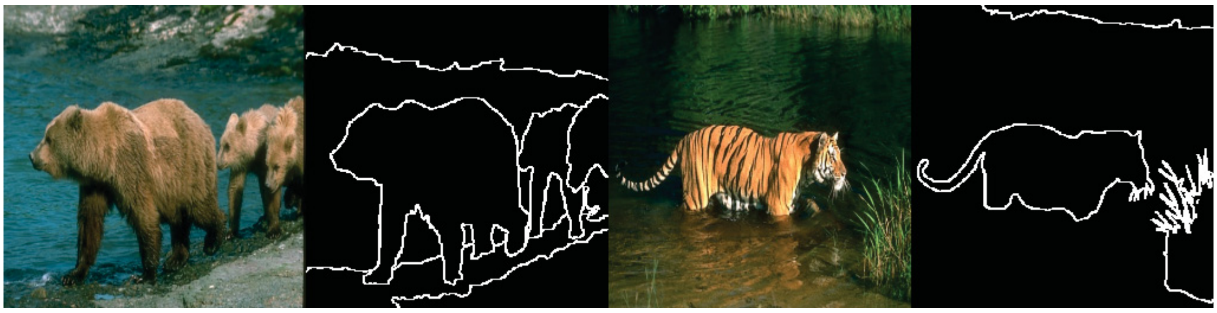


Figure 5.5: Examples extracted from the BSDS500 dataset, along with their respective contour images.

The serial section Transmission Electron Microscopy (ssTEM) dataset[2] [15] is a set of 60 images, along with their ground truth segmentation, 512x512 pixels grayscale, divided into 30 images for training, and 30 for the test. For the proposed experiments, we applied data augmentation techniques in order to enlarge the dataset. In total, we provide 300 images for training, 30 for validation, and 30 for the test, all resized to 256x256 pixels.

Table 5.4: Segmentation accuracy and loss for each dataset, with mean accuracy and standard deviation.

Model	Accuracy	Loss	Mean (StdDev)
BSDS500			
GE/DNN	53.46	0,16	50,04 ± 0,036
UNET	51.59	0,05	49,45 ± 0,021
ISBI12			
GE/DNN	86.10	0,15	82,77 ± 0,057
UNET	84.44	0,17	82,72 ± 0,017

Table 5.4 presents the image similarity metrics computed between the predictions and the ground truth of the original U-Net architecture, the best evolved U-Net, and the mean metrics for all evolved networks, over the test set. Our approach took, on average, 48 hours per run, considering the two datasets, running on a cluster equipped with Intel Xeon CPU processors, with 96 GB RAM, and no GPU. For the BSDS500 dataset, the Specificity and Sensitivity had a greater impact on the weighted average, while the Dice and mainly the Jaccard index were more difficult to satisfy, not responding to the changes in the image as easily as the other metrics. According to the weighted average, the best-evolved network is presented as slightly better than the original U-Net. Moreover, for the ISBI12 dataset, the Jaccard and Dice indexes had a more significant influence than the BSDS500. Once again, the best-evolved network achieved a higher weighted average than the U-Net. For both datasets, the Dice index was the metric that differed the most, meaning that this metric was more valuable for the ISBI12 dataset (segmentation problem) rather than the BSDS500 (edge detection). The best-evolved network is presented as a better solution than the original U-Net, although the mean among all evolved networks presented a worse performance compared to the original U-Net. The one design that achieved the best performance is also a smaller, less complex, and faster network to train, with 29,685 parameters, compared to the 31,032,837 from the U-Net.

[2]http://brainiac2.mit.edu/isbi_challenge/home

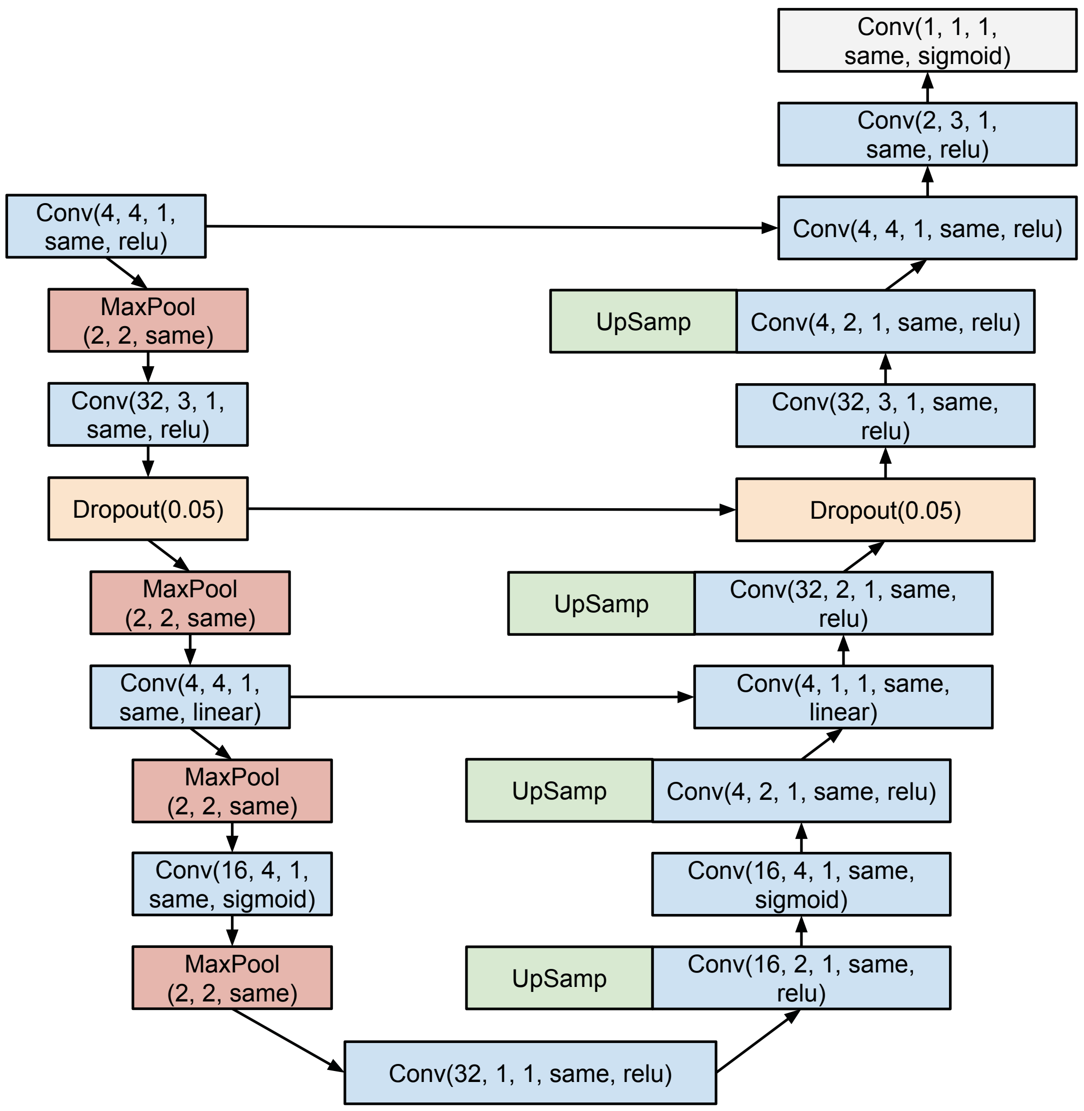

Conv(1, 1, 1, same, sigmoid)
Conv(2, 3, 1, same, relu)
Conv(4, 4, 1, same, relu)
Conv(4, 4, 1, same, relu)
MaxPool (2, 2, same)
UpSamp
Conv(4, 2, 1, same, relu)
Conv(32, 3, 1, same, relu)
Conv(32, 3, 1, same, relu)
Dropout(0.05)
Dropout(0.05)
MaxPool (2, 2, same)
UpSamp
Conv(32, 2, 1, same, relu)
Conv(4, 4, 1, same, linear)
Conv(4, 1, 1, same, linear)
MaxPool (2, 2, same)
UpSamp
Conv(4, 2, 1, same, relu)
Conv(16, 4, 1, same, sigmoid)
Conv(16, 4, 1, same, sigmoid)
MaxPool (2, 2, same)
UpSamp
Conv(16, 2, 1, same, relu)
Conv(32, 1, 1, same, relu)

Figure 5.6: Best U-Net architecture designed by the GE/DNN approach.

The best desing found by GE/DNN is presented in Figure 5.6. Its architecture contains three “bridge connections”, compared to the four connections in the original U-Net. Moreover, it includes a dropout connection, that was placed as the layer to be copied to the expanding path, which is unusual in these networks.

5.4 DESIGN OF NETWORKS FOR TEXT CLASSIFICATION

This application presents an investigation on the design of GNN architectures using GE/DNN, applied to text classification tasks. Text classification is a classic problem in natural language processing (NLP), where the main objective is to label textual elements, such as phrases, queries, paragraphs, and documents. In general, text classification tasks include sentiment analysis (analyzing the opinions of people in textual data), news categorization (to obtain information of interest in real-time), and topic classification (to identify the theme or topics of a text) [76].

In this context, neural network approaches have been used in several of these tasks. Recurring Neural Networks (RNN), for example, is applied to capture word dependencies. A Convolutional Neural Network (CNN) learns how to recognize patterns in the text, highlighting unique or discriminating features, making this type of architecture quite popular for text classification. Some approaches have combined RNN and LSTM as in [7], and RNN and CNN as in [8].

Recently, from the representation of texts through graphs, some text classification models based on convolution in graphs (GNN) have been proposed [110]. In these approaches, the text is represented by a graph G(V, E), where V is the set of nodes that usually represent the text units (words, terms, collocations) or documents; and E is the set of edges that represent the relationships between words, or between words and documents. These relationships vary between models, i.e., they can represent lexical or semantic relations, contextual overlap, or a combination of all of them. Other studies used graph convolutional networks (GCNs) for NLP tasks: [71] used GCN to incorporate syntactic information in neural models; [9] proposed a model for graph-to-sequence learning that promoted recent advances in neural encoder-decoder architectures; and [17] proposed a modification for the GCN architecture by introducing a bidirectional mechanism for convolution directed graph, leading to an improved F1 score in the used dataset.

There are different perspectives of convolution on graphs present in the literature. These approaches differ mainly in the propagation function. Table 5.5 shows the convolution methods used in this work.

Table 5.5: Graph convolution layer methods.

Name	**Description**	**Author(s)**
GraphConv	Uses adjacency matrix with added self-loops and a degree matrix to propagate discriminating nodes	Kipf and Welling [47]
GraphSageConv	Uses a function to aggregate the neighborhood of a node	Hamilton et al. [36]
ARMAConv	Graph convolutional layer based on a rational graph filter	Bianchi et al. [11]
GraphAttention	This layer computes a convolution similar to GraphConv. However, it uses the attention mechanism to weigh the adjacency matrix instead of using the normalized Laplacian	Velickovic et al. [106]
APPNP	A graph convolutional layer implementing the APPNP operator	Klicpera et al. [48]

Furthermore, the advent of BERT [23] created new perspectives for text classification. BERT was designed to pre-train deep bidirectional representations from an unlabeled text by jointly conditioning the left and right context in all layers. “As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications”.

In general, it is essential to note that graph-based convolution models are closely linked to the adjacency matrix (constructed from the graph). Therefore, the edge values present in the adjacency matrix can direct the convolution to distinct niches in the vector space, thus justifying an in-depth analysis of this graph construction aspect.

5.4.1 GNN grammar

We developed a grammar (Figure 5.7) that combines an easy way of defining which components can be used to design the networks with structural rules that guide the designs towards valid architectures, minimizing invalid models.

Our main objective is to build a grammar capable of combining different convolutional layers applied for GNNs, in a single neural network architecture. It includes the main components of the convolutional graph model: the types of convolutional layers and dropout layers, as well as their parameters: for example, the number of units, activation function, and dropout rate. Each convolutional layer presented in Table 5.5 can extract specific features from the input set, highlighting different portions of the graph. Combining these factors could bring a performance gain when evaluated in text classification. We applied restrictions to the grammar in order to decrease the creation of invalid or inefficient architectures. The rules are the following:

- The model will have at least one graph convolution and dense layer;
- The last layer will always be a graph convolution node with ‘softmax’ activation and the number of units set as the number of classes of the dataset;

⟨*gnn*⟩ ::= ⟨*conv*⟩ ⟨*layer_c*⟩ ⟨*dense*⟩

⟨*layer_c*⟩ ::= ⟨*conv*⟩ | ⟨*dropout*⟩ | ⟨*layer_c*⟩ ⟨*layer_c*⟩

⟨*dense*⟩ ::= conv ⟨*conv_type*⟩ ⟨*units*⟩ softmax

⟨*conv*⟩ ::= conv ⟨*conv_type*⟩ ⟨*units*⟩ ⟨*activation*⟩

⟨*conv_type*⟩ ::= GraphConv | GraphSageConv | ARMAConv
  | GraphAttention | APPNP

⟨*dropout*⟩ ::= dropout ⟨*rate*⟩

⟨*units*⟩ ::= 32 | 64 | 128 | 256

⟨*activation*⟩ ::= relu | sigmoid

⟨*rate*⟩ ::= [0.0, 0.25, 0.5]

Figure 5.7: GNN grammar. It includes five different types of convolutional layers for GNNs.

5.4.2 Experiments

Table 5.6 presents the configuration of the evolutionary search. We perform 10 independent runs, with a population of 20 solutions, for 25 generations. Moreover, genetic operators include tournament selection, our Half and Half recombination (described in Section 4.7), with one-point crossover and non-terminal mutation and replacement with elitism. During the evolution, each model is trained using the Adam optimizer [46], with default parameters, for 50 epochs, and the fitness is calculated using the validation set. In the end, the best design found is re-trained with a learning rate of 0.001 and weight decay of $5 * 10^{-4}$, for 500 epochs, similar to the CNN experiment in Section 5.2.

Table 5.6: Parameters of the evolutionary search of GNNs.

Parameter	Value	Parameter	Value
Population size	20	Generations	25
One-point Crossover	60%	Non-Term. Mutation	40%
Tournament size	5	# of Elites	25%
Model Epochs	50/500	# of Runs	10

We run experiments on four widely-used text classification benchmarks: R8, R52[3], Ohsumed[4], and Movie Review (MR)[5]. Our approach has two stages: building a representation of texts and designing a neural network adapted to the domain. The texts of our corpus follow the flow presented in [110], except for the representation of the sentences. Originally, sentences were represented by vectors of zeros. We use Sentence-Bert [86] in order to generate a vector representation for sentences[6]. The models from Sentence-Bert are based on transformer networks such as BERT [23], RoBERTa [62] and XLM-RoBERTa[20]. In general, BERT-based approaches generate vector representations in vector space such that similar text is close and can be efficiently found using cosine similarity. There are two main reasons to use it: 1) BERT is pre-trained on a large amount of data, and in this way it is possible to find semantic relationships in a large vocabulary; 2) BERT can account for a word's context, i.e., it returns different vectors for the same word depending on the words around it, and this feature has fundamental relevance for contextual analysis.

Table 5.7 summarizes the descriptive statistics for each dataset. The number of words is potentially different from the number of nodes due to two main reasons: 1) in the study model, each document is also represented by a node. 2) Not all words present in the dataset will become a node in the representation, mainly due to low frequency.

Table 5.7: Descriptive statistics of each dataset.

Dataset	Doc	Words	Nodes	Edges	Classes
R8	7,674	6,792	14,466	4,024,898	8
R52	9,100	7,890	16,990	5,118,159	52
Ohsumed	7,400	13,323	20,723	8,384,734	23
MR	10,662	20,473	31,135	1,665,491	2

In Table 5.8, we present the results of the GNNs designed by our methods, comparing them with other GNN architectures. The architecture of each of these GNN was provided from the best practices in the literature. Highlighting the experiment over the dataset MR, our approach reaches the highest accuracy values among the presented methods. As in the CNN experiment, when executed under the same conditions as our best networks, the GE/DNN networks were generally able to achieve higher accuracy values. In many cases, GE/DNN generates quite

[3]https://www.cs.umb.edu/ smimarog/ textmining/datasets/

[4]ftp://medir.ohsu.edu/pub/ohsumed

[5]http://www.cs.cornell.edu/people/pabo/movie-review-data/

[6]For this task, we use the pre-trained model paraphrase-MiniLM-L12-v2 available at: https://www.sbert.net/docs/pretrained_models.html

complex networks. However, it was possible to observe the construction of networks with different convolution layers due to the proposed grammar.

We highlight the architecture of our best GNN for the MR dataset in Figure 5.8, which applied different graph layers in a very compact architecture. However, our best solution is a network that took 13,5 hours, on average, to be found. In contrast, the second best approach (Sentence-Bert + GraphAttention) took about 8 hours. All experiments were executed on a Google Cloud Compute Engine instance, with 8vCPU, 32gb RAM and no GPU.

Table 5.8: Classification accuracy and loss of different models on the four datasets, with average accuracy and standard deviation.

Sentence-Bert + Model	Accuracy	Loss	Mean (StdDev)
MR			
GraphConv	72,09	1,86	72,00 ± 0,01
GraphSageConv	62,51	1,82	62,10 ± 0,02
ARMAConv	60,40	1,78	60,31 ± 0,01
GraphAttention	74,00	1,78	73,20 ± 0,17
APPNP	50,99	1,79	50,48 ± 0,02
GE/DNN	75,03	1,81	74,08 ± 0,01
R8			
GraphConv	84,00	1,60	83,90 ± 0,01
GraphSageConv	80,50	1,59	80,44 ± 0,01
ARMAConv	80,00	1,75	79,93 ± 0,03
GraphAttention	85,02	1,71	84,33 ± 0,13
APPNP	66,80	1,49	66,30 ± 0,03
GE/DNN	87,03	1,62	85,00 ± 0,01
R52			
GraphConv	81,05	1,84	81,00 ± 0,03
GraphSageConv	68,96	1,83	67,18 ± 0,47
ARMAConv	61,20	1,85	60,90 ± 0,40
GraphAttention	84,00	1,92	82,09 ± 0,77
APPNP	60,50	1,85	59,98 ± 0,19
GE/DNN	89,01	1,86	87,09 ± 0,19
Ohsumed			
GraphConv	55,03	1,87	55,00 ± 0,02
GraphSageConv	45,00	1,99	43,64 ± 0,59
ARMAConv	46,70	1,99	45,66 ± 0,42
GraphAttention	58,00	1,99	55,03 ± 0,89
APPNP	51,07	1,98	50,42 ± 0,17
GE/DNN	64,99	1,98	63,00 ± 0,21

5.5 CONCLUDING REMARKS

On the first application, it was tested basic features of the proposed approach. The grammar defined a number of layers and hyperparameters as building blocks for CNNs, and also added a custom layers, the residual layer, as another option, which helped designing better architectures. For the second application, a different grammar and build strategy were necessary. We assumed

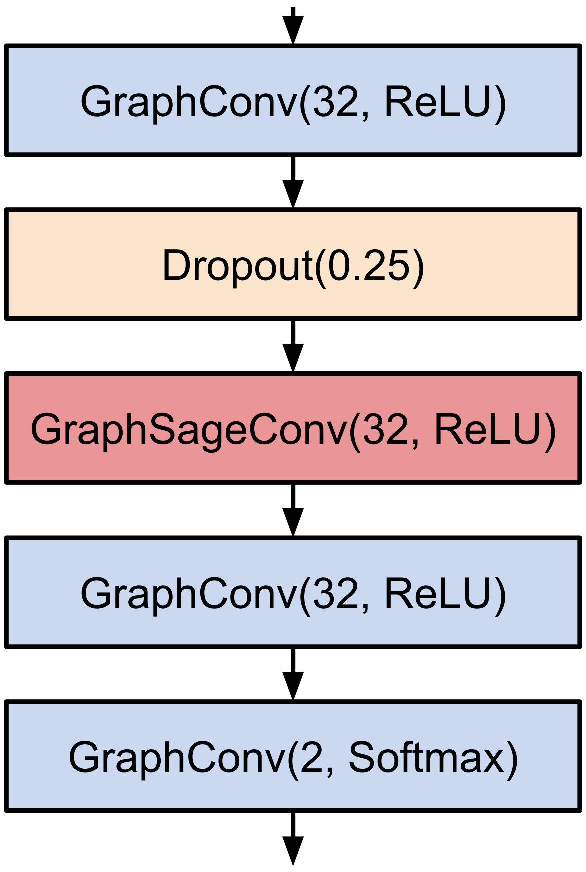


Figure 5.8: Best GNN architecture designed by the GE/DNN approach for the MR dataset.

the U-Net as a symmetric network, from which the grammar was used to define half of the architecture, while the other half was inferred. It showed that our approach is capable to handle more complex architectures, when employing the correct grammar and mapping process. Designing GNNs takes a similar approach to what was already shown in the first application, however, this time applied to a complete different domain, which are networks for graphs. This time, our approach was presented as a flexible tool, that required only the definition of the layers, adapted to graphs, and a building process that could handle the multi input/output required for text problems.

Revisiting the research questions: RQ1 asked if the approach is easily adapted to the different applications. As demonstrated, three different types of networks, with different requirements and objectives were successfully addressed by only defining a grammar and the building process. Different grammars were proposed for each application, with different layers, hyperparameters, and architecture choices. For some of the applications, restrictions were added to the grammar in order to prevent the generation of invalid models, however, even though it might decrease the space of possible designs, the approach did not faced diversity problems. Moreover, the fitness is also flexible, as each problem might require different metrics in order to properly evaluate the quality of the solutions.

RQ2 asks whether the proposed approach is comparable to other neuroevolution techniques. The first application performs a comparison between our approach and CGP, another GP-based approach that uses graphs, showing that the networks generated by GE/DNN were able to outperform the CGP networks, and thus, showing the potential of our approach. To provide a fair comparison, we used the methodology proposed by works we compare to, comparing our best models with theirs. It included how we trained our models during and after the evolutionary process, and also the optimizer used on the best architecture. Similarly, in RQ3 the question extends to manually designed networks. This scenario was covered in both the second and third applications, where our designed networks were tested against manually designed architectures, achieving higher results on the proposed experiments. The results obtained here reinforces the hypothesis, that the automatic design of DNNs is able to generate networks that are able to surpass manually design networks, and in some cases, through unusual choices for the layers and hyperparameters.

Lastly, RQ4 asks about the novelty of this approach. Despite the fact that grammar-based approaches are under explored, compared to other approaches, it is worth mentioning all the work that has been put into exploring the different grammar strategies, along with the building process, for each application. Furthermore, the fine-tuning performed on the search engine, with the proposal of more suitable recombination operators, and the movement acceptance, all contributed to navigate the search space for better designs. More than better accuracy results, the novel aspect presented by our approach is presented by the combination of classical topics on the design of DNNs, proposing new ways to use grammars to define and guide how the architectures are built. The idea is reinforced considering the three different applications proposed, showing the generality of the approach. This is shown through the three proposed applications

6 FINAL REMARKS

DL is a prevalent topic, with extensive studies being developed for a variety of tasks. State-of-the-art architectures are being continuously modified and fine-tuned by experts, increasing the complexity and number of hyperparameters to a point where it takes a significant effort to make further progress. This challenge is part of the motivation for the growing interest in applying techniques to help design DNNs. Neuroevolution and other NAS techniques have been studied and applied with success to the design of architectures, showing the potential, for instance, of evolutionary-based approaches, in generating and recombining solutions as a promising way to improve the quality of the networks.

6.1 CONTRIBUTIONS AND LIMITATIONS

In this work, we proposed an approach called GE/DNN, for the design of DNNs, which combines existing techniques used for the design of algorithms, with novel mechanisms adapted to design DNNs. We described how a grammar could be used to define layers and hyperparameters as building blocks for DNN architectures. Explored the indirect encoding and mapping process, used in DSGE, with the well-known evolutionary schema for generating, recombining and evaluating solutions. GE/DNN was designed to be a flexible tool, which can be applied to various machine learning problems, requiring the user to only define the grammar and how the networks are built. On the other hand, evaluating these designs have high computational cost, regardless of an optimized set of parameters, balancing the amount of evolved networks. We evaluated our approach in three different applications, each on the design of networks for different domains. The first application focused on the design of CNNs for image classification tasks, proposing a grammar with the most common layers used for CNNs, and a residual block composed of a particular set of layers. Our designed networks were able to outperform another GP-based evolutionary approach when executing the training under the same conditions. The second application involved the design of U-Nets for image segmentation problems. The particularities of the U-Net were used to propose a symmetric grammar that built networks by only designing the left side and inferring the right side. GE/DNN was tested on two real-world datasets, the BSDS500 for edge detection and the ISBI12 for image segmentation, where we were able to obtain an architecture that outperformed the original U-Net, in both datasets. The third application focused on the design of GNNs for text classification tasks. Sentence-Bert was used to generate the representation for the sentences, and state-of-the-art modules were used to compose part of the proposed grammar for this application. The evolved networks were then compared to these state-of-the-art modules individually, presenting a better result when combined by our approach than in the original architectures.

The following list presents the main contributions of GE/DNN.

- Proposed a grammar strategy for DNNs that combines rules for layers, hyperparameters and architectures;
- Created new recombination operators that explores the improved encoding from DSGE;
- Applied the approach to different scenarios to show the potential as a flexible tool;
- Demonstrated that it is possible to automatically generate architectures competitive to other automated techniques.

Similarly, the following list presents some of the existing limitations of the approach:

- More complex architectures will require further exploration of the proposed grammar;
- Each different scenario also requires a evaluation strategy that is suitable to the type of network;
- Currently, the approach only generates the architecture, not including tuning the weights;
- In general, the whole evolutionary process consumes a lot of computational resources.

6.2 FUTURE DIRECTIONS

By using only a fraction of the dataset when designing the CNNs, we significantly reduced the time and computational resources involved while also finding high-performing and also compact architectures. More experiments with similar approaches might help reducing costs related to training the networks. For instance, similarly to the hypothesis that was empirically tested, where the models were evaluated with no training. Even though the results were not positive, it can be faced as one among many possibilities, that should be more explored. Moreover, we explored two types of architectures, sequential and U-shaped, and a variety of layer types, for three different problems. Each application tested different features of our approach, such as representation power, adaptability and flexibility. Testing the approach for other architectures and domains brings value and useful insights that can help expanding the existing approach. Extended work can be performed on the search engine, proposing suitable operators for specific situations, similar to what was presented with our Half and Half mechanism, that proved to be more useful than regular crossover and mutation. Considering different criteria for selecting the architectures could be implemented as a way to explore multi-objective search spaces, such as novelty along with quality. Furthermore, exploring mixing the "design from scratch", with other perspectives, for instance, evolving parts of the network separately, taking inspiration from transfer learning works, similar to what was explored in the U-Net grammar.

6.3 PUBLICATIONS

The following list contains the works that were generated throughout this research, with paper published in well-known, national and international, congresses of the evolutionary computation field.

- [58] Ricardo H. R. Lima, Aurora T. R. Pozo. *Evolving convolutional neural networks through grammatical evolution*. Genetic and Evolutionary Computation Conference (GECCO) 2019.
- [60] Ricardo H. R. Lima, Aurora T. R. Pozo. *Automatic Design of Convolutional Neural Networks using Grammatical Evolution*. Brazilian Conference on Intelligent Systems (BRACIS) 2019.
- [59] Ricardo H. R. Lima, Aurora T. R. Pozo, Alexander Mendiburu, Roberto Santana. *A Symmetric grammar approach for designing segmentation models*. IEEE Congress on Evolutionary Computation (CEC) 2020
- [56] Ricardo H. R. Lima, Aurora T. R. Pozo, Alexander Mendiburu, Roberto Santana. *Automatic Design of Deep Neural Networks Applied to Image Segmentation Problems*. European Conference on Genetic Programming (EuroGP) 2021

- Ricardo H. R. Lima, Dimmy K. Magalhães, Aurora T. R. Pozo, Alexander Mendiburu, Roberto Santana. *A Grammar-based GP approach applied to the design of deep neural networks*. Journal of Genetic Programming and Evolvable Machines 2022. Accepted for publication.

REFERENCES

[1] Shadi Al-Zu'bi, Bilal Hawashin, Ala Mughaid, and Thar Baker. Efficient 3D medical image segmentation algorithm over a secured multimedia network. *Multimedia Tools and Applications*, pages 1–19, 2020.

[2] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour detection and hierarchical image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(5):898–916, may 2011.

[3] Filipe Assunçao, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. Towards the evolution of multi-layered neural networks: A dynamic structured grammatical evolution approach. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 393–400. ACM, 2017.

[4] Filipe Assunçao, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. DENSER: Deep evolutionary network structured representation. *Genetic Programming and Evolvable Machines*, 20(1):5–35, 2019.

[5] Thomas Back and H-P Schwefel. Evolutionary computation: An overview. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 20–29. IEEE, 1996.

[6] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[7] Anupam Baliyan, Akshit Batra, and Sunil Pratap Singh. Multilingual sentiment analysis using RNN-LSTM and neural machine translation. In *2021 8th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 710–713. IEEE, 2021.

[8] Mohammad Ehsan Basiri, Shahla Nemati, Moloud Abdar, Erik Cambria, and U Rajendra Acharya. Abcdm: An attention-based bidirectional CNN-RNN deep model for sentiment analysis. *Future Generation Computer Systems*, 115:279–294, 2021.

[9] Daniel Beck, Gholamreza Haffari, and Trevor Cohn. Graph-to-sequence learning using gated graph neural networks. *CoRR*, abs/1806.09835, 2018.

[10] Song Bian, Xiaowei Xu, Weiwen Jiang, Yiyu Shi, and Takashi Sato. BUNET: Blind Medical Image Segmentation Based on Secure UNET. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 612–622. Springer, 2020.

[11] Filippo Maria Bianchi, Daniele Grattarola, Lorenzo Livi, and Cesare Alippi. Graph neural networks with convolutional ARMA filters. *CoRR*, abs/1901.01343, 2019.

[12] Chris M Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832, 1994.

[13] Léon Bottou et al. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8):12, 1991.

[14] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

[15] Albert Cardona, Stephan Saalfeld, Stephan Preibisch, Benjamin Schmid, Anchi Cheng, Jim Pulokas, Pavel Tomancak, and Volker Hartenstein. An integrated micro-and macroarchitectural analysis of the drosophila brain by computer-assisted serial section electron microscopy. *PLoS biology*, 8(10):e1000502, 2010.

[16] Leandro N Castro, Fernando J Von Zuben, and Helder Knidel. *Artificial immune systems*. Springer, 2007.

[17] Alberto Cetoli, Stefano Bragaglia, Andrew D. O'Harney, and Marc Sloan. Graph convolutional networks for named entity recognition. *CoRR*, abs/1709.10053, 2017.

[18] Roberto Cipolla, Sebastiano Battiato, Giovanni Maria Farinella, et al. *Machine learning for computer vision*, volume 5. Springer, 2013.

[19] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, 2008.

[20] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale. *CoRR*, 2019.

[21] Javier Del Ser, Eneko Osaba, Daniel Molina, Xin-She Yang, Sancho Salcedo-Sanz, David Camacho, Swagatam Das, Ponnuthurai N Suganthan, Carlos A Coello Coello, and Francisco Herrera. Bio-inspired computation: Where we stand and what's next. *Swarm and Evolutionary Computation*, 48:220–250, 2019.

[22] Li Deng and Xiao Li. Machine learning paradigms for speech recognition: An overview. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(5):1060–1089, 2013.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[24] Shifei Ding, Chunyang Su, and Junzhao Yu. An optimizing BP neural network algorithm based on genetic algorithm. *Artificial Intelligence Review*, 36(2):153–162, 2011.

[25] Alexey Dosovitskiy, Jost Tobias Springenberg, Martin Riedmiller, and Thomas Brox. Discriminative unsupervised feature learning with convolutional neural networks. *Advances in neural information processing systems*, 27, 2014.

[26] Michal Drozdzal, Eugene Vorontsov, Gabriel Chartrand, Samuel Kadoury, and Chris Pal. The Importance of Skip Connections in Biomedical Image Segmentation. In *Deep Learning and Data Labeling for Medical Applications*, pages 179–187. Springer International Publishing, 2016.

[27] Russell C Eberhart, Yuhui Shi, and James Kennedy. *Swarm intelligence*. Elsevier, 2001.

[28] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[29] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

[30] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20:1–21, 2019.

[31] Bradley J Erickson, Panagiotis Korfiatis, Zeynettin Akkus, and Timothy L Kline. Machine learning for medical imaging. *Radiographics*, 37(2):505–515, 2017.

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

[33] David B Fogel, Lawrence J Fogel, and VW Porto. Evolving neural networks. *Biological Cybernetics*, 63(6):487–493, 1990.

[34] Edgar Galván and Peter Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. *IEEE Transactions on Artificial Intelligence*, 2021.

[35] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*, pages 6645–6649. Ieee, 2013.

[36] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, pages 770–778, 2016.

[38] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[39] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[40] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[41] Junyan Hu, Hanlin Niu, Joaquin Carrasco, Barry Lennox, and Farshad Arvin. Voronoi-based multi-robot autonomous exploration in unknown environments via deep reinforcement learning. *IEEE Transactions on Vehicular Technology*, 69(12):14413–14423, 2020.

[42] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[43] Alois Huning. Evolutionsstrategie. optimierung technischer systeme nach prinzipien der biologischen evolution, 1976.

[44] Renlong Jie and Junbin Gao. Differentiable neural architecture search for high-dimensional time series forecasting. *IEEE Access*, 9:20922–20932, 2021.

[45] Wolfgang Kantschik and Wolfgang Banzhaf. Linear-graph gp - a new gp structure. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea Tettamanzi, editors, *Genetic Programming*, pages 83–92, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

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

[47] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[48] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Personalized embedding propagation: Combining neural networks on graphs with personalized pagerank. *CoRR*, abs/1810.05997, 2018.

[49] John R Koza, Forrest H Bennett, David Andre, and Martin A Keane. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Artificial Intelligence in Design'96*, pages 151–170. Springer, 1996.

[50] John R Koza and James P Rice. Automatic programming of robots using genetic programming. In *AAAI*, volume 92, pages 194–207. Citeseer, 1992.

[51] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[52] Thomas Kurbiel and Shahrzad Khaleghian. Training of deep neural networks based on distance measures using rmsprop. *arXiv preprint arXiv:1708.01911*, 2017.

[53] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

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

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

[56] Ricardo H. R. Lima, Aurora T. R. Pozo, Alexander Mendiburu, and Roberto Santana. Automatic design of deep neural networks applied to image segmentation problems. In *Genetic Programming - 24th European Conference, EuroGP 2021, Held as Part of EvoStar 2021, Virtual Event, April 7-9, 2021, Proceedings*, volume 12691 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2021.

[57] Ricardo Henrique Remes Lima and Aurora Trinidad Ramirez Pozo. A study on auto-configuration of multi-objective particle swarm optimization algorithm. In *Proceedings of the 2017 IEEE Congress on Evolutionary Computation (CEC'17)*, pages 718–725. IEEE, 2017.

[58] Ricardo Henrique Remes Lima and Aurora Trinidad Ramirez Pozo. Evolving convolutional neural networks through grammatical evolution. In *Proceedings of the 2019 Genetic and Evolutionary Computation Conference (GECCO'19)*, pages 179–180. ACM, 2019.

[59] Ricardo Henrique Remes Lima, Aurora Trinidad Ramirez Pozo, Alexander Mendiburu, and Roberto Santana. A Symmetric grammar approach for designing segmentation models. In *Proceedings of the 2020 IEEE Congress on Evolutionary Computation (CEC'20)*, pages 1–8. IEEE, 2020.

[60] Ricardo Henrique Remes Lima, Aurora Trinidad Ramirez Pozo, and Roberto Santana. Automatic design of convolutional neural networks using grammatical evolution. In *Proceedings of the 8th Brazilian Conference on Intelligent Systems, BRACIS 2019, Salvador, Brazil, October 15-18, 2019*, pages 329–334. IEEE, 2019.

[61] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34, 2018.

[62] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv*, 2019.

[63] Trevor Londt, Xiaoying Gao, and Peter Andreae. Evolving character-level densenet architectures using genetic programming. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 665–680. Springer, 2021.

[64] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[65] Manuel Lopez-Ibanez and Thomas Stutzle. The automatic design of multiobjective ant colony optimization algorithms. *IEEE Transactions on Evolutionary Computation*, 16(6):861–875, 2012.

[66] Pablo Ribalta Lorenzo and Jakub Nalepa. Memetic evolution of deep neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 505–512, 2018.

[67] Nuno Lourenço, Filipe Assunção, Francisco B Pereira, Ernesto Costa, and Penousal Machado. *Structured Grammatical Evolution: A Dynamic Approach*. Springer, 2018.

[68] Nuno Lourenço, Francisco Pereira, and Ernesto Costa. Evolving evolutionary algorithms. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO'12)*, pages 51–58. ACM, 2012.

[69] Nuno Lourenço, Francisco B Pereira, and Ernesto Costa. Unveiling the properties of structured grammatical evolution. *Genetic Programming and Evolvable Machines*, 17(3):251–289, 2016.

[70] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/~sean/book/metaheuristics/.

[71] Diego Marcheggiani and Ivan Titov. Encoding sentences with graph convolutional networks for semantic role labeling. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural*

Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017, pages 1506–1515. Association for Computational Linguistics, 2017.

[72] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the age of neural networks and brain computing*, pages 293–312. Elsevier, 2019.

[73] Geoffrey F Miller, Peter M Todd, and Shailesh U Hegde. Designing Neural Networks using Genetic Algorithms. In *Proceedings of the International Conference on Genetic Algorithms (ICGA'89)*, volume 89, pages 379–384, 1989.

[74] Julian F Miller et al. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proceedings of the genetic and evolutionary computation conference*, volume 2, pages 1135–1142, 1999.

[75] F. Milletari, N. Navab, and S. Ahmadi. V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation. In *2016 Fourth International Conference on 3D Vision (3DV'16)*, pages 565–571, 2016.

[76] Shervin Minaee, Nal Kalchbrenner, Erik Cambria, Narjes Nikzad, Meysam Chenaghlu, and Jianfeng Gao. Deep learning–based text classification: A comprehensive review. *ACM Computing Surveys (CSUR)*, 54(3):1–40, 2021.

[77] P Mirunalini, C Aravindan, A Thamizh Nambi, S Poorvaja, and V Pooja Priya. Segmentation of coronary arteries from CTA axial slices using deep learning techniques. In *Proceedings of the 2019 IEEE Region 10 Conference (TENCON'19)*, pages 2074–2080. IEEE, 2019.

[78] Tom Mitchell. Machine learning. 1997.

[79] MS Nidhya, L Jayanthi, RM Sekar, J Jeyabharathi, and Mrs Poonam. Analysis of machine learning algorithms for spam filtering. *Annals of the Romanian Society for Cell Biology*, pages 3469–3476, 2021.

[80] O. Oktay, E. Ferrante, K. Kamnitsas, M. Heinrich, W. Bai, J. Caballero, S. A. Cook, A. de Marvao, T. Dawes, D. P. O'Regan, B. Kainz, B. Glocker, and D. Rueckert. Anatomically constrained neural networks (ACNNs): Application to cardiac image enhancement and segmentation. *IEEE Transactions on Medical Imaging*, 37(2):384–395, 2018.

[81] Michael O'Neill and Conor Ryan. Grammatical Evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[82] T. Perkis. Stack-based genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation (CEC). IEEE World Congress on Computational Intelligence (WCCI)*, volume 1, pages 148–153, 1994.

[83] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *CoRR*, 2017.

[84] Riccardo Poli and John Koza. *Genetic programming*. Springer, 2014.

[85] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Aging evolution for image classifier architecture search. In *AAAI conference on artificial intelligence*, volume 2, page 2, 2019.

[86] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[87] Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(1):25–41, 2017.

[88] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[89] Stuart Russell and Peter Norvig. Artificial intelligence: A modern approach, global edition 4th. *Foundations*, 19:23, 2021.

[90] Conor Ryan, John James Collins, and Michael O'Neill. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *Proceedings of the European Conference on Genetic Programming (EuroGP'98)*, pages 83–96. Springer, 1998.

[91] D Sabarinathan, M Parisa Beham, SM Roomi, et al. Hyper vision net: Kidney tumor segmentation using coordinate convolutional layer and attention unit. *arXiv preprint arXiv:1908.03339*, 2019.

[92] Abhinav Sagar and RajKumar Soundrapandiyan. Semantic segmentation with multi scale spatial attention for self driving cars. *arXiv preprint arXiv:2007.12685*, 2020.

[93] G. Sainath, S. Vignesh, S. Siddarth, and G. Suganya. Application of neuroevolution in autonomous cars. In *International Virtual Conference on Industry 4.0*, pages 301–311. Springer Singapore, 2021.

[94] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[95] Linda G Shapiro and George C Stockman. *Computer Vision*. New Jersey, Prentice-Hall, 2001.

[96] M Shinozuka and B Mansouri. Synthetic aperture radar and remote sensing technologies for structural health monitoring of civil infrastructure systems. In *Structural health monitoring of civil infrastructure systems*, pages 113–151. Elsevier, 2009.

[97] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1:24–35, 2019.

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

[99] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[100] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A Genetic Programming Approach to Designing Convolutional Neural Network Architectures. In *Proceedings of the 2017 Genetic and Evolutionary Computation Conference (GECCO'17)*, pages 497–504. ACM, 2017.

[101] Yanan Sun, Bing Xue, Mengjie Zhang, and Gary G Yen. Evolving Deep Convolutional Neural Networks for Image Classification. *IEEE Transactions on Evolutionary Computation*, 24(2):394–407, 2019.

[102] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[103] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15)*, June 2015.

[104] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[105] El-Ghazali Talbi. Automated design of deep neural networks: A survey and unified taxonomy. *ACM Computing Surveys (CSUR)*, 54(2):1–37, 2021.

[106] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[107] G. Wang, W. Li, M. A. Zuluaga, R. Pratt, P. A. Patel, M. Aertsen, T. Doel, A. L. David, J. Deprest, S. Ourselin, and T. Vercauteren. Interactive medical image segmentation using deep learning with image-specific fine tuning. *IEEE Transactions on Medical Imaging*, 37(7):1562–1573, 2018.

[108] Lorenz Wendlinger, Julian Stier, and Michael Granitzer. Evofficient: Reproducing a Cartesian Genetic Programming Method. In *European Conference on Genetic Programming (Part of EvoStar)*, pages 162–178. Springer, 2021.

[109] Darrell Whitley, Timothy Starkweather, and Christopher Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14(3):347–361, 1990.

[110] Liang Yao, Chengsheng Mao, and Yuan Luo. Graph convolutional networks for text classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7370–7377, 2019.

[111] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306*, 2018.

[112] Arber Zela, Aaron Klein, Stefan Falkner, and Frank Hutter. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *arXiv preprint arXiv:1807.06906*, 2018.

[113] Xun Zhou, AK Qin, Maoguo Gong, and Kay Chen Tan. A survey on evolutionary construction of deep neural networks. *IEEE Transactions on Evolutionary Computation*, 25(5):894–912, 2021.

[114] Marc-André Zöller and Marco F Huber. Benchmark and survey of automated machine learning frameworks. *Journal of artificial intelligence research*, 70:409–472, 2021.

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

[116] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.