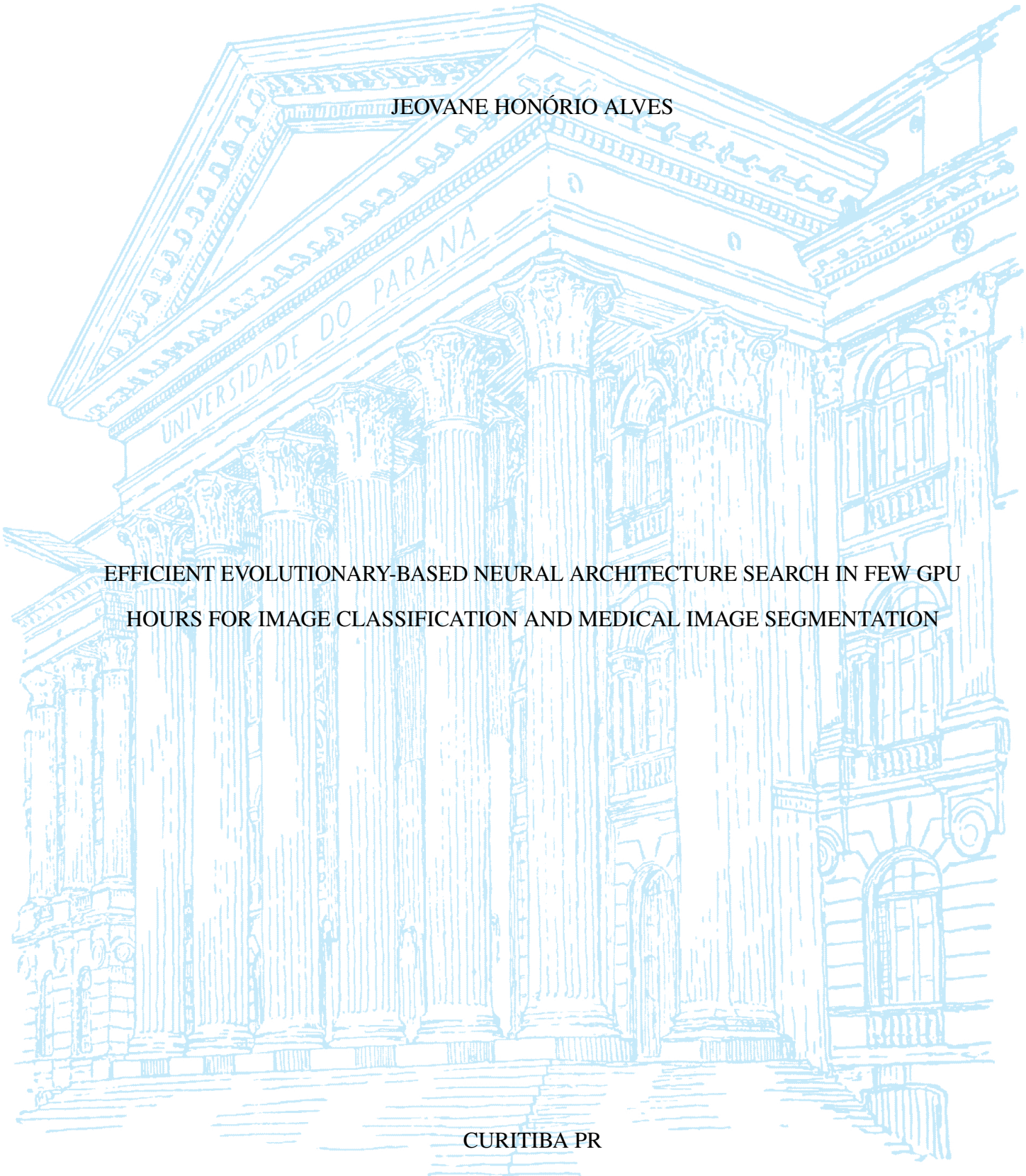
UNIVERSIDADE FEDERAL DO PARANÁ

JEOVANE HONÓRIO ALVES



EFFICIENT EVOLUTIONARY-BASED NEURAL ARCHITECTURE SEARCH IN FEW GPU HOURS FOR IMAGE CLASSIFICATION AND MEDICAL IMAGE SEGMENTATION

CURITIBA PR

2021

JEOVANE HONÓRIO ALVES

EFFICIENT EVOLUTIONARY-BASED NEURAL ARCHITECTURE SEARCH IN FEW GPU HOURS FOR IMAGE CLASSIFICATION AND MEDICAL IMAGE SEGMENTATION

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: Lucas Ferrari de Oliveira.

CURITIBA PR

2021

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

A474e Alves, Jeovane Honório
Efficient evolutionary-based neural architecture search in few GPU hours for image classification and medical image segmentation [recurso eletrônico] / Jeovane Honório Alves – Curitiba, 2021.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática.

Orientador: Lucas Ferrari de Oliveira

1. Aprendizagem de máquina. 2. Arquitetura de computador. 3. Redes neurais (Computação). I. Universidade Federal do Paraná. II. Oliveira, Lucas Ferrari de. III. Título.

CDD: 006.4

Bibliotecária: Roseny Rivelini Morciani CRB-9/1585

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **JEOVANE HONORIO ALVES** intitulada: **Efficient Evolutionary-based Neural Architecture Search in few GPU hours for Image Classification and Medical Image Segmentation**, sob orientação do Prof. Dr. LUCAS FERRARI DE OLIVEIRA, 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, 20 de Setembro de 2021.

Assinatura Eletrônica
21/09/2021 12:01:35.0
LUCAS FERRARI DE OLIVEIRA
Presidente da Banca Examinadora

Assinatura Eletrônica
21/09/2021 15:49:22.0
LUIZ EDUARDO SOARES DE OLIVEIRA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
21/09/2021 12:48:46.0
MYRIAM REGATTIERI DE BIASE DA SILVA DELGADO
Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

Assinatura Eletrônica
21/09/2021 11:52:11.0
PAULO MAZZONCINI DE AZEVEDO MARQUES
Avaliador Externo (UNIVERSIDADE DE SÃO PAULO)

"There is not a square inch in the whole domain of our human existence over which Christ, who is Sovereign over all, does not cry, Mine!"
– Abraham Kuyper

ACKNOWLEDGEMENTS

Agradeço imensamente a Jesus Cristo, Deus e Senhor sobre todas as coisas, por trazer "vida" a minha vida. Não só esse doutorado não seria concretizado sem Ele, mas também não teria sentido. Ofertar este trabalho (mesmo com todas as suas imperfeições) a Ele, para mim, é demasiadamente honroso.

Quero agradecer aos meus pais, Julio e Elizabeth, por terem me dado todo o amor e educação que me fez ser o homem que sou hoje. Em especial ao meu pai por ter me dado imenso apoio nesses últimos anos para a concretização desta longa jornada acadêmica. Agradeço aos meus irmãos, Julio e Juliana, por serem meus grandes e eternos amigos. Sem eles, não entenderia o valor do crescimento. Também quero agradecer a minha vó Norbelina, que batalhou muito pela família e sempre perseverou na fé, mesmo com as inúmeras dificuldades da vida. Todo o seu sacrifício gerou diversos bons frutos. Sem ele, certamente não teria chegado onde estou. Esta conquista é de todos vocês: minha abençoada família.

Agradeço também a você, Brunna, por ser a minha melhor amiga e minha maior apoiadora. Seu suporte e cuidado me deram muita força para enfrentar os desafios desta empreitada, principalmente na reta final. Obrigado por me encorajar a ser um cristão melhor!

Agradeço ao meu orientador, prof. Lucas Ferrari de Oliveira, pela confiança e pelo imenso apoio durante todos esses anos de mestrado e doutorado. Agradeço também a banca, composta pelos professores Luiz Eduardo de Oliveira, Myriam Delgado e Paulo Mazzoncini, pelas ricas críticas e sugestões.

Agradeço aos meus amigos do PPGInf pelas risadas, apoios e diversas etapas nessa longa jornada: Ademar Alves, Aline Dartora, André Hochuli, Bruno Krinski, Caroline Cordeiro, Cides Bezerra, Daniel Ruiz, Diego Tomé, Eduardo Almeida, Fabio Spanhol, Ivan Jr., Johanna Rogalsky, Leandro Pulgatti, Leonardo Amaral, Luiz Zanlorensi, Marcelino Abel, Paulo Almeida, Pedro Holanda, Ramiro Lucas, Rayson Laroca, Renato Melo, Teruo Maruyama, Valber Zacarkim e Walmir Couto.

Não posso deixar de mencionar a família dada por Deus, chamada "Jesus na UFPR". Vocês me ensinaram o que é ser "corpo de Cristo" e o valor da amizade. Dentre eles, agradecimentos especiais a Arthur Cordeiro, Bianka Woloszyn, Daniel Cunha, Dieudonne Kabemba, Gleici Perola, Hamer Iboshi, Levi Muniz, Lucas Henrique, Luiz Hammerer, Matheus Augusto, Paola Yasmin, Pedro "Guanandy", Pedro "Kondy", Rafael Damiani, Raphael Teixeira, Silas Nikkel, Thalyta Genaro, Vanessa Jatwa, Vilas Boas e William Watson. E também, fazendo parte dessa família, os amados missionários Kyeongtae Rho (Davi) e Eunjung Park (Sara). Obrigado por todo o carinho e por serem os maiores exemplos, para mim, de como um cristão deve se doar ao próximo.

Agradeço aos meus amigos Carla Fortunato, Elris Aparecida, Francielly Blitzkow, Gabriel Palmas, Guilherme Coelho, Luis Salles Jr., Madger Moraes, Max de Carvalho, Natan Batalha, Renato Celso, Sanna Alves e Vinicius Lacerda, pelo suporte e encorajamento em diversas etapas da minha vida. Obrigado por me darem forças para perseverar!

Por fim, agradeço as inúmeras pessoas que passaram na minha vida e contribuíram, de certa forma, para que eu seja uma pessoa melhor. Se o seu nome não está aqui, certamente é devido a minha péssima memória.

RESUMO

O uso de aprendizagem profunda (AP) está crescendo rapidamente, já que o poder computacional atual fornece otimização e inferência rápidas. Além disso, vários métodos exclusivos de AP estão evoluindo, permitindo resultados superiores em visão computacional, reconhecimento de voz e análise de texto. Os métodos AP extraem característica automaticamente para melhor representação de um problema específico, removendo o árduo trabalho do desenvolvimento de descritores de características dos métodos convencionais. Mesmo que esse processo seja automatizado, a criação inteligente de redes neurais é necessária para o aprendizado adequado da representação, o que requer conhecimento em AP. O campo de busca de arquiteturas neurais (BAN) foca no desenvolvimento de abordagens inteligentes que projetam redes robustas automaticamente para reduzir o conhecimento exigido para o desenvolvimento de redes eficientes. BAN pode fornecer maneiras de descobrir diferentes representações de rede, melhorando o estado da arte em diferentes aplicações. Embora BAN seja relativamente nova, várias abordagens foram desenvolvidas para descobrir modelos robustos. Métodos eficientes baseados em evolução são amplamente populares em BAN, mas seu alto consumo de placa gráfica (de alguns dias a meses) desencoraja o uso prático. No presente trabalho, propomos duas abordagens BAN baseadas na evolução eficiente com baixo custo de processamento, exigindo apenas algumas horas de processamento na placa gráfica (menos de doze em uma RTX 2080Ti) para descobrir modelos competitivos. Nossas abordagens extraem conceitos da programação de expressão gênica para representar e gerar redes baseadas em células robustas combinadas com rápido treinamento de candidatos, compartilhamento de peso e combinações dinâmicas. Além disso, os métodos propostos são empregados em um espaço de busca mais amplo, com mais células representando uma rede única. Nossa hipótese central é que BAN baseado na evolução pode ser usado em uma busca com baixo custo (combinada com uma estratégia robusta e busca eficiente) em diversas tarefas de visão computacional sem perder competitividade. Nossos métodos são avaliados em diferentes problemas para validar nossa hipótese: classificação de imagens e segmentação semântica de imagens médicas. Para tanto, as bases de dados *CIFAR* são estudadas para a tarefa de classificação e o desafio *CHAOS* para a tarefa de segmentação. As menores taxas de erro encontradas nas bases *CIFAR-10* e *CIFAR-100* foram $2,17\% \pm 0,10$ e $15,47\% \pm 0,51$, respectivamente. Quanto às tarefas do desafio *CHAOS*, os valores de *Dice* ficaram entre 90% e 96%. Os resultados obtidos com nossas propostas em ambas as tarefas mostraram a descoberta de redes robustas para ambas as tarefas com baixo custo na fase de busca, sendo competitivas em relação ao estado da arte em ambos os desafios.

Palavras-chave: aprendizagem de máquina automatizado, busca de arquiteturas neurais, aprendizagem profunda, classificação de imagens, segmentação semântica

ABSTRACT

Deep learning (DL) usage is growing fast since current computational power provides fast optimization and inference. Furthermore, several unique DL methods are evolving, enabling superior computer vision, speech recognition, and text analysis results. DL methods automatically extract features to represent a specific problem better, removing the hardworking of feature engineering from conventional methods. Even if this process is automated, intelligent network design is necessary for proper representation learning, which requires expertise in DL. The neural architecture search (NAS) field focuses on developing intelligent approaches that automatically design robust networks to reduce the expertise required for developing efficient networks. NAS may provide ways to discover different network representations, improving the state-of-the-art in different applications. Although NAS is relatively new, several approaches were developed for discovering robust models. Efficient evolutionary-based methods are widely popular in NAS, but their high GPU consumption (from a few days to months) discourages practical use. In the present work, we propose two efficient evolutionary-based NAS approaches with low-GPU cost, requiring only a few GPU hours (less than twelve in an RTX 2080Ti) to discover competitive models. Our approaches extract concepts from gene expression programming to represent and generate robust cell-based networks combined with fast candidate training, weight sharing, and dynamic combinations. Furthermore, the proposed methods are employed in a broader search space, with more cells representing a unique network. Our central hypothesis is that evolutionary-based NAS can be used in a low-cost GPU search (combined with a robust strategy and efficient search) in diverse computer vision tasks without losing competitiveness. Our methods are evaluated in different problems to validate our hypothesis: image classification and medical image semantic segmentation. For this purpose, the CIFAR datasets are studied for the classification task and the CHAOS challenge for the segmentation task. The lowest error rates found in CIFAR-10 and CIFAR-100 datasets were 2.17% ± 0.10 and 15.47% ± 0.51, respectively. As for the CHAOS challenge tasks, the dice scores were between 90% and 96%. The obtained results from our proposal in both tasks shown the discovery of robust networks for both tasks with little GPU cost in the search phase, being competitive to state-of-the-art approaches in both challenges.

Keywords: automated machine learning, neural architecture search, deep learning, image classification, semantic segmentation

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ADF	Automatically Defined Function
AutoML	Automated Machine Learning
BN	Batch Normalization
BO	Bayesian Optimization
CAN	Conditional Adversarial Network
cGAN	Conditional Generative Adversarial Network
CGP	Cartesian Genetic Programming
CNF	Convolutional Neural Fabrics
CNN	Convolutional Neural Network
CSTENAS	Cell Swapping-based Training for Evolutionary-based Neural Architecture Search
CT	Computer Tomography
DAG	Directed Acyclic Graph
DARTS	Differentiable Architecture Search
DE	Differential Evolution
DL	Deep Learning
DSC	Dice similarity SCore
EAS	Efficient Architecture Search
EP	Evolutionary Programming
ES	Evolutionary Strategies
FP	False Positive
FN	False Negative
GA	Genetic Algorithm
GAP	Global Average Pooling
GAS	Gradient-based search using Differentiable Architecture Sampler
GEP	Gene Expression Programming
GP	Genetic Programming
GPU	Graphics Processing Unit
MAdds	Multiplications and Additions
M	Million
ML	Machine Learning
MRI	Magnetic Resonance Imaging
NaN	Not A Number
NAS	Neural Architecture Search
NASGEP	Neural Architecture Search using Gene Expression Programming

NASH	NAS by Hillclimbing
NAT	Neural Architecture Transfer
PLNT	Path-level network transformation
PNAS	Progressive Neural Architecture Search
PSO	Particle Swarm Optimization
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
ROI	Region of Interest
SGAS	Sequential Greedy Architecture Search
SGD	Stochastic Gradient Descent
TP	True Positive
TN	True Negative

LIST OF SYMBOLS

B	Number of blocks in a cell
G	Maximum number of generations to execute in the search phase
k	Number of individuals that participates of a tournament selection
N	Number of cells stacked in a stage
S	Number of individuals to be preserved in a population
s	Stride value of an operation (e.g., convolution, pooling)
W	Initial width (i.e., number of channels) of a model after the convolutional stem
w	Tensor containing the weights of a layer
Z	Number of children to generate in the reproduction step

CONTENTS

1 INTRODUCTION

Employment of new technologies to solve social problems or to automate processes is growing considerably in recent years with new techniques, ideas, and processing power. Processes like Spam detection, individual identification, and product recommendation are present in online services and even personal smartphones. Many approaches used in these services are categorized in the Machine Learning (ML) field, and even in the class of Deep Learning (DL) techniques (LeCun et al., 2015).

In Machine Learning, a sub-field of Computer Science, we develop algorithms that learn from various samples to solve a particular problem. Compact representations are extracted to represent these samples better since these samples may contain too much data to label from the raw data. This small representation is generally called a feature vector. A feature vector can be generated with a hand-crafted technique applied to a raw input (e.g., image, music, text). This feature vector is generally inputted to another technique (e.g., classifier) for identifying/classifying the input (Bishop, 2006). This hand-crafted engineering may impose limitations in representing a particular problem. It is crucial to carefully develop a set of techniques that correctly represents the problem in question, increasing the necessity of expert intervention and time-consuming study (LeCun et al., 2015; Bengio et al., 2021).

The representation learning field enters the context with algorithms that automatically extract possible good representations directly from raw input. Thus, it reduces the requirement to study and craft robust feature descriptors. Deep learning techniques, which are representation learning methods with multiple levels of representation, are being heavily used in recent years. DL has been increasingly applied to different problems concerning image recognition, biometry, machine translation, and signal classification (LeCun et al., 2015; Bengio et al., 2021).,Usage of convolutional neural networks (CNN), for example, covers diverse problems like license plate recognition (Laroca et al., 2021b), medical image segmentation (Alves et al., 2018), face recognition (Silva et al., 2018; Salomon et al., 2020), meter reading (Laroca et al., 2021a), iris recognition (Zanlorensi et al., 2018; Lucio et al., 2019), periocular recognition (Zanlorensi et al., 2020b,a,c, 2021), video surveillance (Luz et al., 2018) and autonomous navigation (Ruiz and Todt, 2021). Although conventional ML can be used to solve problems like these (e.g., (Cordeiro et al., 2018)) or even others like data augmentation (e.g., (Ruiz et al., 2019, 2020)), DL has been effectively used to solve several problems.

With the popularity and reducing costs of Graphical Processing Units (GPU), several DL techniques could be developed and improved to solve these problems. Its high success can be mainly stated because of the automated feature engineering approach, exchanging the responsibility of crafting relevant features from human-expert to the machine. The popularity of deep learning and easy employment boosted more studies about components and normalizations. Also, improvements in the network structure itself, for not only a better understanding of models but also to increase performance (e.g., in terms of speed and accuracy) (LeCun et al., 2015; Goodfellow et al., 2016).

Designing artificial neural networks requires domain expertise since different operations combinations and hyper-parameters highly change the outcome. Attaining a suitable configuration demands many trials until a good combination is found. It was necessary to craft techniques to extract features from raw data in conventional machine learning. In DL, it is necessary to craft the network to discover features based on some data. Although the network automatically extracts features, knowledge of how to design the network is required. Because its structure

defines how well it can represent some problems. To increase the high level of this problem solution, Neural Architecture Search (NAS) aims to generate a problem-specific model without tuning the network. NAS, a subfield of AutoML (Automated Machine Learning), automatically generates a robust network, which needs a deep learning expert for a good outcome. Also, it enables the discovery of new effective patterns in the network structure by searching in a diverse space with less bias than a human expert (Hutter et al., 2018).

There are many different NAS approaches and how they produce new models to be evaluated. One of them is the Neural Architecture Search (same name as the subfield) developed by Zoph and Le (2016). A recurrent neural network (RNN) controller samples an architecture and trains a child network. Then, the network accuracy is passed to the controller to improve it, generating better architectures through time (illustrated in Figure 1.1). The main objective of a NAS approach is to travel through a determined search space to find suitable models for a specific dataset without human intervention in the architecture design.

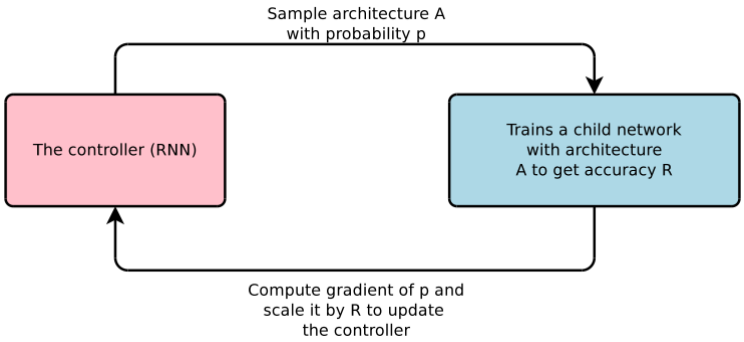


Figure 1.1: Overview of the NAS approach proposed by Zoph and Le (2016).

Although NAS is a recent field, with only the last five years with relevant and diverse researches, works increased rapidly. AutoML.org has a list[1] containing diverse research papers related to NAS. At the beginning of 2019, there were around one hundred papers. In mid-2021, more than one thousand papers are referenced there.

The primary purpose of our work is to develop a NAS approach that produces specific data models quickly and low-costly. It is mainly based on Gene Expression Programming (GEP) for search space representation and candidate model generation. In the following sections, we briefly describe the problem to be addressed and our proposal, with challenges, motivations, and contributions to the area of architecture search.

1.1 MOTIVATION

NAS may increase performance in a vast ecosystem of problems focused on adapting networks based on a specific problem. Also, it may propose a friendly way to non-experts for deep network modeling. So, the proposed work is being developed based on the following motivations:

- Facilitate the development of customized networks for specific tasks – whether being fully automated or for generation of possible candidate models which experts would further improve;
- Discoveries of new representations which may improve not only results in specific problems but may aid further research in how different operations may be combined. We

[1] https://www.ml4aad.org/automl/literature-on-neural-architecture-search/

focus on representation discovery in cell-based networks. A *cell* is a small representation that combines a few operations (like concatenations, additions, and convolutions). Then, this cell representation is replicated many times, generally stacked sequentially, to form a network;

- Fast and efficient search methods that would only require more accessible computational resources. Like providing competitive models using less than one day of GPU time.

1.2 PROBLEM DEFINITION

In the case of neural architecture search, three main dimensions need to be defined to categorize a NAS approach: search space, search strategy, and performance estimation strategy (Hutter et al., 2018). These dimensions define: (1) which possible models will be represented; (2) how these models will be discovered; and (3) how we can define the quality of a model. These dimensions are discussed below:

- **Search space:** Defines which models will be represented and can be discovered through NAS search (examples can be seen in Figure 1.2). A vast search space may introduce a diverse ecosystem of models, which would not limit it to specific and sub-good representations. However, it may also introduce too many noisy architectures, exhausting computational resources, and not discovering a good representation in time. So, it is crucial to reduce the search space to a minimal space without compromising the ability to find innovative and good models. Depending on how this reduction is implemented, it may include only models based on hand-crafted architectures and remove innovative ideas (heavy inclusion of human bias). This dimension focus on **search space representations which can maximize good and innovative models and minimize noisy ones.**

- **Search strategy:** Defines how an algorithm should travel through the search space. Therefore, it is highly influenced by how this search space is constructed. The strategy defined needs to discover suitable architectures fast (i.e., exploit the current landscape being searched). Together, without being stuck in a local good (i.e., explore new lands to find innovative networks). A premature convergence highly causes the latter. If the algorithm is too greedy to find suitable architectures that focus only on a small space, removing different but possibly better architectures. A strategy focused on exploration may discover models with different structures between them. However, its quality would be lower since its focus on minor improvements would be smaller. This dimension focus on **search strategies which provide a good trade-off between exploration and exploitation.**

- **Performance estimation strategy:** Defines how an algorithm should evaluate the models discovered. The primary approach is to train a model for E epochs and validate unseen data. This approach would limit the number of architectures to be evaluated in limited time and resource, as the value of E or the number of samples increase. The problem in reducing these numbers is to increase the discrepancy between the current model and the same trained until convergence. Other approaches like network morphisms and one-shot architecture search reduce training time and approximate fitness of current models to their fully-trained counterparts. This dimension focus on **performance estimation strategies that aid in model quality prediction with also consuming as little computational resource as possible.**

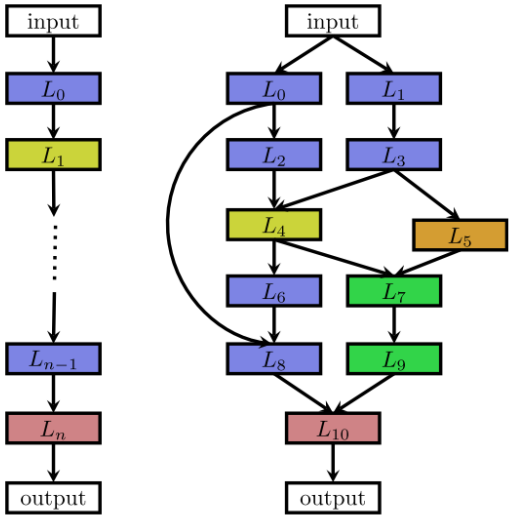


Figure 1.2: Examples of different architecture spaces. In the left, we can see a more sequential network (VGG-like), which each L_i is a layer of some type (convolution, pooling, etc) with different parameters. In the right, a space with branches and skip-connections (Inception-like). From (Hutter et al., 2018).

A NAS approach needs to address these three dimensions to produce networks with relevant improvements, considering the resources needed to be used. The challenges in developing it, mainly focusing on our work's direction, are described next.

1.3 CHALLENGES

Optimization of networks is not something new. Evolutionary algorithms tried to develop efficient but straightforward and shallow networks. Nevertheless, the NAS field – search and optimization of deep learning architectures, is recent. Over the five years, diverse research was employed to find customized and efficient models. However, this field is at an initial stage. Thus, many problems are yet to be solved. This recent trend, combined with the complexity in finding novelty and efficient deep neural networks, is a characteristic of NAS research. In our work, the following challenges are found and need to be addressed:

- Developing a robust yet efficient architecture search, which provides competitive models without exhaustive computational resource consumption;
- Designing a wide but effective search space containing novelty network representations. Not only different than manually designed networks but also effectively represents the samples (e.g., images, texts, audio) evaluated;
- Proposing and evaluating an efficient methodology for architecture search in terms of accuracy and speed, with limited resources (in processing and time);
- Evaluating the representation of the best candidates, as to understand which blocks and patterns contributed for them to be chosen;
- Developing a NAS approach that can be generalized to different network types (e.g., convnets, recurrent nets, auto-encoders) and problems (e.g., image classification, segmentation, time series prediction) with little modification.

1.4 HYPOTHESIS

Our central hypothesis is that **evolutionary-based NAS can be used in a low-cost GPU search (combined with a robust strategy and efficient search) to be employed in diverse computer vision tasks without losing its competitiveness**. With this hypothesis in mind, we developed custom NAS approaches to be able to answer some of the following questions:

- Can we reduce the search phase to less than a GPU day of evolutionary approaches without hindering the search of good models?
- Is it possible to increase the cell-based search space to employ different structures for each model stage without accuracy loss?
- Is it possible to remove the requirement of fixing a good number of blocks for the network cells?
- Can we migrate to other domains, like image segmentation with a different search space, and provide competitive networks for a specific problem?

With the hypothesis confirmed, further research in evolutionary-based NAS approaches can be focused on with low-cost search in mind. Also, new patterns and a broader search space can be evaluated without consuming excessive computational resources.

1.5 OBJECTIVES

This work aims to develop a NAS methodology based on GEP – its representation adapted to deep networks and its selection and reproduction systems. The following features are also proposed in our approach (with their respective objectives):

- Development and comparison of two evolutionary-based approaches: one with sequential training and another with dynamic training. These approaches focus on the fast search phase using weight sharing/inheritance and few training iterations per candidate;
- Application of GEP features to generate the search space – like using Automatically Defined Functions (ADFs) as blocks of a cell;
- Not only are these blocks' structures shared between candidate models but their weights are also stored and inherited by newer candidates for fast convergence and fewer GPU processing times. Weight sharing and inheritance (focused in similar combinations) aid in achieving faster convergence;
- Generalization for other kinds of deep networks problems besides classification with convnets, in this case, medical semantic segmentation with cross-modality – computer tomography (CT) and magnetic resonance imaging (MRI).

1.6 CONTRIBUTION

Based on the motivation behind the NAS study, the objectives established, and the work developed, the following contributions were made:

- To the best of our knowledge, the proposed thesis was the first GEP-based NAS, where a work-in-progress was published in (Alves and Oliveira, 2020);

- Two different evolutionary-based NAS approaches (static and dynamic strategies) for efficient and fast architecture search, using only 0.5 GPU day in the search phase with a single NVIDIA RTX 2080Ti;
- Evaluation of a dynamic evolutionary-based NAS approach in contrast with the sequential nature of a common evolutionary-based NAS;
- A more complex cell-based representation that does not require configuring a fixed number of blocks. Our approaches can find efficient models only by configuring an interval of the possible number of blocks;
- Easily adapted approaches to new problems without much modification and search cost. Furthermore, it provides competitive networks without transfer learning and can employ its usage for further improvements;
- An easy and low-cost way to generate custom models for specific tasks and datasets. These methods can aid users to achieve satisfactory results in unfamiliar problems or help inexperienced users to employ custom models to solve tasks of interest. Also, reduce the time to finetune models outsourcing this task to an automated search.
- And, finally, approaches ready to be applied in image classification tasks and medical image segmentation with cross-modality.

1.7 DOCUMENT ORGANIZATION

In the first part of our thesis, we present the theoretical basis to understand our approach and related works focused on CNN and NAS approaches. Chapter 2 presents concepts regarding deep learning, convnets' structure, and optimization, and metrics to evaluate these models. In Chapter 3, we present the concepts to understand the gene expression programming, used as the basis of our search strategies to understand our thesis.

In Chapter 4, descriptions about related works regarding NAS, their results and contributions are found. A better understanding of the NAS field and dimensions to which a particular approach may be categorized is also given. The datasets evaluated in our thesis and studied in these related works are also described. An overall discussion is present, showing some directions to be attended.

In the second part, the proposed thesis with its experiments is discussed. In Chapter 5 we construct the basis of our approach, showing how candidate models are generated, different policies to attend some particularities in generating deep convolutional networks, and many strategies to aid in the architecture search. As for Chapter 6, we present our proposed thesis for neural architecture search, employing two evolutionary-based NAS approaches.

Chapter 7 contains information about the experiments to be employed, the ablation study and other results, then the overall discussion and comparison with other works in Chapter 8. Finally, Chapter 9 presents the final considerations of our thesis.

Part I

Theoretical Basis

2 DEEP LEARNING

This chapter aims to briefly describe concepts from deep learning (mainly related to convolutional networks).

The central idea in deep learning is to learn the representation of some problem through analyzing a representative amount of data. Patterns representing local or global features are discovered using the backpropagation algorithm, changing the network structure and hyper-parameters for a better representation. Improvements in different areas like image and speech recognition, computer vision, and bioinformatics are found, setting the state-of-the-art in different problems (LeCun et al., 2015; Goodfellow et al., 2016; Bengio et al., 2021).

2.1 CONVOLUTIONAL NEURAL NETWORKS

Like conventional networks, a convolutional neural network (CNN or convnets) consists of input, output, and hidden layers. An image is used as the CNN's input. Several operations are applied to this image in the hidden layers. The probabilities of this image being from specific classes are returned as the network output after being optimized using the backpropagation algorithm (Rumelhart et al., 1988). In these hidden layers, there is the convolution operator (cross-correlation, mathematically). The purpose of this operator is to extract information from the neighborhood of pixels in the image using small rectangles – called kernels. With this, information from an image can be represented more efficiently, as crucial information is extracted with a large number of calculations (like fully connected layers, which employ cross-correlation in the entire image at once) (Goodfellow et al., 2016).

Another commonly used operator is the pooling (maximum or average). Using a kernel generally of size (2x2) extracts the maximum or average value from a neighborhood. It is highly used combined with the stride attribute, which sets how many pixels a kernel will jump horizontally or vertically in the window-sliding to obtain information from the entire image. With a stride value equal to 2, the network can reduce the image size by half. This striding decreases the computational power necessary for feature extraction in more prominent neighborhoods (instead of increasing kernel size). Stride can also be used in convolution operations, which will not only learn to extract useful information but also represent the feature map with smaller sizes (compression) (Goodfellow et al., 2016).

Increasing the depth of a network inserts a problem called vanishing gradient. The resulting calculation may include values close to or equal to zero depending on convolutions applied to the image. These values incur irrelevant information to be propagated to the output. One idea to address this problem is the residual information ((Srivastava et al., 2015; He et al., 2016)). The standard approach is to sum a convolutional block's output with its input. This approach reduces the possibility of vanishing gradients and uses information generated in previous layers on the latter. Given a previous layer's output h_{i-1} and a set of operations F, the next layer's output h_i is calculated as:

$$h_i = F(h_{i-1}) + h_{i-1}. \tag{2.1}$$

2.2 OPERATIONS IN CONVNETS

In this section, we aim to describe in more detail the different operations used in the state-of-the-art approaches involving ConvNets, but mainly focused on the operations used in this work.

2.2.1 Normal Convolution

Also only called convolution, this operation is the basic one in this type of network (also, the basis for identifying convolutional networks). Although commonly called convolution, it is a cross-correlation. Thus, the weight flipping from the true-convolution does not apply here (Zhang et al., 2020).

Input and a kernel (i.e., the weights of a convolution operation) are combined to generate the output. Figure 2.1 illustrates a convolution in a two-dimensional input. A kernel is applied as a window sliding in the entire input. Each input position is multiplied by the corresponding position at the kernel (see the blue area). Then, the results are summed and inserted into a corresponding position in the output. After, the kernel slides a s value, which is called stride (generally, $s = 1$).

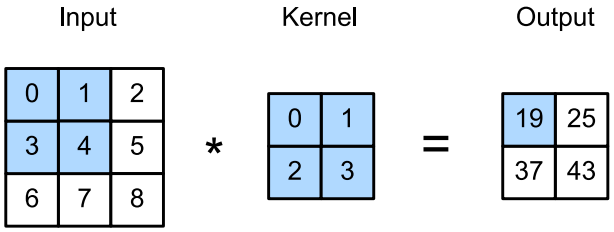


Figure 2.1: Simple example of a convolution applied to an input. From (Zhang et al., 2020).

The output is generated according to the input and kernel, even its size. In the previous example, an input of 3×3 and a kernel of 2×2 generated an output of 2×2. The output size can be controlled according to the sizes of input and kernels, but also with other hyper-parameters.

Generally, the most common hyper-parameters used for this purpose are the padding and stride. The padding can increase the input size, generally with zeros, to increase the output size. For example, we can use the padding of $p = 1$ in the previous input to generate an output of size 4×4. Figure 2.2 illustrates this process.

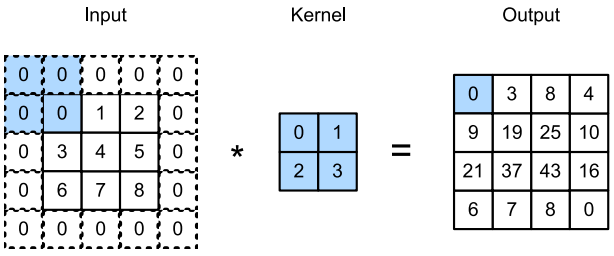


Figure 2.2: A convolution with padding. From (Zhang et al., 2020).

In this case, we can use padding to increase the output size. However, there are some cases in which we may desire to reduce the output size. Changing the stride value may help us achieve that. The default value for stride is $s =$ for every dimension in the input. For example, a 2D input has $s = 1$ for its width and height. So, the windows slide one position ($s = 1$) for the following calculation (horizontally or vertically). One of the most common approaches for using

convolution with striding is to set the value $s = 1$ to reduce the output size to half of the input size (Zhang et al., 2020).

Another example is illustrated in Figure 2.3. Here, a stride of 2 for the width and 3 for the height are used. So, four calculations are applied to the input. First, the calculation is applied to the upper left of the input. Second, a sliding of two positions is applied to the right, and the kernel convolutes the input (blue region). Third, the kernel returns to the left-most of the input, but three positions are sliding to the bottom. Then the bottom-left region is convoluted (also in blue). The final operation is applied after sliding to the right (with the bottom left as the point of origin).

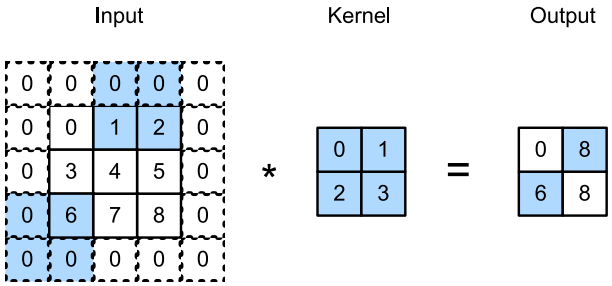


Figure 2.3: A convolution with stride $s = (2, 3)$. From (Zhang et al., 2020).

This step occurs when a kernel is applied to a single-channel input. However, deep convolutional models have multiple channels. So, how is convolution applied in multiple channel inputs? Figure 2.4 illustrates that.

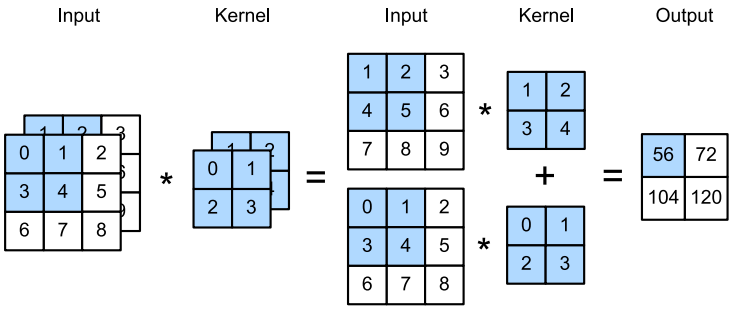


Figure 2.4: Example of the normal convolution applied to multiple channels. From (Zhang et al., 2020).

In the case of normal convolutions, each channel of output may have multiple kernels. In the illustration, we have a single-channel output and an input with two channels. Each channel of the input has a different kernel associated with it and one channel of the output. After applying the kernels in the channels, the resulted calculation are summed to generate the output feature map.

The layer represented in Figure 2.4 has eight parameters (two kernels with four parameters each). In the case of normal convolutions, we can calculate the number of parameters of a layer with the following formula:

$$l_w = i_c * k_h * k_w * o_c, \tag{2.2}$$

with i_c being the input channel size, (k_h, k_w) the kernel size and o_c the output channel size expected. If the output channel size o_c is increased to 16, this layer would have 128 parameters $(2 * 2 * 2 * 16 = 128)$.

Usage of padding and stride contributes to preserving the feature map size or reduces it to divisible sizes. Usage of stride combined with successive convolutions increases the receptive field size, contributing to the extraction of more global features as we go through the depth of a model.

2.2.2 Pointwise Convolution

Previously, a normal convolution was described. The 1×1 convolution, commonly called pointwise convolution, is equal to a normal convolution but with a kernel size of 1×1. The main idea here is not to extract the correlation between local pixels but the correlation between channels. This correlation can be seen in Figure 2.5. It was first seen in the Network In Network approach (Lin et al., 2013), being widely used for dimensionality changing and extracting channel relationships.

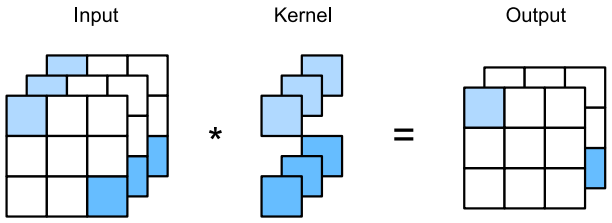


Figure 2.5: A pointwise convolution. From (Zhang et al., 2020).

Generally, a pointwise convolution will have padding equal to zero and stride $s = 1$, generating an output with a size equal to the input. Since this convolution has a kernel of 1×1, the number of parameters is significantly reduced without compromising the correlation between multiple channels. For example, with input with 64 channels and output of 128 channels, we have a weight size of 8192 parameters using the pointwise convolution. Using a 3×3 kernel, we would use more than 70 thousand parameters.

2.2.3 Depthwise Convolution

Quite the reverse of the pointwise convolution, the depthwise convolution focus on the local correlation of individual channels (Chollet, 2017). Inspired by the grouped convolutions (seen in (Xie et al., 2017)), the channels of input are split, and then a kernel (one for each channel) is applied to its respective channel. Then, the outputs are stacked together again. An illustration can be seen in Figure 2.6.

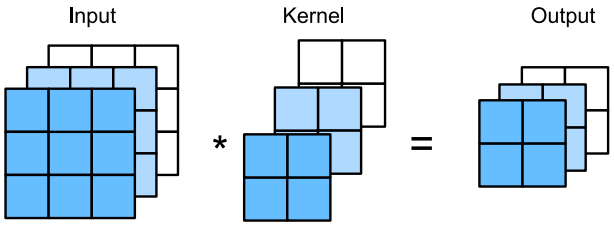


Figure 2.6: A depthwise convolution.

2.2.4 Depthwise Separable Convolution

A regular convolution applies cross-correlation spatial and channel-wise simultaneously. However, the number of parameters used is drastically increased as the number of channels and kernel size

increase. A large number of parameters not only increases computational resource usage but also causes overfitting. Thus, some authors choose alternative solutions to reduce parameter counting with minimal impact on the model prediction. The depthwise separable convolution was proposed to apply spatial and channel-wise cross-correlation with parameter reduction. These two steps were split into the two convolutions previously described – pointwise and depthwise convolutions, applying them sequentially (Chollet, 2017).

Although convolutions' order can be switched, the idea is to apply both convolutions sequentially to determine the correlation between channels and in-channel local pixels. To access the parameter reduction, we can pick up an example layer with a kernel size of 3×3, input channel size of 64, and output channel size of 128. Using Equation 2.2, almost 74 thousand parameters are necessary for this single operation. If a depthwise separable convolution replaces this operation, we can have a different number of parameters according to the order of convolutions.

First, we need to describe the formula to calculate the parameters of the depthwise convolution, which is seen as follows:

$$l_w^d = h_c * k_h * k_w, \tag{2.3}$$

which h_c is the number of channels the hidden layers will serve as the input. Since only spatial-wise cross-correlation is applied, the number of channels remains the same as the input. Now, we can calculate the number of parameters for the two cases – first, the case when the pointwise convolution comes first.

Here, we first use this operation for channel-wise correlation and, generally, to increase channel size. Equation 2.2 is used to calculate the number of parameters of the pointwise convolution, which is 8192 ($64 * 128 = 8192$). Then we apply Equation 2.3 to calculate the parameter size of the depthwise convolution, which is 1152 ($128 * 3 * 3 = 1152$). In the first case, this entire operation uses 9344 parameters (a little more than 12% of the respective normal convolution).

For the second case, we first calculate the number of parameters of the depthwise convolution, then for the pointwise one. For the first operation, we need 576 parameters ($64 * 3 * 3 = 576$). Then, for the second operation, we need 8192 ($64 * 128 = 8192$). In this case, 8768 parameters are used (less than 12%).

In both cases, parameter reduction is remarkably efficient. With this reduction, increasing the number of channels or model depth is common for better prediction. This type of combination is widely used in state-of-the-art approaches, like in (Howard et al., 2019; Tan and Le, 2021).

2.2.5 Dilated Convolution

Also called atrous convolution, the dilated convolution is a type of convolution that inserts holes between a kernel's positions and applies this kernel to the image. Figure 2.7 illustrates a 3×3 kernel ith dilations equal to one (default kernel), two and three. As we can see, increasing the dilation rate increases the receptive field without applying the stride. With according paddings, we can preserve the size of the convoluted input. In segmentation tasks, where the precision of the resolution must be preserved, dilated convolutions can be effective (Yu and Koltun, 2015).

2.2.6 Transposed Convolution

In the previous convolution operations, we saw different features to preserve or reduce the feature map size. However, in some cases, it is necessary to upsample the feature map. For example, in segmentation tasks, models apply successive convolutions (or even pooling) with striding to

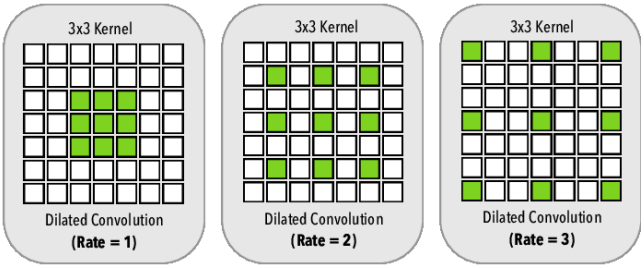


Figure 2.7: Convolutional kernels with different dilation sizes (1, 2 and 3, respectively). From (Perone et al., 2018).

extract more global features with a large receptive field. The last feature map, which is used as a heat map containing the class probabilities of each pixel, has a low spatial resolution. This heat map must have a similar or even equal size to the input for precise segmentation. One approach is to apply only dilated convolutions, but it may have excessive memory consumption (Ronneberger et al., 2015; Zhang et al., 2020).

Another approach is to apply transposed convolutions to the low-spatial feature maps. One famous segmentation model family that uses this approach is the U-Net family (Ronneberger et al., 2015). This network is a convolutional encoder-decoder model. The encoder increases the receptive field and decreasing the feature map size. In the decoder part, transposed convolutions increase the feature map size to the original input size.

Figure 2.8 illustrates the transposed convolution with a 2×2 kernel. For each position of the input, the kernel is applied. After, the resulted matrices are summed to generate the output.

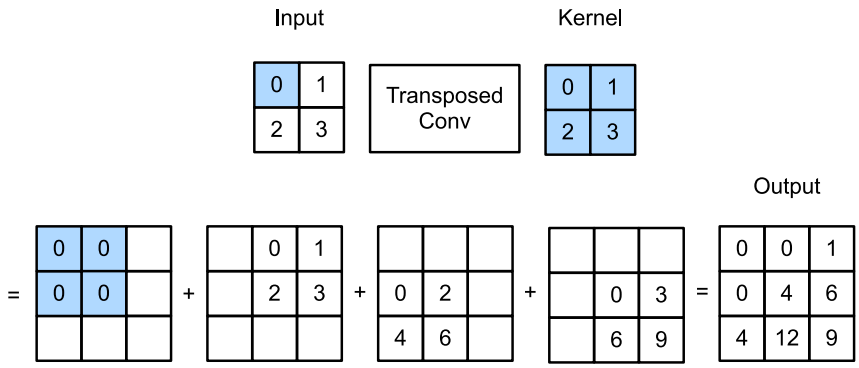


Figure 2.8: Example of a transposed convolution. From (Zhang et al., 2020).

In Figure 2.9 the same transposed convolution is applied, but with a stride = 2. In this case, no overlapping occurs in the resulted matrices.

2.2.7 Pooling

Differently of the convolution, the pooling operator does not have a kernel. Thus no weight optimization is made. Like the convolutions, a window size must be configured. This window slid in the entire input, applying some specific operation. There are two familiar used pooling operators: average and maximum poolings (Zhang et al., 2020).

In Figure 2.10, we can see 2×2 max-pooling applied to the input. In this case, the maximum value of a window of 2×2 is chosen (i.e., the maximum value between four values).

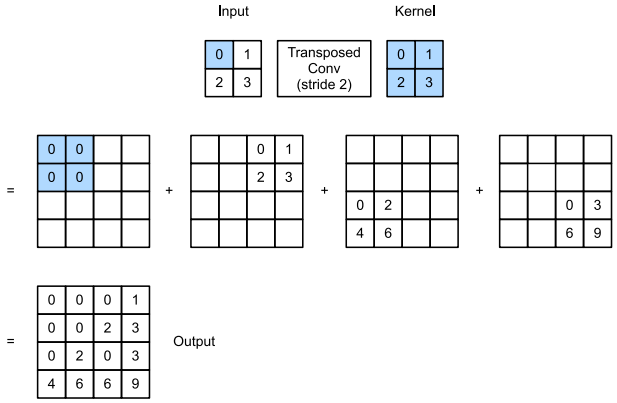


Figure 2.9: Example of a transposed convolution with stride = 2. From (Zhang et al., 2020).

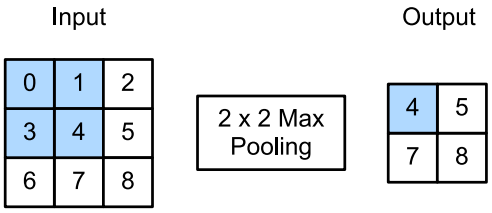


Figure 2.10: Example of a max pooling. From (Zhang et al., 2020).

2.2.7.1 Global Average Pooling

The Global Average Pooling (GAP) is the same operation as the average pooling but applied only once in an entire channel. This operation was designed to reduce and even replace fully connected layers in convnets. GAPs reduce resource computation since there are no parameters to compute or optimize. It also works well combined with convolutions. For each channel, the operation is applied, generating a vector with its size equal to the number of channels (Lin et al., 2013).

2.2.8 Non-linearity with the Activation function

Activation functions are widely used in neural networks, mainly for the addition of non-linearity in these networks. Stacking hidden layers become futile without activations since it would transform a neural network into a linear regression model. Usage of activations enables the increase in the complexity and efficiency of these networks (Goodfellow et al., 2016; Zhang et al., 2020).

2.2.8.1 ReLU

The Rectified Linear Unit (ReLU) is one of the most popular and simple activation functions. Equation 2.4 shows the formula for this activation. The ReLU activation only outputs values greater or equal to zero. If an inputted value is negative, ReLU outputs a zero (Glorot et al., 2011).

$$ReLU(x) = max(x, 0) \tag{2.4}$$

Figure 2.11 shows a plot of ReLU's output between an interval. Its derivatives work well enough for faster optimization and reduce the vanish gradient problem (Zhang et al., 2020).

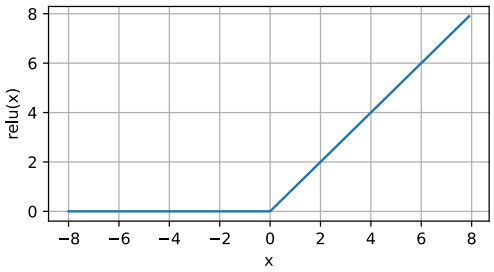


Figure 2.11: Output of the ReLU activation. From (Zhang et al., 2020).

2.2.9 Batch Normalization

One of the most impressive improvements in deep learning is Batch Normalization (BN). Training of neural networks drastically decreases with its usage since it focuses on reducing internal covariate shift (i.e., changing the data distribution). Also, BN helps in the gradient flow by reducing gradient dependency on the parameter scale. Usage of BN minimizes the usage of dropout for regularization of the weights (Ioffe and Szegedy, 2015).

2.3 EVALUATING A MODEL

Evaluation of a model is necessary to reduce the generalization error (evaluating a validation set) and to reduce training error (evaluating a training set). A metric must be carefully chosen depending on the task in hand (Zhang et al., 2020). Therefore, we describe in this section some evaluation metrics necessary for better optimization of the generated models. Nevertheless, first, we must define the following metrics:

- True Positive (TP): Sample from the positive class is correctly classified as such;
- True Negative (TN): Sample from the negative class is correctly classified as negative;
- False Positive (FP): Sample from the negative class is classified as positive;
- False Negative (FN): Sample from the positive class is classified as negative.

2.3.1 Accuracy

Accuracy is commonly used to evaluate image classification in balanced datasets. Using the metrics defined above, we can calculate the accuracy of a method with the following formula (Mower, 2005):

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.5}$$

2.3.2 Dice similarity score

The Dice similarity score (DSC) was independently proposed in (Sorensen, 1948) and (Dice, 1945), which is also called F1-score and widely used in ML. It calculates the similarity of two samples (e.g., predictions and correct labels). Also, very used do assess the quality of an image segmentation without taking into account the background. DSC is calculated as follows:

$$DSC = \frac{2TP}{2TP + FP + FN} \tag{2.6}$$

2.3.3 Loss function

The loss function dictates how to evaluate the training error of a model. This function evaluates how well the probabilities generated (generally returned by a softmax function) reflect a training sample. Depending on how we define a loss function, we can optimize a model differently. For example, a loss function can reduce the precision and recall of a training set and disregard the correct classification of negative samples (Zhang et al., 2020).

In this work, mainly two loss functions are used (which are also popular in their fields): cross-entropy and soft-Dice loss functions. Before entering in more details, we must define some things: I is the sample to be evaluated. Can be images from a dataset (for image classification) or pixels/voxels from an image; y_i is the actual probability of the ith sample to be from his class; $\hat{y}_i$ is the predicted probability of the ith sample to be from this class; Y and $\hat{Y}$ are the set of actual probabilities and set of predictions, respectively (Bertels et al., 2019).

2.3.3.1 Cross-entropy loss function

The cross-entropy loss function is one of the most popular loss functions for convnets optimization. It is widely used for model optimization in classification problems. This function aims to minimize the negative log-likelihood. The cross-entropy loss function is calculated as follows (Bertels et al., 2019):

$$CE(\hat{Y}, Y) = -\sum_{i=0}^{I-1} [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)] \tag{2.7}$$

2.3.3.2 Soft-Dice loss function

Based on the Dice Similarity Coefficient, the soft-Dice function calculates the loss based on the Dice score but using the probabilities. Thus, we can optimize a model based on how well it is making its prediction of the regions of interest (ROI), excluding the correct prediction of the negative class (generally the background for image segmentation) of the loss calculation. The soft-Dice loss function is calculated with the following equation (Bertels et al., 2019):

$$SD(\hat{Y}, Y) = 1 - \frac{2 \sum_{i=0}^{I-1} \hat{y}_i y_i}{\sum_{i=0}^{I-1} \hat{y}_i + \sum_{i=0}^{I-1} y_i} \tag{2.8}$$

2.4 OPTIMIZING A DEEP MODEL

After structuring a neural network, we must training it with several samples for the model to be robust to new data. There are a few hyper-parameters that need to be configured for efficient model optimization. In this section, the approaches used to optimize the automatically generated models are discussed.

2.4.1 Optimizers

One of the most important parts of deep learning training is to configure an optimizer. The loss function is the objective function of an optimizer. These optimizers aim to reduce the loss of the training dataset (Zhang et al., 2020).

2.4.1.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an iterative optimizer which uses minibatches to calculate an expectation of the gradient. SGD is used instead of computing the Gradient Descent of the entire data in one way. It is not only faster and computationally viable, but SGD also have much better convergence. To optimize a weight $\mathbf{w}$ with SGD, the following equation is applied,

$$\mathbf{w_t} \leftarrow \mathbf{w_{t-1}} - \eta \nabla f(\mathbf{w_{t-1}}), \tag{2.9}$$

where η is the learning rate (controls the step in which the weights change) and the $\nabla f(\mathbf{w})$ is the gradient of the loss function $f(\mathbf{w_{t-1}})$ at time t (Zhang et al., 2020).

2.4.1.2 Adam and AdamW

Adam (Kingma and Ba, 2014) and AdamW (Loshchilov and Hutter, 2018) are adaptive learning rate optimizers suitable for optimization in problems with noisy and sparse gradients. Compared with SGD, they achieve convergence much faster and are especially appropriate when hyper-parameter tuning is not viable. Adam optimizer calculates the weight update by the following equation:

$$\mathbf{w_t} \leftarrow \mathbf{w_{t-1}} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.10}$$

with the following parameters aiding the calculation of the weight update:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \tag{2.11}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \tag{2.12}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \tag{2.13}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \tag{2.14}$$

$$g_t = \nabla_\theta f_t(\theta_{t-1}) \tag{2.15}$$

with η as the learning rate, ϵ as a small constant (typically 1e-8) to prevent division by zero, and $\beta_1 = .9$ and $\beta_2 = .999$ as the forgetting parameters. AdamW is a modification of the original Adam but decouples the weight decay from the gradient update. This modification helps the generalization power of the model.

2.4.2 Learning Rate Scheduler

Learning rate contributes considerably to optimizing a model, and a suboptimum value may hinder this optimization. For example, high values may result in divergence and weight exploding, but low values may result in high training time and stuck in local minima. Learning rate schedulers enter the scenario to automatically change the learning rate according to the time (Zhang et al., 2020). In this work, mainly two similar learning rate schedulers are used: cosine annealing and cosine one-cycle.

2.4.2.1 *Cosine Annealing*

The cosine annealing scheduler aims to persist in the first epochs with a high learning rate η_0 (defined manually) and decreases continuously to refine the model with lower learning rates (Loshchilov and Hutter, 2016; Zhang et al., 2020). The learning rate decreases until η_T (which is commonly used as $\eta_T = 0$). The following equation demonstrates the calculation of the learning rate at step t (from zero to T):

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2}\left(1 + \cos(\pi t/T)\right) \tag{2.16}$$

Figure 2.12 illustrates how the learning rate decreases over time.

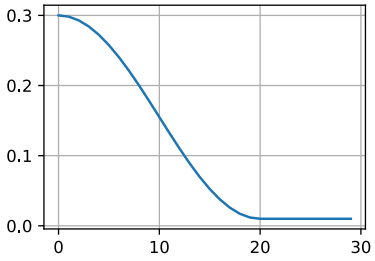


Figure 2.12: Plot of the cosine annealing scheduler with initial learning rate of 0.3 and steps $T = 20$. From (Zhang et al., 2020).

2.4.2.2 *Cosine One-Cycle*

Like the cosine annealing, the cosine one-cycle learning rate scheduler generates the learning rate based on the cosine. However, cosine annealing begins with a low learning rate (typically, $\eta_0 = \eta/25$) and increases along the time until the scheduler reaches the configured learning rate η. Then, it begins to decrease like cosine annealing. This scheduler begins with a low learning rate to aid in weight stabilization. Also, the momentum changes inversely to the learning rate (Smith, 2018). Figure 2.13 illustrates an example of the cosine one-cycle learning rate.

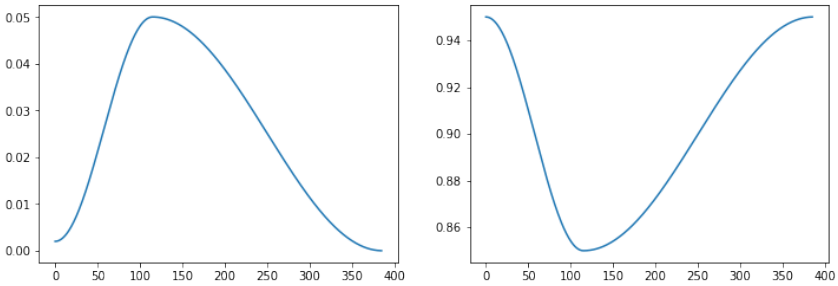


Figure 2.13: Example of learning rate and momentum values calculated by the Cosine One-Cycle Scheduler for hundred of epochs. From `https://fastai1.fast.ai/callbacks.one_cycle.html`.

2.5 DATA AUGMENTATION

One of the most common problems in ML and DL is over-fitting, which is when a model is highly biased to the training data. The model fits these samples in such a way that reduces its generalization significantly. Thus, the prediction of other samples becomes poor. Generally, low variability in the training set causes over-fitting. To increase sample variability and to reduce

over-fitting, a common approach is to apply data augmentation (Goodfellow et al., 2016; Zhang et al., 2020).

Data augmentation aims to generate similar training samples but random changes that may present robust variations to the model. For example, flipping and translation may introduce objects of interest in different positions. Random changes to the brightness, contrast, and saturation may increase the model's robustness to these variations. The main idea is to provide augmentation that contributes to model invariant to these changes (Goodfellow et al., 2016).

One of the most used data augmentation strategies is the AutoAugment (Cubuk et al., 2018). A set of policies was found in popular datasets (e.g., CIFAR, ImageNet, SVHN) that reduced overall test error in these datasets. Online data augmentation is employed when using AutoAugment. Online data augmentation is when, at each training batch, random augmentations are applied to the batch. This approach increases data variability, which also reduces the probability of a model memorizing the same data.

One of the augmentations employed in Auto Augment is AutoContrast. This augmentation maximizes the contrast by stretching the histogram, changing the darkest pixel to black and the lightest pixel to white. Then, other pixel values are modified accordingly (Cubuk et al., 2018). Random horizontal flipping and random cropping (a specific region of the image) are standard data augmentation methods applied to image classification, being popularized in (He et al., 2016). Combined with AutoAugment, Cutout (DeVries and Taylor, 2017) is another robust data augmentation technique. Cutout randomly applies a black square in a random position of the input. This approach forces the model to learn features of a class from different places of an image.

Not only does image classification benefit from data augmentation. For example, medical image segmentation greatly benefits from different augmentations techniques since low sample data is available. Different affine, elastic, and pixel-wise augmentation methods can be applied at training time to increase generalization (Nalepa et al., 2019). Some of these augmentations are elastic deformation (Simard et al., 2003), curvature flow (Malladi and Sethian, 1996), shear, translation and rotation.

2.6 CONCLUSION

In this chapter, we briefly explained diverse concepts related to deep learning. The structure of a convolutional network, different operations, its optimization, and model evaluation were discussed. In the next chapter, some concepts of evolutionary computation are described.

3 EVOLUTIONARY COMPUTATION

This chapter briefly describes concepts from evolutionary computation (mainly related to gene expression programming) used to understand the proposed approach.

Evolutionary computation is a sub-area of computer science in which its algorithms and concepts are based on the biological process of evolution – mainly Darwinian one. Processes from nature are interpreted and convert to ideas that may simulate or be roughly inspired by these processes (Eiben et al., 2003).

As we can see in nature, the power of evolution is incredibly evident in how species adapted and survived in many hazardous environments. The core of evolution is the process of trial and error: a vast amount of combinations are tested to solve a specific problem. Different from a random search, in evolution, the many combinations pass on their genetic to further generations, which may guide to a more intelligent search than only combining random data. Also, the presence of mutation incurs an exploration of different landscapes not traveled by prior generations.

In an environment with limited resources, a group of individuals – called population – strive to survive and reproduce with other individuals and pass on their genetic material to the next generations. An evolutionary algorithm (EA) is a computational technique constructed on this philosophy.

The individual can refer to the candidate solution (phenotype), i.e., the solution to solve a specific problem, or to the EA representation (genotype), also called a chromosome. This chromosome can be a string or even a tree-like structure. It represents a candidate solution and can be defined as the coordinates of this solution in a space problem. Then, some EA aims to travel through this space to find excellent coordinates for a specific problem.

The population is a set of these coordinates but also a set of candidate solutions. It is necessary to find out the best and relevant individuals to improve candidate solutions. A fitness score is calculated for every individual to evaluate the population. Then, a reproduction step is employed to generate the next generation of individuals. This step focuses on obtaining patterns of the best individuals (but also of less fit ones), mutating, and combining (reproduction step) them to generate new individuals (children) who may present better fitness scores.

After children's evaluation, a survivor selection mechanism reduces the population to the initial size. An iteration of this process is called generation. This evolutionary process is executed until some generations are completed, or another rule happens (e.g., an individual with an excellent fitness score). The overall process of evolution is illustrated in Figure 3.1.

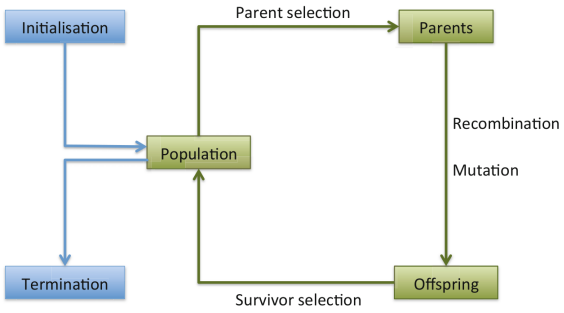


Figure 3.1: General process of an evolutionary algorithm. From (Eiben et al., 2003).

The most popular EAs are the genetic algorithm (GA), genetic programming (GP), evolutionary strategies (ES), evolutionary programming (EP), differential evolution (DE), and particle swarm optimization (PSO). Aside from these algorithms, there is a vast ecosystem of evolutionary-based techniques. One of them, similar to GAs and GPs, is gene expression programming – which will be described in the next section.

3.1 GENE EXPRESSION PROGRAMMING

Gene Expression Programming (GEP) is an evolutionary algorithm with linear (genotype) and ramified (phenotype) representations (Ferreira, 2001). It combines the simplicity of genetic algorithms for reproduction and the generation of complex models from genetic programming. In GEP, an initial set of individuals (i.e., population) is generated. Each individual consists of a string-like genotype with a fixed size, a representation of the real model. It is similar to DNA, containing elements that change the model based on their sequence and position. GEP translated this genotype to a tree-like representation, which is the real model of the individual (phenotype).

3.1.1 The genotype and phenotype representations

The genotype is divided into two sequences: head and tail. The head sequence contains functions and terminals. As for the tail, only terminals are found. However, what is the difference between them? As the name suggests, functions are elements that have an input (or many inputs). An operation is applied to it, producing some output (although the output may have different sizes or dimensions than the inputs, it is considered a unique output). Terminals are data to be used as inputs for functions (terminals do not have inputs), so no operation is applied when a terminal is found.

Lengths of head and tail are calculated based on a chosen value for the head and the size of head and arity (maximum number of inputs between used functions) for the tail. Being h the size of the head and a the arity, the length of the tail is calculated as follows:

$$t = h(a - 1) + 1. \tag{3.1}$$

For example, imagine that we are working with the function set composed of {+, -, *, /, squared root (as Q)} and terminal set {a, b, c}. In this case, the arity is equal to 2 ($a = 2$). Choosing a value of h equal to 8, we have the size of the tail as $t = 9$. An example individual can be seen below (tail in bold):

$$+a - *Qbc/\mathbf{abbcabcab}$$

With this genotype representation separated in head and tail, the expression tree (ET) – the phenotype – can be generated, using the first element of the head as the tree's root. Then, the tree is assembled with the following elements of the head (and then the tail) until all functions have input. There are two approaches to set the order of assembly:

- **Width-first**: As elements are put in the ET, it grows depth-wise. The order, in this case, is width-first – inputs of functions in the current depth are first filled before going to the next depth level. Generally, ETs, in this case, are wider if there is a high quantity of functions in the head.
- **Depth-first**: Different from the width-first, the idea is to focus on filling functions and their functions depth-wise, always trying to fill the newer functions inserted into the ET. In this case, ETs with many functions in the head will have a more deep structure.

3.1.1.1 Directed Acyclic Graph in GEP

The default phenotype of GEP is a tree with many levels. A tree phenotype may limit the possible configurations since a node in lower levels can not be used as an input of upper-level nodes. To insert node re-usability, Quan and Yang (2007) changes the GEP phenotype to support Directed Acyclic Graphs (DAG). This new genotype has two chromosome types: main and topological. The first contains the elements of an individual. Then, the topological has I sequences, where I is the number of possible inputs of any element in the individual. The topological sequences contain the index of the inputs of each element. As we can see in Figure 3.2 (with $I = 2$), an element may contain only indexes less than the element index. This pattern removes the possibility of phenotypes with cycles.

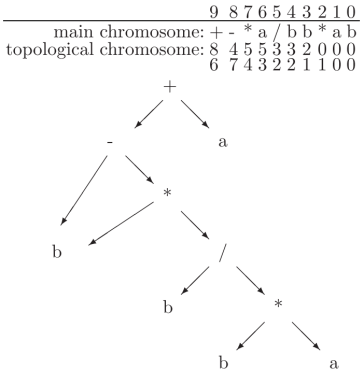


Figure 3.2: Example of the new genotype representation proposed in (Quan and Yang, 2007).

3.1.2 Genes with Automatically Defined Functions

In GEP, genes are considered parts of an entire chromosome. Generally, an individual is composed of many genes, with each gene being a sub-ET. Then, these sub-ETs are combined with some operator (addition, for example). There is also a more advanced approach that treats genes as elements – an Automatically Defined Function (ADF) (Ferreira, 2006a), being first introduced by Koza (1992), in Genetic Programming, for code reuse.

In its simplest form, a program/individual is composed of only one gene. This gene is a sequence of head and tail, which is translated to an expression tree. The genes' sequences are sequentially included in the chromosome sequence, where reproduction operators are applied. Individuals can also be multi-genetic, generating many ETs later combined with a function (addition, for example).

There is the ADF approach, which is composed of functions and terminals, similar to conventional genes. The main difference is the inclusion of a homeotic gene. This particular gene represents the main program of the GEP individual and its relations with the other genes (i.e., ADFs). In its default norm, terminals are removed and replaced with ADFs treated as the homeotic gene's terminals. The chromosome, ETs from each ADF, and the main program ET are illustrated in Figure 3.3.

Another ADF representation is the C-ADF proposed by Zhong et al. (2016) in their SL-GEP (Self-Learning GEP) approach. It is similar to the default ADF, except in which elements are replaced in the homeotic gene. In this case, terminals are considered as variables and constants, like in conventional genes. As for the function set, not only functions but ADFs are included. So, ADFs are now treated as functions, not terminals. This strategy enhanced the

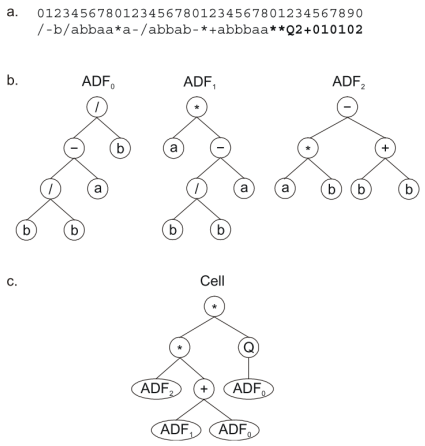


Figure 3.3: ADF examples. From (Ferreira, 2006a).

ability to combine different genes, as one gene can be the input (in default ADF, genes can be only combined by functions).

3.2 EVOLVING POPULATION WITH REPRODUCTION

First-generation individuals may have different genotypes, phenotypes, and fitness scores. Although good individuals may be found in this generation, evolving is necessary to explore many spaces and exploit good patterns. This evolutionary process is called, like in the biological life, reproduction (Eiben et al., 2003).

The basic idea is to insert variation in the current generation to generate new individuals for the next. This variation can change one element from the individual (mutation), many elements (transposition), or even change them between individuals (recombination). Variation is mainly employed in the best individuals, with tournament selection and round-robin techniques, encouraging exploration of new horizons without losing the earning already gained. More detailed descriptions are present in the following subsections for each type of variation used in this work.

3.2.1 Inserting abnormalities with mutation

One of the most simple and powerful operators in evolutionary computation is mutation. The basic idea is to change elements of an individual randomly to add variability. Based on the biological mutation, its purpose is to insert abnormalities into the population to explore different patterns, which may direct to better models. Random change of elements from a genotype encourages exploration of new spaces and can also change the phenotype considerably. Generally, mutations are unbiased, with uniform probabilities of selecting a random element (Eiben et al., 2003).

Mutation in GEP occurs similarly to other EC approaches, being applied to any chromosome position with any element, provided the latter is from the respective pool (function and terminals for the head and terminal only for the tail). Drastically changes in the expression tree structure can be seen with mutations in the head part (Ferreira, 2006b). For example, if a function of arity = 2 is mutated to a terminal, the branch is cut off. The inverse is also possible by growing the expression tree with an arity bigger than the current one or replacing terminal elements.

3.2.2 Inherit information from both parents with the recombination operator

Another interesting variational operator is the recombination operator. Unlike the previous ones, the crossover operator uses genetic information from two parents (or more, in some approaches), swapping a subsequence from one parent to another in the same position. This exchange is also based on the biological environment, in which two parents exchange information to generate a fitter individual. The intent in evolutionary computation is to combine parts of suitable individuals to check if this new combination produces better results (Eiben et al., 2003). There are many types of recombinations in GEP terms (not exclusively), but, in this work, only two are used: one-point and two-point.

In the first case, a random position between elements is chosen. For individual A, the subsequence to the left of this position is copied to a new child C to the same position, and the subsequence to the right is copied to a new child D (also to the same position). The exact process is employed for individual B except that the left subsequence is copied to child D and the right subsequence to child C.

The second type, two-point recombination, has a similar process. For it, not only one but two positions between elements are randomly chosen. Considering these positions as X and Y, respectively, the elements of an individual between these positions X and Y are copied to a new child combined with the subsequences to the left of X and the right of Y from another parent.

3.3 PICKING THE BEST INDIVIDUALS FOR REPRODUCTION: PARENT SELECTION

After population evaluation, a reproduction step is executed to generate offspring from the population, which individuals will act as parents influence the quality and variability of the children. Policies that improve overall fitness without losing the innovation of new individuals are necessary to guide the search to better spaces. One that focuses too much on quality may guide and stuck the search to a local optimum. In cases that focus too much on innovations (variability), the search may be similar to a random search, i.e., information from previous generations would be useless.

3.3.1 Tournament Selection

The Tournament Selection (TS) picks an individual based on his fitness. As the name says, a tournament is realized to select the fittest individual. From a population with N individuals, k (< N) are chosen randomly, and the individual with better fitness is selected. The main idea is to select the best individuals and let less fit (and maybe innovated) individuals reproduce. One thing to note is that the $k - 1$ individuals are never selected to reproduce.

3.4 SURVIVOR SELECTION MECHANISM

After evaluating the population, which compromises evaluating the current generation, parent selection to generate children, and evaluation of the latter, it is necessary to select the individuals to pass to the next generation. The main motivations are to stabilize the population size and forward only relevant individuals. If no individual is removed, population size may grow, so it is not feasible for evaluation, showing the necessity to remove some individuals.

As for what individuals to be removed, it enters in the relevance for further reproduction. The fittest individuals and individuals that, although have worse fitness, may present innovated

patterns that will contribute considerably to improve children from further generations. Some approaches commonly used are described below:

3.4.1 Age-Based Replacement

Age-based replacement focus on preserving young individuals even if their fitness is lower than the old ones. Different rules may be applied to remove the oldest individual or ones that have achieved a certain number of generations at each generation. This approach opens space for newer individuals to be evaluated and improved.

Removal of the oldest individual guarantees that at least one child will pass to the next generation. Also, focus on evaluating recent individuals, always setting a limited number of generations to evaluate each individual. A more aggressive approach is to set the lifetime limit of individuals manually. In this case, it is guaranteed that no individual will be evaluated at certain times.

When individuals have weights to be optimized, too old ones may have much higher fitness than the overall population, and fitness in followed generations may not change considerably. It is ideal for removing these individuals when they reach a state with little or no improvement to address this pattern.

3.4.2 Fitness-Based Replacement

The remove-worst policy aims to remove N individuals with worst fitness since they produce minor relevance to solve a specific problem (also in being a parent for further generations). Generally, the N value is not so high compared with the current number of individuals in the population, since otherwise would remove variability between individuals (few parents with similar structures may produce similar children, focusing on a small local search).

In elitism, the fittest individual survives to the next generation, always providing good material for further reproduction. Even if produced children would obtain lower fitness, there is always change in further generations to produce better individuals than this one.

3.5 CONCLUSION

In this chapter, concepts regarding GEP were discussed. The genotype/phenotype representation, replacement, and reproduction are some of the topics explained. Different adaptations to these methods are made and discussed in our methodology.

4 RELATED WORKS

This chapter presents the diverse architecture search works developed (from 2016 onwards and mostly accepted in renowned conferences and journals), mainly for convnet generation. The CIFAR datasets are our focus since they are evaluated in our proposed thesis. Then, we present an overall discussion about NAS proposals, analyzing their pros, cons, and what can be treated in the future. Finally, conventional and NAS-based approaches for medical semantic segmentation were studied, evaluating their results on the CHAOS challenge. Thus, we introduce this chapter describing the CIFAR datasets and the CHAOS challenge.

4.1 DATASETS

In this section, the datasets tested in our thesis are described. The CIFAR datasets and the CHAOS challenge are discussed below.

4.1.1 CIFAR datasets

The CIFAR[1] (Krizhevsky and Hinton, 2009) datasets are very popular in works involving image classification and NAS approaches. They are labeled subsets of the Tiny Image dataset (which consists of 80 million images).

There are two datasets: CIFAR-10 and CIFAR-100, which the former consist of images from ten classes and, for the latter, one hundred. Each dataset is composed of 60000 images, divided into 50000 for training and 10000 for testing. Also, the number of samples is divided equally between classes (6000 for each class in CIFAR-10).

The CIFAR-10 training subset (45k for training and 5k for validation) was employed to evaluate our proposal in most experiments. The testing subset will be used in further experiments to evade over-fitting to the test subset. Its classes consist of: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck images (examples are shown in Figure 4.1).

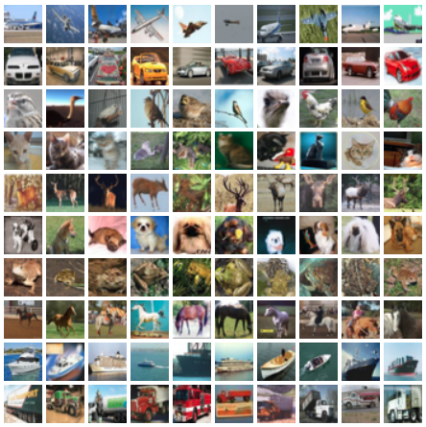


Figure 4.1: Examples of samples from CIFAR-10 dataset.

The CIFAR-100 dataset was also evaluated. It consists of 100 classes, which are separated into the following superclasses: aquatic mammals, fish, flowers, food containers, fruit and vegetables, household electrical devices, household furniture, insects, large carnivores,

[1] https://www.cs.toronto.edu/~kriz/cifar.html

large man-made outdoor things, large natural outdoor scenes, large omnivores and herbivores, medium-sized mammals, non-insect invertebrates, people, reptiles, small mammals, trees and vehicles. Besides a more significant number of classes, each class in CIFAR-100 dataset has 600 samples (500 for training and 100 for testing).

4.1.2 CHAOS challenge

To provide a robust evaluation of automated semantic segmentation approaches on abdominal images, the Combined Healthy Abdominal Organ Segmentation (CHAOS) challenge (Kavur et al., 2021) was developed. The developed dataset (available at (Kavur et al., 2019)) contains CT and MRI data that multiple experts annotated. Then a consensus was created for the final ground truth. The organs to be segmented are the liver, left kidney, right kidney, and spleen. Examples can be seen in Figure 4.2.

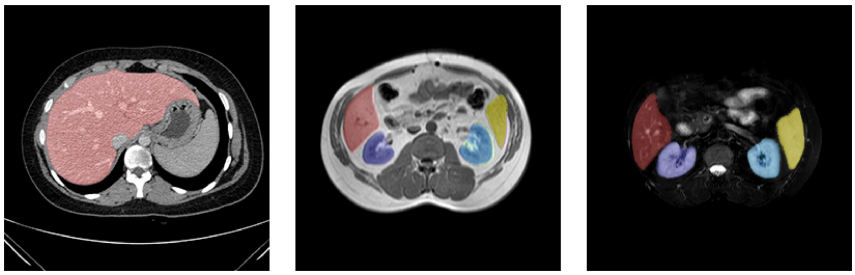


Figure 4.2: Examples of different segmented organs from CHAOS dataset. From (Kavur et al., 2021).

CHAOS challenge involves the segmentation of four organs with cross-modality (CT and MRI) and multi-modal (two pulse sequences for MRI). Forty patients went through a CT scan, and another 40 patients went through MRI. In both cases, neither organ had some complications (i.e., they are healthy cases). Both CT and MRI subsets were grouped in 20 patients for training and 20 patients for testing. The ground truth of the testing subsets is not publicly available. The segmented subset must be uploaded to the Grand-Challenge[2] website for the metrics to be computed.

The CT scans have an in-plane resolution of 512×512 (which is very common for CT slices). Exams' number of slices is between $[78 - 294]$, with an average of 160 slices and a total of 6407 slices for the entire CT dataset.

Each patient had three different exams for the MRI dataset: T1-DUAL in-phase, opposed-phase, and T2-SPIR. The first two were registered (thus sharing the same ground truth). Thus, there is a total of 120 exams. The number of slices between $[26 - 50]$, with an average of 32 and a total of 3868 for pulse (total of three pulses). Slice resolution is 256×256. One thing to note is that it is expected to submit only one prediction for T1 and another one for T2-SPIR. For T1, both phases can be used simultaneously, creating a two-channel input.

4.2 NAS APPROACHES

As discussed previously in the introduction, works in the field of neural architecture search can be categorized in what kind of search space, search strategy, and performance estimation strategy are employed, following the study realized by Hutter et al. (2018). First, we like to group the recent and studied NAS approaches into their search space representation, then discuss the latter dimensions.

[2]The website can be accessed at `https://chaos.grand-challenge.org/`

Superficially, search space representations can be divided into chain-structured and multi-branched networks. The former representation works similar to VGG-Net, where one layer is the input of another, forming a sequential network (e.g., NAS approaches such as (Zoph and Le, 2016; Baker et al., 2016; Suganuma et al., 2017; Cai et al., 2018a)). As for the latter type, networks like the Inception model can be categorized, with different branches joined together. A residual network, although having a structure similar to a chain-structured network, can be categorized as a (simple) multi-branched network with its skip connections (as in (Brock et al., 2017; Real et al., 2017; Liu et al., 2017b; Cai et al., 2018a,b)).

Initial studies tried different structures for the whole model, which may increase complexity in finding an optimized architecture as too many choices are found. Thus, studies were proposed to limit the search space using the concept of cells (also called motifs in some works). The idea of a cell is to represent a convolutional block and then replicate this block across the network, similar to hand-crafted networks (e.g., a residual block is replicated, changing only the number of channels and feature map size depending on which part is placed). This approach considerably limits the choices to be made, focus only on a specific cell to be replicated.

Many works introduced this policy, such as (Real et al., 2017, 2018), with also implementing cell types: normal and reduction ones. These cells are composed of small modules that contain two operations combined with the summation, then these blocks (generally between four and five) are combined with concatenation. The only real difference is that in the latter, feature map reduction is applied. Similar to these approaches, in the hierarchical NAS proposed in (Liu et al., 2017b), small combinations (called low-level motifs) of operations (e.g., convolutions, poolings) were combined (more freely than the previous cell implementation).

In the case of search strategies, most works are categorized as in evolutionary computation ((Miller et al., 1989; Yao, 1999; Stanley and Miikkulainen, 2002; Floreano et al., 2008; Miikkulainen et al., 2017; Real et al., 2017, 2018; Lorenzo and Nalepa, 2018; Lu et al., 2018; Prellberg and Kramer, 2018; Elsken et al., 2019)), reinforcement learning ((Zoph and Le, 2016; Baker et al., 2016; Zoph et al., 2017; Zhong et al., 2018)), Bayesian optimization ((Kandasamy et al., 2018; Zela et al., 2018)), random search ((Liu et al., 2017b; Real et al., 2018)) and gradient-based methods ((Liu et al., 2018; Xie et al., 2018)) (Hutter et al., 2018).

Neural architecture search may need a large number of GPU time to try different combinations of structures until a satisfactory one may be found. Hence, several strategies were proposed to mitigate search and resource consumption. One of the most straightforward strategies is weight inheritance. As the name says, the purpose is to pass the trained weights from an old model to another new one. Then, training each model from scratch may not be necessary to evaluate them. This policy is based on the concept of transfer learning, where already (or almost) trained models would achieve superior results than others with random weights.

Other techniques to decrease each model's training time are commonly used in hand-crafted networks, such as learning rate schedulers. One fast and straightforward approach is the usage of the SGDR technique with cosine annealing, proposed in (Loshchilov and Hutter, 2016). Although warm restart is not employed, cosine annealing for learning rate decay is commonly used for generally 20 epochs, achieving high accuracy in little time. It was employed in different works (e.g., (Elsken et al., 2017; Brock et al., 2017; Real et al., 2017; Lu et al., 2018; Elsken et al., 2019; Real et al., 2018)) to increase the number of models to be evaluated with also choosing models with a higher probability of being good ones (as this kind of approach inserts good and fast convergence in training). Also, data augmentation such as mean/standard deviation normalization, random cropping, CutOut, and Auto-Augment policies may be included to reduce over-fitting models. Following, we like to present and describe the recent and promising NAS approaches studied, ordered by year:

4.2.1 Neural Architecture Search with Reinforcement Learning (2016)

Zoph and Le (2016) proposed a recurrent neural network (RNN) trained with the policy gradient approach (REINFORCE rule), called Neural Architecture Search (NAS), to find out the best models for the CIFAR-10 (convnet) and Penn Treebank (LSTM-like network) datasets.

An RNN controller is employed to generate blocks (e.g., convolution) and their hyperparameters (e.g., number of filters). The latter are generated as a sequence of tokens, being this sequence a representation of one specific layer. The network is composed of many sequences as the entire CNN model. Also, a parameter-server scheme was used to improve training speed.

Using baseline data augmentation, the best CNN model achieved a 3.65% error rate in the CIFAR-10 dataset. In the Penn Treebank, a test set perplexity of 62.4 was obtained. Also, transfer learning was employed in the character language modeling task, with a perplexity of 1.214.

4.2.2 Designing Neural Network Architectures using Reinforcement Learning (2016)

Using the Q-learning reinforcement learning approach, Baker et al. (2016) proposed a framework called MetaQNN for the generation of chain-structure convnets. It involves a ϵ-greedy exploration strategy with experience replay. First, the value of ϵ is equal to one for exploring new models. After hundreds of iterations, this value is decreased until 0.1 to force exploitation and increase accuracy. Besides, rewards are based on validation accuracy.

Architecture search was executed using 10 NVIDIA GPUs between 8 and 10 days. In CIFAR datasets, global contrast normalization, random mirror, and random translation were applied to the images. The best error rate in CIFAR-10 was 6.92% and 27.14% for CIFAR-100.

4.2.3 Convolutional Neural Fabrics (2016)

Saxena and Verbeek (2016) developed the Convolutional Neural Fabrics (CNF), a 3D connection representation of response maps containing the values of the neural network hyperparameters. These response maps are locally connected based on the layer position, feature map size, and the number of feature maps.

Experiments on classification (CIFAR-10, MNIST) and segmentation (Part Labels) were conducted to assess CNR's versatility. In classification, the best models achieved error rates of 7.43% and 0.33% on CIFAR-10 and MNIST datasets, respectively. As for the segmentation step, two accuracies were presented: superpixel and pixel-based, which attained rates of 95.63% and 94.82%.

4.2.4 Hierarchical Representations for Efficient Architecture Search (2017)

Their search strategy was based on random search and evolutionary search (tournament selection). Liu et al. (2017b) proposed a hierarchical search space representation, where it is composed of basic motifs, and then these motifs are combined to generate the network cell. A set of mutation operators are applied to create diversity (also in the population creation).

Experiments were executed using 200 GPUs, with 1 hour (random search with over 200 architectures trained) and 1.5 days (evolutionary search) in the CIFAR-10 dataset. Best models were found using evolutionary search, with an error rate of 3.63% ± 0.10% in CIFAR-10, and 20.3% (top-1) and 5.2% (top-5) errors with the cell transferred to ImageNet. With random search, CIFAR-10 errors were 4.04% ± 0.2% and 3.91% ± 0.15% (long run), showing the capability of this representation even with a simple search strategy.

4.2.5 Large-Scale Evolution of Image Classifiers (2017)

The evolutionary-based neural architecture search work proposed by Real et al. (2017) focused on an ample search space combined with simple evolutionary techniques, like mutation, to achieve a 5.4% error rate on the CIFAR-10 and 23% on the CIFAR-100.

The idea is to found out an optimized model without human intervention. This approach employs the complexifying strategy, an evolutionary process applied to simple initial models like the NEAT algorithm. At each generation, their structural complexity increases with mutation and recombination, as necessary.

The generated models in each generation are refined with a tournament selection approach. Then children are reproduced through mutation and recombination to achieve higher fitness. Also, if possible, these children inherit the parent's weights (mostly from convolutional blocks). The main problem in this approach is the high and expensive resource usage in training the many individuals generated.

4.2.6 Evolving Deep Neural Networks (2017)

Another NEAT-based work is the CoDeepNEAT, proposed by Miikkulainen et al. (2017), which extends NEAT with co-evolutionary optimization of components. In this approach, two components, blueprints, and modules are evolved separately. For the CIFAR-10 dataset, initial populations of 25 blueprints and 45 modules are generated, and from these populations, 100 convolutional models are constructed and trained for each generation.

The best CIFAR-10 model achieved 7.3% error, and the generated LSTM model for Penn Treebank improved by 5% if compared with vanilla LSTM. One thing to note is the fast convergence of the best models, caused by the approach's focus on obtaining models with good fitness right from the start. The idea behind modules and blueprints is to evolve substructures (modules) and their relationships/connections (blueprints) separately, since in some cases, one or another may be the cause of suboptimum (or good) fitness. This strategy may kill better models with slow convergence.

4.2.7 Simple and Efficient Architecture Search for Convolutional Neural Networks (2017)

Aiming to generate neural architectures without a large consumption of resources like some evolutionary approaches and attaining satisfactory accuracy, Elsken et al. (2017) proposed the Neural Architecture Search by Hillclimbing (NASH). They used network morphisms combined with hill-climbing to produce, at each step, various modifications and pass on the best model to the next step. Error rates in CIFAR-10 were down to 4.4%, with an ensemble model, and 19.6% for CIFAR-100.

4.2.8 Learning Transferable Architectures for Scalable Image Recognition (2017)

Zoph et al. (2017) proposed a new search space, called NASNet search space, combined with cell search, intent on finding out the best cells in CIFAR-10 and execute transfer learning, only in the architecture, to the ImageNet dataset. Besides, they introduced the scheduled drop-path, which aims to drop connections in the cells at some probability, and this value is linearly increased as the training occurs.

Two types of cells are being searched: normal and reduction (can be seen in Figure 4.3). They are similar, except that reduction cells reduce the feature map size (with poolings or strided convolutions), and normal cells are generally repeated successively. Convolutional models are then created with these two cells, which are generated with a controller RNN (trained

with Proximal Policy Optimization), predicting their blocks like (Liu et al., 2017a). Operations in two previous inputs are applied in the cell, in which the normal cells are stacked N times, and reduction cells (generally only one) are included between two stacks of normal cells. Like in (Zoph and Le, 2016), reinforcement learning is used for operation and input decisions in the blocks.

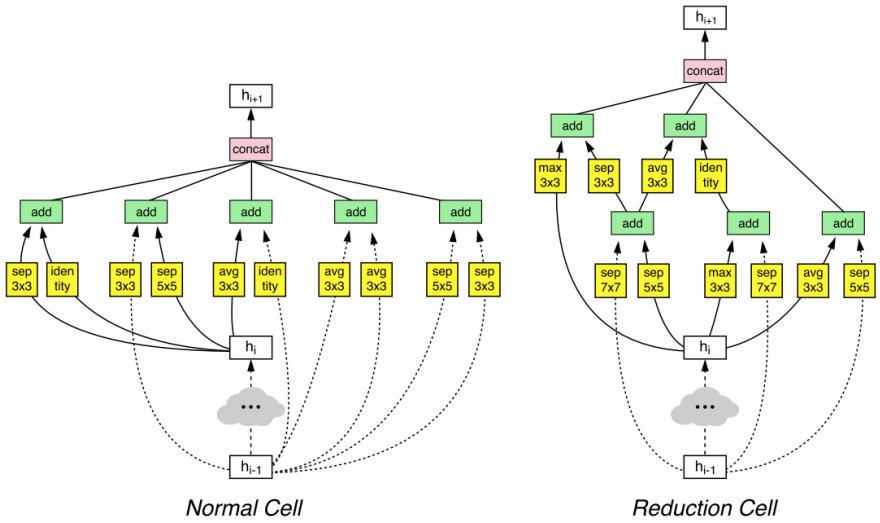


Figure 4.3: Normal and reductions cells from a NASNet model. From (Zoph et al., 2017).

With a work queue of 500 Nvidia P100 GPUs for four days, this approach takes two thousand GPU hours (approximately 83 GPU days) to obtain the best models. Best mean error rates (across five runs) of 2.97%, for baseline augmentation, and 2.40%, for baseline plus CutOut (best run was 2.19%), were found with the NASNet-A model in CIFAR-10 at the cost of 27.6 million parameters). The same NASNet-A was slightly adapted to the ImageNet (more than 88 million parameters), achieving Top-1 error of 17.3% and Top-5 error of 3.8%. In a mobile-constrained setting, Top-1 error of 26% and Top-5 error of 8.4% with 8.3 million parameters. They also analyzed object detection performance on the COCO's *mini-val* and *test-dev* datasets, obtaining mean average precision (mAP) of 43.2% and 29.6% (mobile) for *mini-val* and 43.1% for *test-dev*.

4.2.9 Progressive Neural Architecture Search (2017)

Liu et al. (2017a) developed the Progressive Neural Architecture Search (PNAS), which progressively increases the complexity of the generated modules, achieving similar accuracies than NASNet with lesser resource consumption.

The authors of these modules, called cells, are improved and replicated several times to create the convolutional network. A cell is composed of blocks, combining two operations applied separately to some inputs and joined by another operation (i.e., addition). The method begins training K cells with one block and then expands to K cells with two blocks. K cells are selected by using a surrogate function. Then, they are trained. This process is executed until cells with B blocks are created.

Experiments in CIFAR-10 (3.41% ± 0.09% error rate) and ImageNet, both in normal (Top-1 17.1% and Top-5 3.8% error) and mobile (Top-1 25.8% and Top-5 8.1% error) settings, showed satisfactory results in this methodology. The best cell found by the proposed work with the networks for CIFAR-10 and ImageNet can be seen in Figure 4.4.

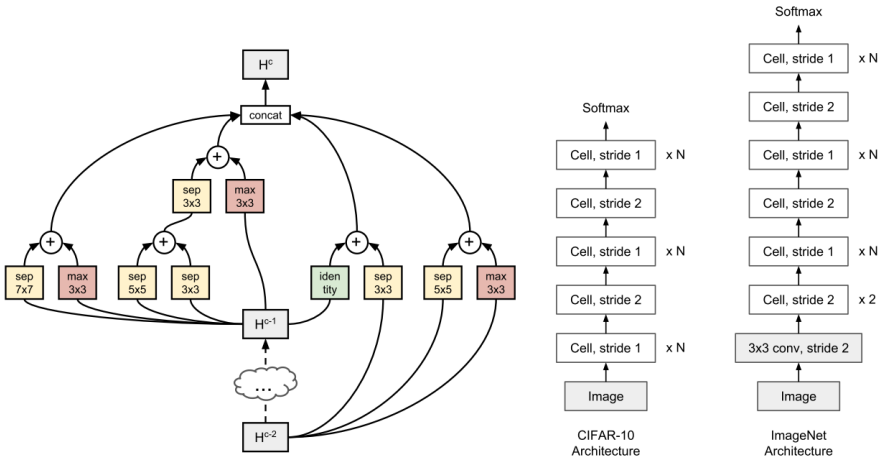


Figure 4.4: The best cell found with the CIFAR-10 and ImageNet architectures. From (Liu et al., 2017a).

4.2.10 A Genetic Programming Approach to Designing Convolutional Neural Network Architectures (2017)

Suganuma et al. (2017) designed a Cartesian genetic programming (CGP) for CNN architecture search. The programs are represented as directed acyclic graphs in a two-dimensional grid where, between the inputs and outputs, there is $N \times M$ intermediate nodes with some function (e.g., convolutions and poolings with different proprieties, summation), being N the number of rows and M the number of columns. These nodes can have two states: active and inactive, which will result in this appearance or not in the real CNN model (phenotype).

The genotype in CGP is a collection of 3-tuples, where each one of them maps a node with its function and two inputs, except the input (which has no inputs) and the output (with only one input). If a specific node is not included in some tuple as an input of another node, it will not be expressed in the phenotype, being inactive for the current individual. In Figure 4.5, a (2×3)-grid genotype is shown with its respectively phenotype representation. One thing to note is that in node 6, a max-pooling is included to normalize tensor sizes.

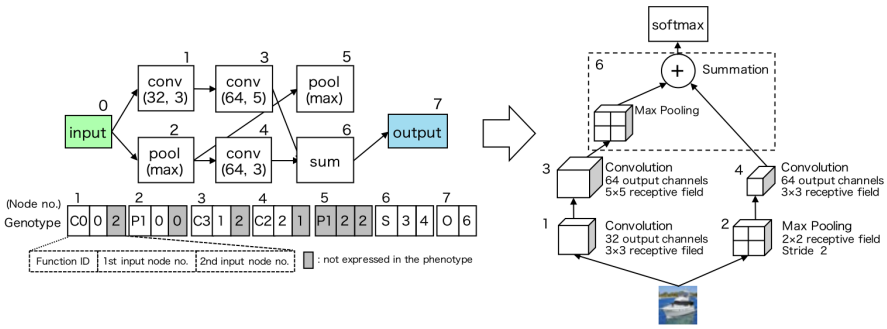


Figure 4.5: Genotype and phenotype representations in CGP. From (Suganuma et al., 2017).

For experiments, the CIFAR-10 dataset was employed. Two architecture searches were executed, using ConvBlocks (without residual information in the block) and ResBlocks (residual information activated). In the best models found, the ResSet approach had the lowest error, with a value of 5.66% against 5.80% from the ConvSet approach. The number of parameters was around 2.84 and 1.50 million, respectively. Using two NVIDIA TITAN X GPUs, these models were discovered between 10 and 12 days.

4.2.11 Efficient Architecture Search by Network Transformation (2018)

Following different approaches using network transformation, Cai et al. (2018a) proposed the Efficient Architecture Search (EAS) framework, which encompasses many transformation operators (Net2Net operations and function-preserving transformation, for weight reuse), which are chosen based on a reinforcement learning-based meta-controller. Bidirectional LSTM is employed for variable-size network handling and a broad vision of the network for modification in its structure. Furthermore, the framework is adapted to generate models based on the complex DenseNet architecture space.

Three main experiments were executed on the CIFAR-10 dataset (using mean+std correction and random cropping): architecture search on a small scale which uses a simple and small network as start-point; more extensive exploration using the best models from the previous experiment as start-points; and search on DenseNet architecture space using the DenseNet-BC model as its start-point. In the first experiment, after 300 networks were sampled, the best one is trained for 100 epochs, then used to be the start-point for the second stage, which is done similarly to the first stage. The best model was found and trained using 5 NVIDIA GTX 1080 GPUs in less than two days, with an error rate of 4.89%. As for the second experiment, around two days were necessary to obtain an error rate of 4.23%. Then, using the DenseNet space, a model with 3.44% error was found, significantly improving the results compared with the previous search space.

4.2.12 Practical Block-wise Neural Network Architecture Generation (2018)

Zhong et al. (2018) proposed an asynchronous Q-learning framework to generate Inception-like block CNNs (with also skip-connections), which is also an early-stop strategy. For block representation, they propose the Network Structure Code (NSC). This five-dimensional vector stores the following information about an operation: index, type (e.g., convolution, pooling, identity, concatenation), kernel size (if applied), and indexes of two previous operations to be used as inputs (only addition and concatenation use the two indexes). In Figure 4.6 we can see the best blocks found.

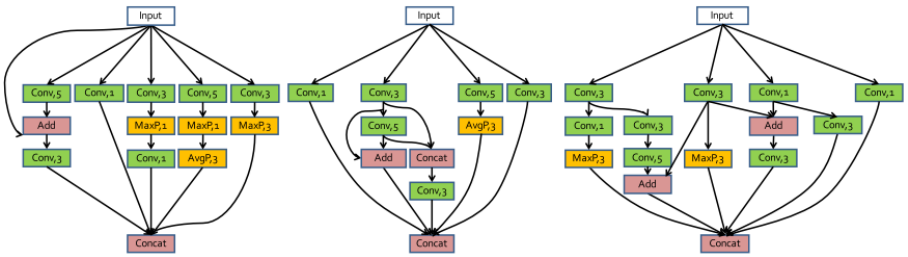


Figure 4.6: Best generated Block-QNN cells. From (Zhong et al., 2018).

This NSC is the state from Q-learning, in which the actions are the decision to make for further operations, which then constructs a convolutional block for a CNN model. To train the agent, Q-learning with epsilon-greedy strategy and experience replay was employed. Besides, validation accuracy is used in the model reward. An early analysis of early stopped models showed that some good models might have inferior accuracy than bad ones if prematurely stopped. They also verified that their final accuracy negatively correlates to FLOPs (FLoating-point Operations Per Second) and density (number of parameters). So, they proposed an early-stopping strategy based on these two values to redefine the reward function from Q-learning, decreasing the difference between the redefined reward and the final accuracy considerably.

Architecture search was executed based on the CIFAR-100 dataset, using 32 GPUs for three days (96 GPU days). Different from most NAS approaches, fewer epochs were used in training candidate models (12 epochs). Also, no data augmentation was employed. Block transfer was applied to CIFAR-10 and ImageNet datasets. For both CIFAR datasets, the best error rates were 3.54% and 18.06% (almost 40 million parameters) for 10 and 100 classes, respectively.

4.2.13 Efficient Neural Architecture Search via Parameter Sharing (2018)

Seeing the bottleneck in training NAS models until convergence for accuracy evaluation, Pham et al. (2018) employed weight sharing in their Efficient Neural Architecture Search (ENAS) method. Models are generated using an LSTM controller with 100 hidden units which used softmax classifiers for sample decisions.

At 24 million parameters, the ENAS model for the Penn Treebank dataset managed to score a test perplexity of 55.8. As for the CIFAR-10 dataset, macro and micro searches were applied. The former was focused on convolutional models (structure similar to ResNet models). The latter focused on convolutional cells, which concatenated B blocks (summation of two transformations in some inputs).

In the macro search, the best model, which has 21.3 million parameters and was found in seven hours (using the NVIDIA GEFORCE 1080TI), achieved an error rate of 4.23%. An increase of features of this model (38 million parameters) improved the error rate to 3.87%. Improvements were shown in the micro space search in the error rate and parameters. In almost 12 hours, a model with 4.6 million parameters was obtained. This model achieved error rates of 3.54% and 2.89% (CutOut enabled). The proposed search combined with weight sharing contributed to reliable results without excessive resource consumption.

4.2.14 Regularized Evolution for Image Classifier Architecture Search (2018)

In AmoebaNet, Real et al. (2018) proposed a regularized evolution, in the tournament selection, to remove not the worst model from a sample but the oldest. As for other characteristics, their approach is based on (Zoph et al., 2017) (which they call their baseline study), mainly using the normal/reduction cells (an example of one of the best models can be seen in Figure 4.7).

A comparison between evolution and RL (from (Zoph et al., 2017)) and also between regularized and standard tournament selection was shown in the experiments, divided into small- and large-scale. For the former, grayscale versions of CIFAR-10 and ImageNet were employed beyond the MNIST dataset. Also, experiments were employed in CPU instead of GPU. Regarding the large-scale experiments, CIFAR-10 and baseline search space (SP-I) were used. Beyond the search space from the baseline study (SP-I), two other search spaces were analyzed: one which has more variability of operations (SP-II) and one with more complex tree structures (SP-III).

Overall, the mean accuracy of models in the regularized version was superior (through different hyper-parameters), encouraging the removal of the oldest, not the worst model in samples of the tournament selection step. Also, evolution was better than RL and random search (significant for the latter). The best model in CIFAR-10, found in the SP-II, achieved errors of 2.98% ± 0.05% for baseline augmentation and 2.13% ± 0.04% with also Cutout. For ImageNet, 16.9% for Top-1 and 3.7% for Top-5 in the normal setting and 24.3% for Top-1 and 7.6% for Top-5 for mobile setting. Although this approach showed excellent accuracy in the models found, it was by a high cost in resources (up to 3150 GPU days).

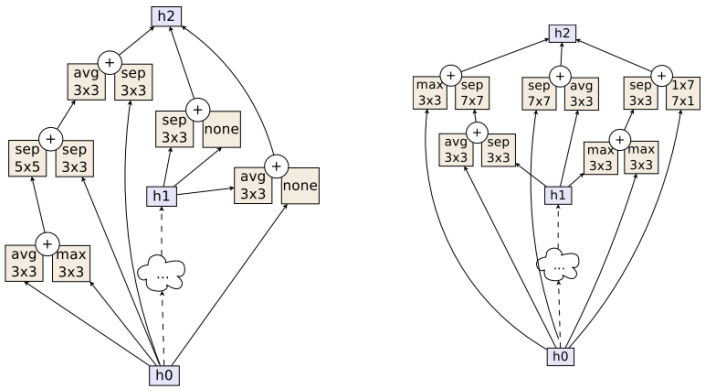


Figure 4.7: Normal and reduction cells from AmoebaNet, which are slightly more complex than the NAS ones. From (Real et al., 2018).

4.2.15 DARTS: Differentiable Architecture Search (2018)

Different from other works that apply a discrete and categorical optimization, Liu et al. (2018) proposed a continuous representation of the architecture, being optimized using gradient descent. They called it Differentiable Architecture Search (DARTS).

The cells in DARTS are directed acyclic graphs (DAG) of N nodes. Different operations connect these nodes (e.g., convolutions and poolings for convnets), being the former inputs or outputs of these operations. The last node concatenates the outputs of nodes without further connections. Each connection contains weights of the possible operations to be applied (with softmax), transforming the discrete representation into a continuous one and employing a gradient descent. The bi-level optimization problem (for the architecture search and operations) used an approximation that intercalates these two optimizations. Two types were proposed: first and second-order approximations.

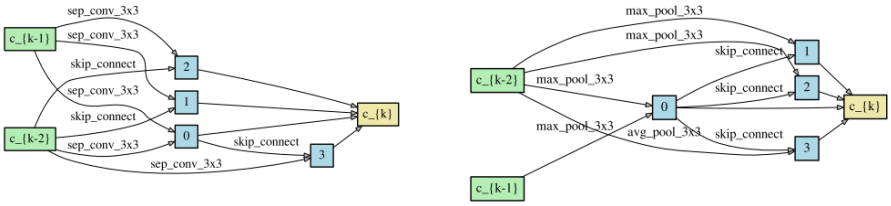


Figure 4.8: Normal and reduction cells found out by the DARTS approach. From (Liu et al., 2018).

The CIFAR-10, ImageNet, Penn Treebank, and WikiText-2 datasets were evaluated to assess the method's reliability. First, for CIFAR-10, models trained with Cutout obtained test errors of $3.00 \pm 0.14\%$ (first-order) and $2.76\% \pm 0.09\%$ (second-order), being found with 1.5 and 4 GPUs days, respectively. Test perplexity of the Penn Treebank was down to 58.3 (second-order). Transfer learning for the remaining datasets was employed, which had test errors of 26.9% for Top-1 and 9.0% for Top-5 in the ImageNet mobile setting, and for WikiText-2, a test perplexity of 66.9.

4.2.16 Neural Architecture Search with Bayesian Optimisation and Optimal Transport (2018)

An exciting idea for efficient distribution in the search space using model comparison was proposed in (Kandasamy et al., 2018). Kandasamy et al. (2018) developed the Optimal Transport Metrics for Architectures of Neural Networks (OTMANN), some rules based on optimal transport to evaluate the similarity between neural networks. Using this technique, one can take little steps in model transformation and take bigger ones to navigate to diverse locations in the search space. The overall structure of the neural network and features specifically for each operation present in the model are evaluated to determine the similarity between the two models.

They also proposed the "Neural Architecture Search with Bayesian Optimisation and Optimal Transport" (NASBOT) framework, including an evolutionary algorithm search and Bayesian optimization to generate neural networks efficiently.

Although results in CIFAR-10 were inferior in the state-of-the-art, with an error rate of 12.3% ± 0.3%, resource usage was considerably inferior. Four NVIDIA Titan Xp GPUs were used within seven hours. Also, applying the proposed distance metric in other methods may improve architecture search since the search space would be more sparsely navigated.

4.2.17 PPP-Net: Platform-aware Progressive Search for Pareto-optimal Neural Architectures (2018)

Based on the work in (Liu et al., 2017a), Dong et al. (2018) developed a multi-objective approach with Pareto Optimality to generate neural networks not only with high accuracy but also adapted to high constrained platforms (e.g., mobile devices).

The Platform-aware Progressive search for Pareto-optimal Net (PPP-Net) uses the Sequential Model-Based Optimization (SMBO) algorithm to search for an optimal model. The generated models alternate between normalization (BN only, BN+ReLU, and no operation) and convolutional layers (convs and group convs of kernels 1x1 and 3x3). The objectives to be attained were minimizing error rate, the number of parameters, FLOPS (floating-point operations per second), and inference time (in an NVIDIA Titan X GPU).

Experiments were executed using the CIFAR-10 dataset. The most balanced model was the PPP-Net-B with an error rate of 4.58%, 0.52M parameters, 63.5M FLOPS, and inference time of 0.025 seconds.

4.2.18 Towards Automated Deep Learning: Efficient Joint Neural Architecture and Hyperparameter Search (2018)

According to Zela et al. (2018), focus only on the neural architecture search may result in suboptimal best models, mainly if they are generated with few epochs as the correlation between models training with few and many epochs are minimal. Therefore, they proposed a combination of neural architecture search with hyperparameter optimization, which employs Bayesian Optimization and Hyper-band (Zela et al., 2018).

The own NAS step is treated as a categorical hyper-parameter optimization. The many parameters are efficiently explored with multivariate Kernel Density Estimators (probabilistic model from Bayesian optimization) and the Successive Halving subroutine (from Hyper-band). With 32 GPU days, they attained a test error of 3.18% ± 0.16 in the CIFAR-10 dataset with a three-hour budget.

4.2.19 Memetic Evolution of Deep Neural Networks (2018)

An evolutionary approach based on a memetic algorithm and Gaussian mutation for generation of segmentation and classification models was proposed by Lorenzo and Nalepa (2018) with a limited time budget, one hour for an MRI segmentation task and two hours for the CIFAR-10 dataset. It also implements the incremental technique, beginning with a one-layer model and increasing through time.

A symmetric matrix is employed to refer to the connections between feature maps, being only the top triangle populated with zeros and ones (if there is a connection or not). Each matrix refers to one individual. Then through the distribution of these individuals, mutation generates new candidates by executing the Gaussian process and then maximum expected improvement. Also, the crossover between matrices is used (even if they have different sizes).

Experiments were executed with and without Gaussian mutation in a more detailed setting for the MRI segmentation task, showing high population diversity and model compactness using Gaussian mutation without compromising the model fitness. Besides the high error rate in CIFAR-10 (27.73%), an emphasis on the execution time is necessary. Most works generate state-of-the-art models in GPU days or even weeks. In this case, only two hours were used to achieve this result. A more long-term training may show more satisfactory accuracies.

4.2.20 Path-Level Network Transformation for Efficient Architecture Search (2018)

Cai et al. (2018b) focused in path-level modifications rather than layer-level ones. Simple layers are transformed into a tree-like path (Figure 4.9) with also intending to preserve the weights already trained.

Their approach uses function-preserving network transformation for this preservation, more specifically, the Net2WiderNet and Net2DeeperNet operations from the Net2Net technique. They are transformed into the different convnet operations available (normal and separable convolution with different kernel sizes, e.g.). A tree-structure LSTM is employed to generated and improve the available models.

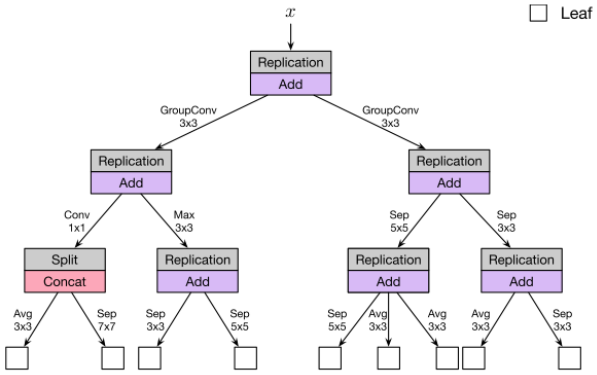


Figure 4.9: Tree-like representation of the best discovered cell by Cai et al. (2018b) approach.

Error rates of 2.30% in CIFAR-10 with a PyramidNet based model using 14.3M parameters (trained with DropPath and CutOut for 600 epochs) and 25.4% top-1 error in the ImageNet mobile setting with a CondenseNet based model were attained with the best-generated models.

4.2.21 Understanding and Simplifying One-Shot Architecture Search (2018)

In (Bender et al., 2018), a one-shot architecture search was proposed. Bender et al. (2018) generated a neural network with many different paths, which some are dropped, in prediction time, to assess the best architecture configuration. As we can see in Figure 4.10, a sequential network is constructed where some cells are placed (in blue). The controller controls each cell input, with some blocks to be chosen (and their inputs). In these blocks, several operations are placed to be forwarded (after training) to evaluate which combination of operations (block and cell-wise) can have superior results.

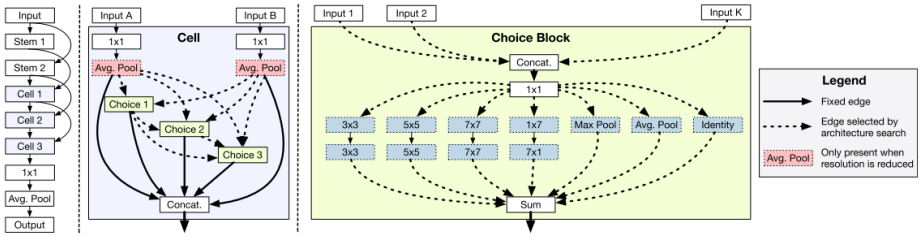


Figure 4.10: Worflow of the one-shot architecture search approach. From (Bender et al., 2018).

In CIFAR-10, the best search configuration achieved an average error rate of 3.9%±0.02% with the ten best-generated models and an average of 19.3M±0.6M parameters. As for the ImageNet dataset, 24.8% ± 0.4% error rate and 11.9M±1.5M parameters. One of the problems of this approach is the high need for memory to place the entire network, limiting the number of combinations of operations and blocks to be evaluated.

4.2.22 Efficient Multi-objective Neural Architecture Search via Lamarckian Evolution (2018)

Elsken et al. (2019) try to attend two main problems in NAS, which are the extensive resource optimization and production of models highly focused on predictive power without regards to performance. They proposed a multi-objective NAS approach called LEMONADE (Lamarckian Evolutionary algorithm for Multi-Objective Neural Architecture DEsign), based on Lamarckian evolution using modified network morphism operators to generate its models. Cells produced by this approach are not fixed by a determined structure (such as NAS search space-based approaches), as illustrated in Figure 4.11.

The network morphism operators are employed to increase the capacity of a network, for example, insert a block or increase the number of channels. These operators are an excellent approach to increase the performance of models but also insert higher resource consumption. To address this, they also adopted approximate network morphisms for capacity reduction (e.g., remove connections, decrease the number of channels). A Pareto-front population is maintained for parent selection and reproduction, and parents are selected with probability inversely proportional to their densities. Children's survival is also based on this, aiming to select parents and children with different resource requirements.

Architecture search was mainly employed in CIFAR-10, using SGD with cosine annealing where each model was trained for 20 epochs, learning rate decay from 0.01 to 0 and batch size equal to 64. The search was executed using 56 GPU days, albeit only significant changes in the first 24 GPU days. Models with different parameters and error rates were obtained: 8.9% for a model with 47K parameters, 5.5% for 190K, 4.6% for 882K, and 3.6% error rate for 3.4 million parameters.

Figure 4.11: Some of the cells found by the multi-objective NAS approach. From (Elsken et al., 2019).

4.2.23 SNAS: Stochastic Neural Architecture Search (2018)

In order to obtain architectures with a similar pipeline and search space to the NAS approach, albeit without consuming a considerable amount of resources, Xie et al. (2018) introduced SNAS (Stochastic Neural Architecture Search), a NAS approach that follows a similar idea from back-propagation to constructed customized models.

Like in DARTS, the connections between operations (edges) are continuous. To transform it to continuous, they use the concrete distribution approach to relax the discrete architecture distribution. Also, credit reward is done after training (delayed rewards), but using the training/testing loss during the process. Figure 4.12 illustrates the forward pass in SNAS. In this case, a cell with four nodes is connected. A matrix containing possible operations (including no operation) between these nodes controls how a cell will be represented.

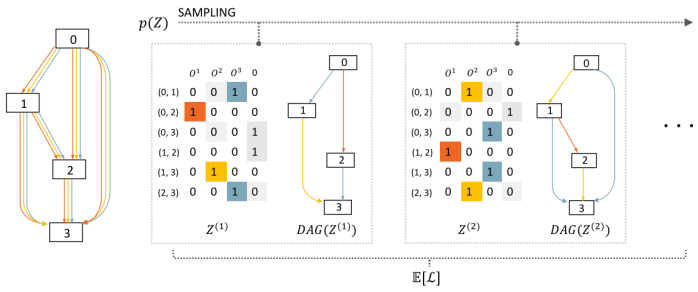


Figure 4.12: The forward pass in SNAS with a matrix representing which edges will be active. From (Xie et al., 2018).

A three-stage experiment pipeline was employed in SNAS, consisting of a stage where good convolutional cells were trained in a small network and chosen for the next stage. In the second stage, these cells were stacked to obtain a more extensive and better network. Then, the final stage consists of transferring learned cells in a small dataset (CIFAR-10) that may be reusable in larger datasets like ImageNet. In the first stage, an error rate in CIFAR-10 down to 9% was attained in 1.5 GPU days using the NVIDIA TITAN Xp GPUs. Using the same time, the best models (with 2.8M parameters) achieved a test error rate of 2.85 ± 0.02 with moderate constraint and CutOut data augmentation in the second stage.

4.2.24 sharpDARTS: Faster and More Accurate Differentiable Architecture Search (2019)

Another DARTS-based approach is the one developed in (Hundt et al., 2019). In this paper, a search space analysis is performed, and, after that, a custom block structure is proposed. Not only that, but a customized learning rate schedule called Cosine Power Annealing is also proposed. It aims to increase the learning rate curve descent, which, according to the authors, improves the validation error stabilization across epochs.

The SharpSepConv is the proposed block structure of this paper. It has two sequences of ReLU, followed by depthwise and pointwise convolutions, and then batch normalization. Every convolution operation (including dilated ones) follows this structure. Also, operations that increase the channels of intermediary convolutions were included. This structure contributed to improving overall results. Then, search is employed in a HyperCuboid search space-optimized using a Differentiable Hyperparameter Grid Search.

The sharpDARTS approach had satisfactory improvements compared to DARTS. Also, the search phase was executed between 0.8-1.2 GPU days on an NVIDIA RTX 2080Ti. For CIFAR-10, a validation error of 1.98% ± 0.07 was obtained and 5.8% ± 0.3 for CIFAR-10.1.

4.2.25 ProxylessNAS: Direct neural architecture search on target task and hardware (2020)

Differentiable NAS and other techniques based on the one-shot and supermodel approaches can suffer from high memory consumption because of the different possible pathways to create candidate models. Thus, these approaches would generally search within limited configurations (i.e., fewer combinations and model depth/wideness). In (Cai et al., 2018c), inactive connections are dropped (i.e., not used in memory) in each forward. Only active connections (i.e., connections from the current candidate) are processed, thus reducing memory usage. Gradient-descend and Reinforcement Learning are employed as their search strategies (though not jointly). They did not use a cell-stacking approach, but the entire architecture was searched. For the candidate network modeling, three stages of $B = 18$ blocks each are constructed. The number of output channels of the final block is $F = 400$.

Full models can be searched and evaluated without a proxy stage (search and full model training are executed with the same structure). With this in mind, Cai et al. (2018c) employed architecture search evaluation in different environments: GPU, CPU, and mobile. With little more than 8 GPU days, ProxylessNAS achieved a 2.08% error in the CIFAR-10 dataset with 5.7M parameters.

4.2.26 Searching for A Robust Neural Architecture in Four GPU Hours

Gradient-based search using Differentiable Architecture Sampler (GDAS) is based on the DARTS search space (cells as DAGs) and also employs a differentiable search (Dong and Yang, 2019). Different from DARTS, which optimizes the super-net entirely, GDAS sample one sub-graph at each training batch. This policy reduces competition between subnets and accelerates the search phase.

On an NVIDIA V100, the search phase is executed within four hours. Using a Titan 1080Ti, the search phase length increases to seven hours. The best results were found with a fixed reduction cell (i.e., only searching for the normal cell). On the CIFAR datasets with CutOut, error rates of 2.82% and 18.13% were obtained for 10 and 100 classes. GDAS obtained results similar to DARTS with only a few GPU hours of search.

4.2.27 FairNAS: Rethinking Evaluation Fairness of Weight Sharing Neural Architecture Search

Training in one-shot models may be biased with an unbalanced selection of operations. FairNAS (Chu et al., 2019) employs strict fairness, choosing operations in a balanced way. A training block contains several training steps. In each training step, operations are picked up only if they were not picked before in the current training block. Then loss is calculated, and weights are accumulated. After every operation is picked, a weight update is executed, and a new training block is started. Figure 4.13 illustrates this process.

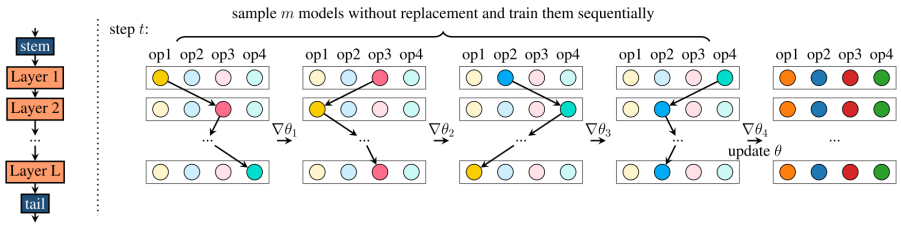


Figure 4.13: Illustration of the training process from the FairNAS approach. At each step, different operations are selected only once per training block. After the training block is finished, weights are updated. From (Chu et al., 2019).

The search phase is executed for 10 GPU days. A model is transferred from ImageNet to CIFAR datasets also using AutoAugment. FairNAS has a MobileNetV2-based search space, with different configurations for the inverted residual block was its possible operations. FairNAS has an error of 1.9% ± 0.1 and 13% ± 0.3 (between three models) for CIFAR-10 and CIFAR-100, respectively.

4.2.28 SGAS: Sequential Greedy Architecture Search

Sequential Greedy Architecture Search (SGAS) is another DARTS-based NAS approach (Li et al., 2020). SGAS approach transforms operation selection in a super-net (i.e., edge activation in a DAG) in sub-problems. After an operation is picked, other alternatives are pruned. Then, super-net optimization is executed until a stand-alone cell is generated (wherein each edge of an operation is fixed). SGAS aims to increase the correlation of the validation error in the search phase with the final test error in the training phase.

In CIFAR-10, SGAS had an error rate of 2.66% ± 0.24 with an average of 3.7M parameters per model between 10 independent runs. The search phase is executed in only six hours with an NVIDIA GTX 1080Ti. Overall, SGAS had superior results to DARTS with way less search cost. Also, Kendall's τ coefficient was higher than 0.4 for SGAS, showing a better correlation between search and training phase errors (DARTS had values of 0.16 and -0.29 for its strategies).

4.2.29 Noisy Differentiable Architecture Search

Another DARTS variation is the Noisy-DARTS (Chu et al., 2020). Since the inclusion of skip connections may introduce a premature convergence, Noisy-DARTS applies noise to the skip connections present in DARTS-cells to reduce bias.

Using seven GPU hours in the search phase with an NVIDIA V100, Noisy-DARTS obtained two models with error rates of 2.47% and 2.39% in the CIFAR-10 dataset. AutoAugment is also used in the data augmentation step. A model searched and trained on ImageNet was

transferred to CIFAR-10 and finetuned. This finetuned model achieved an error rate of 1.72%, explicitly showing a significant improvement when NAS is combined with transfer learning.

4.2.30 Evolving Neural Architecture Using One-Shot Model (2020)

Another one-shot approach, based on DARTS, is the genetic-based EvNas (Sinha and Chen, 2020). Instead of applying gradient descent to find out the best sub-graph, which later would be used as the candidate model, individuals are fetched from a two-dimensional matrix. Similar to DARTS, this matrix represents the many possibilities of connections between inputs and operations. Individuals are trained and then replaced as the search continues. Operations of mutation and crossover were applied, combined with elitism, to find out better candidates. Sinha and Chen (2020) stated that the stochastic nature of the evolutionary search present in this approach would prevent local minima. However, no further study elaborated this affirmation. Specific operations may be more trained in earlier generations, thus increasing the likelihood of newer individuals inheriting them).

EvNas used 4.4 GPU days from an NVIDIA RTX 2080 Ti and similar search and training configurations from DARTS for its search phase. The approach achieved an error rate of 2.47% and 16.37% on the CIFAR-10 and CIFAR-100 datasets with its best-found architecture. One thing to note is that many runs were executed for training one specific candidate, not searching. The other two candidates (with different seeds in the search process) had error rates of around 2.63%, and other search configurations were evaluated, which had errors around 2.80% for CIFAR-10. More search runs are encouraged to evaluate the search process.

4.2.31 Neural Architecture Transfer (2021)

Neural Architecture Transfer (NAT) is a multi-objective evolutionary-based NAS that combines architecture search with transfer learning (Lu et al., 2021). A super-net is trained by sampling subnets and training one for each batch, combined with weight sharing. The idea is to search for better structures in ImageNet to generate a model with pre-trained weights. This model can be finetuned in a small dataset to achieve state-of-the-art results. They applied a multi-objective search to find subnets with better trade-offs according to the defined objectives. In the end, a pre-trained super-net is optimized for a specific task (e.g., image classification) that can be used to find subnets (i.e., models) for specific datasets.

The overall process is employed in 1425 GPU hours on an NVIDIA RTX 2080Ti for ImageNet. For the CIFAR-10 dataset, an addition of 150 hours is used for finetuning. Four models with different parameters and MAdds were selected for each dataset. For CIFAR-10, models with parameters of 4.3M, 4.6M, 6.2M, and 6.9M were chosen, with errors of 2.6%, 2.1%, 1.8%, and 1.6%, respectively. As for CIFAR-100, models with 3.8M, 6.4M, 7.8M, and 9.0M parameters and error rate of 14%, 12.5%, 12.3%, and 11.7%, respectively. Figure 4.14 shows examples of two models (with resized input size of 224×224). Usage of NAS with transfer learning greatly improved the overall results. They also employed surrogate models to pick better models during the search. Although a higher correlation was found between the surrogate model score and the final accuracy, the improvement regarding NAT was not clear without surrogate models.

4.3 OVERALL DISCUSSION IN NAS

In Tables 4.1 and 4.2, a summary of the studied NAS approaches is shown with the results on the CIFAR dataset. Based on the works studied and also the review by (Hutter et al., 2018), some

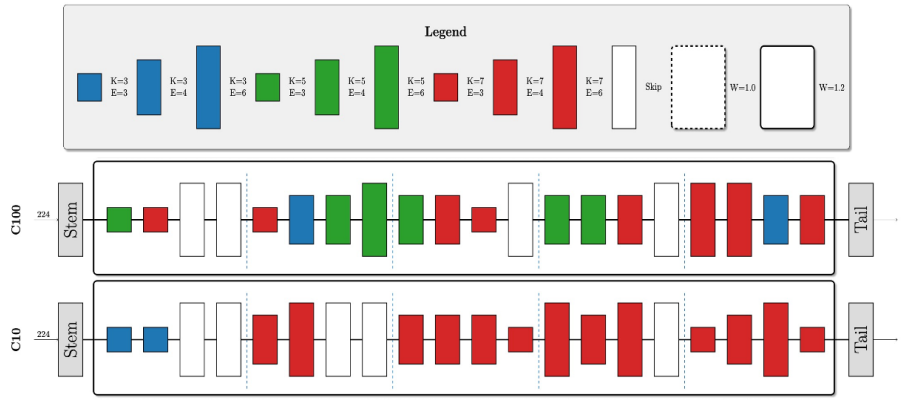


Figure 4.14: Examples of generated models for the CIFAR datasets. NAT uses the MobileNet search space, with only inverted residual blocks. Better visualization in color. From (Lu et al., 2021).

questions are discussed below. With this, a more general idea about the area of deep neural architecture search can be understood.

A trade-off between search space complexity and similarity to existing networks is seen in most works since searching in an ample space is not viable. However, simplification of this space may induce generated models very similar to hand-crafted networks. Ample space is not viable because of the computational cost to roam it, as an immense number of candidates will need to be evaluated. The search strategy employed will need to be efficient and highly intelligent to focus on relevant subspaces. Thus, reduction to a more viable and (probably) fitter space is generally applied.

The problem is when this search space is reduced to simple structures (such as the order of convolutions, activations, and batch normalization) that are simple enough for a human to design. New patterns can not be found, and it is unlikely that generated models will be better than hand-crafted ones with similar or more complex representations.

Usage of multi-branch blocks (i.e., cells) is repeated across the network, and hierarchical blocks are seen to address it. However, the set of operations (e.g., convolutions, activations) is limited to those that worked remarkably in hand-crafted models. Experiments in different elements may present innovative patterns to improve further the objective in question (e.g., increase accuracy, reduce memory consumption, optimize inference time).

Many models are generated and evaluated in evolutionary approaches to arrive at a good point in the search space. Information learned through generations (best parents) is used to refine the search and focus on better locals in the search space. Each model is trained separately, consuming GPU resources considerably. Network morphisms and weight sharing/inheritance play an essential role in reducing the training of each model (similar to transfer learning). Policies to reduce the training time of each model, but increasing the evaluations of models, may aid approaches that use too much GPU time to find out good models.

Table 4.1: Summary of surveyed NAS works with error (in %) for CIFAR-10 dataset, search phase time (in GPU days), and number of parameters (in millions) of the searched fully-trained model(s). * denotes results with a model using transfer learning (trained on ImageNet and finetuned on CIFAR-10). NR denotes for Not Reported.

Approach	**Search Space**	**Search Strategy**	**Error**	**Cost**	**# Param**
RNN & REINFORCE (Zoph and Le, 2016)	Sequential with skip-connection	Reinforcement Learning	3.65	1800	37.4
MetaQNN (Baker et al., 2016)	Sequential	Q-learning with ϵ-greedy exploration strategy	9.09 ± 1.68	80~100	3.71 ± 4.21
CNF (Saxena and Verbeek, 2016)	Sequential	Back-propagation	7.43	NR	21.2
Hierarchical Efficient AS (Liu et al., 2017b)	Hierarchical structures	Random/Evolutionary Search	3.63 ± 0.10	300	NR
NAS (Evolutionary) (Real et al., 2017)	Sequential with skip connection	Evolutionary search with complexifying	5.4	NR	5.4
CoDeepNEAT (Miikkulainen et al., 2017)	Sequential with skip-connection	Evolutionary search	7.3	NR	NR
NASH (Elsken et al., 2017)	Multi-branch (mostly sequential)	Hillclimbing	4.4	1	88
NASNet (Zoph et al., 2017)	Cell-based	RNN with Proximal Policy Optimization	2.40	83	27.6
PNAS (Liu et al., 2017a)	NASNet cell-based	Sequential model-based optimization	3.41 ± 0.09	17	3.2
CGP (Suganuma et al., 2017)	DAG	Cartesian Genetic Programming	5.66	20~24	1.50~2.84
AmoebaNet (Real et al., 2018)	NASNet cell-based	Evolutionary search (tournament selection)	2.13 ± 0.04	3150	34.9
PPP-Net (Dong et al., 2018)	Sequential	Pareto-optimal search with SMBO	4.36	NR	0.52
EAS (Cai et al., 2018a)	Sequential with skip-connection	Bidirectional LSTM with Network Morphism	3.44	< 10	10.7
BlockQNN (Zhong et al., 2018)	Inception-based blocks	Q-learning	3.54	96	39.8

Table 4.1: Summary of surveyed NAS works with error (in %) for CIFAR-10 dataset, search phase time (in GPU days), and number of parameters (in millions) of the searched fully-trained model(s). * denotes results with a model using transfer learning (trained on ImageNet and finetuned on CIFAR-10). NR denotes for Not Reported.

Approach	**Search Space**	**Search Strategy**	**Error**	**Cost**	**# Param**
Efficient NAS (Pham et al., 2018)	NASNet cell-based	LSTM and weight sharing	2.89	0.5	4.6
DARTS (Liu et al., 2018)	DAG	Gradient-based	2.76 ± 0.09	4	3.3
NASBOT (Kandasamy et al., 2018)	Sequential with skip-connection	BO and Optimal Transport	12.3 ± 0.3	1.17	NR
BOHB (Zela et al., 2018)	Sequential with skip-connection	BO and Hyper-band	3.18 ± 0.16	32	27.6
Memetic evolution (Lorenzo and Nalepa, 2018)	Sequential with skip-connection	Memetic evolution & Gaussian mutation	27.73	0.08	NR
PLNT (Cai et al., 2018b)	Tree-like cell	LSTM with Network Morphisms	2.30	8.33	14.3
One-shot AS (Bender et al., 2018)	Inception-like	One-shot & drop-path	3.9 ± 0.02	3.33	19.3 ± 0.6
ProxylessNAS (Cai et al., 2018c)	Tree-based	Gradient-based	2.08	8	5.7
SNAS (Xie et al., 2018)	DAG	Gradient-based	2.85 ± 0.02	3	2.8
sharpDARTS (Hundt et al., 2019)	DAG	Gradient-based	1.98 ± 0.07	1.2	3.6
LEMONADE (Elsken et al., 2019)	Multi-branch network	Multi-Objective with Lamarckian evolution	3.6	56	3.4
GDAS (Dong and Yang, 2019)	DAG	Gradient-based	2.82	0.17	2.5
FairNAS (Chu et al., 2019)	MobileNetV2	Balanced gradient-based	1.9 ± 0.1*	10	5.73 ± 0.15
SGAS (Li et al., 2020)	DAG	Gradient-based with sequential greedy search	2.66 ± 0.24	0.25	3.7
NoisyDARTS (Chu et al., 2020)	DAG	Gradient-based	2.47 & 1.72*	0.29	3.01 & 4.3
EvNas (Sinha and Chen, 2020)	DAG	Evolutionary-based	2.57 ± 0.09	4.4	3.6 ± 0.2
NAT (Lu et al., 2021)	MobileNetV2	Evolutionary-based with transfer learning	1.6*	65.62	6.9

Table 4.2: Surveyed NAS works applied to CIFAR-100 with error (in %), search phase time (in GPU days), and the number of parameters (in millions) of the searched fully-trained model(s). * denotes results with a model using transfer learning (trained on ImageNet and finetuned on CIFAR-100).

Approach	**Error**	**Cost**	**# Param**
MetaQNN (Baker et al., 2016)	27.14	80~100	11.18
NAS (Evolutionary) (Real et al., 2017)	23	NR	5.4
NASH (Elsken et al., 2017)	19.6	1	88
BlockQNN (Zhong et al., 2018)	18.06	96	39.8
GDAS (Dong and Yang, 2019)	18.13	0.17	2.5
FairNAS (Chu et al., 2019)	13 ± 0.3*	10	5.73 ± 0.15
NAT (Lu et al., 2021)	11.7*	65.62	6.9

Some approaches use the same structure to train small models in a large one, reducing the need to evaluate several models and speeding up the search stage. One-shot architectures, for example, train a super-net with different types of operations and then drops connections to verify the best combination of operations (i.e., subnet). Thus, in the end, the best sub-net is chosen as the best candidate model. One-shot gradient-based approaches use continuous representations (gradient-descent optimization) to search for the best representation, where edges are the many operations included in the approach. One-shot NAS may limit the search space to fixed combinations (i.e., a specific DAG). In some approaches, there is the limitation of inserting the entire network into GPU, reducing the many combinations (since they can not fit into a GPU). Newer approaches reduced this limitation by only loading in memory the active edges.

Combining the fast convergence of differential approaches with more diverse representations using reinforcement learning or evolutionary computation may optimize search. For example, induce gradient-descent optimization to representations given by the latter approaches, as they may present different but efficient locals in the search space than the ones already analyzed.

Validation accuracy is the primary parameter estimation strategy employed, using cosine annealing and changing the number of epochs (< 20) (and also model size) to better approximate the accuracy of the fully trained model without consuming GPU time exaggeratedly. Some works used weight inheritance to cut training time and arrive at stages in which the search stage's accuracy is closer to the accuracy in the real network training.

Regarding training time, it is also a problem in several NAS works, where too many GPUs were spent to find out the best architecture. One of the best approaches in CIFAR, AmoebaNet (Real et al., 2018), consumed more than three thousand GPU days to obtain the best model (with 2.13% test error). Although giant companies have the resources to execute this kind of experiment, it may not be viable for other groups.

Most cases only use CIFAR-10 as their case study, and no other dataset is evaluated to check if the approach will generalize to other problems. Few cases employ the search stage in ImageNet, CIFAR-100, or other datasets (including Penn Treebank), even when only transferring the model learned in CIFAR-10 to them. Also, NAS works focus on reducing the test error in a specific dataset. Some approaches applied architecture search on ImageNet for mobile settings. In the context of small devices, even works with FPGA (Field Programmable Gate Array) were implemented (Jiang et al., 2019). Multi-objective is being treated in NAS, with works like NAT

(Lu et al., 2021) employing search based not only on test error but the number of parameters and latency.

Image classification tasks are generally the primary focus on NAS approaches, using CIFAR-10 as their case study. Experiments with ImageNet and other datasets for image classification are also employed. In some cases, image segmentation (Liu et al., 2019a; Nekrasov et al., 2019a,b), detection (Chen et al., 2019) and text processing are evaluated, showing initial works treating different problems. Even problems related to the medical area (Mortazi and Bagci, 2018; Liu et al., 2019b) were studied for architecture search. Although these studies are already started, more experiments in different challenge datasets (in complexity and size) are necessary to be executed to evaluate the potential in NAS for a diversity of problems.

This brief study only analyzes a part of the NAS works developed in the last years. NAS is vastly researched, as we can see in the AutoML.org literature list[3]. Overall, the NAS field presents promising results in different areas, but the reason behind its improvements is yet to be explained. More robust and theoretical works are necessary for a better understanding of how NAS works and to guide researchers for further improvements in NAS (Ren et al., 2021). Different methodologies were developed to address different tasks based on the resources available. However, more constraints in GPU time and memory usage are necessary for these approaches to be viable in production. Also, employing different operations not used in standard deep learning techniques may be done with architecture search. Specific combinations of these operations not found by experts may present relevant improvements in accuracy, resource consumption, and prediction performance.

4.4 WORKS ON THE CHAOS CHALLENGE

This section briefly describes works employed in the CHAOS challenge to solve at least one of the five tasks. Most works were based on the U-Net model (Ronneberger et al., 2015). U-Net is a convolutional encoder-decoder model employed mainly in medical semantic segmentation. The encoder part is similar to other convolutional models, where the network progressive reduces the feature map to extract more global features. Then, in the decoder part, upsampling techniques (e.g., transposed convolution, bilinear upsampling) are employed to increase the feature map size to the input size. This model gained popularity for its efficient use even with a few samples, a common problem in medical images.

The team OvGUMEMoRIAL employed an attention U-Net variation of the work found in (Abraham and Khan, 2019). This attention U-Net model employs soft attention gates, Tversky loss, multi-scale pyramid input, and parametric ReLU. Adam optimizer is used for 120 epochs and a batch size of 256. A combination of U-Net, an hourglass network, and a convolutional autoencoder network was the proposal of the team ISDUE. Dice loss is employed. Also, Adam optimizer with an initial learning rate of 0.001 and batch size = 1. A 3D U-Net (Çiçek et al., 2016) with group normalization (Wu and He, 2018) was employed by the team Lachinov. They modified the original version to employed more complex skip connections and channel expansion (Kavur et al., 2021).

SUMNet (Nandamuri et al., 2019) is combined with a multitask adversarial learning strategy and batch normalization. Team IITKGP-KLIV is the only one in the original CHAOS challenge to employ an approach that does not use U-Net as its basis. Adam optimizer and cross-entropy loss were used to optimize the model. The team METU_MMLAB employs U-Net with a conditional adversarial network (CAN) aiding model training. Also, parametric ReLU is

[3]Which can be accessed at `https://www.automl.org/automl/literature-on-neural-architecture-search/`

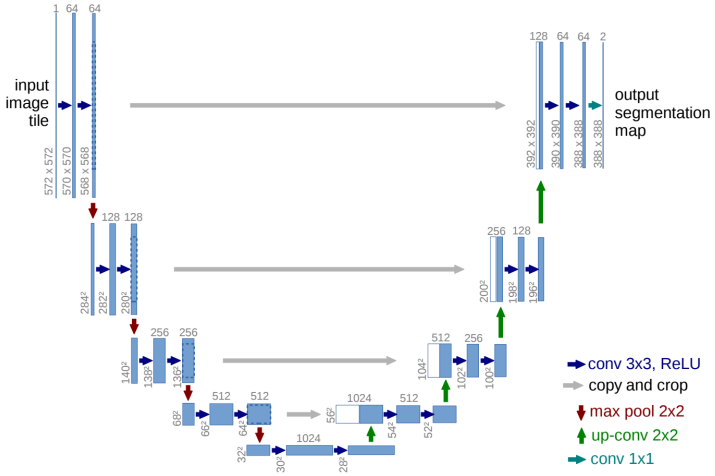


Figure 4.15: Illustration of the U-Net model. From (Ronneberger et al., 2015).

used. 3D connected component analysis is used as its post-processing step. U-Net was extended with a conditional generative adversarial network (Conze et al., 2021) by the team PKDIA. Adam optimizer is used with a learning rate of 1e-4. As for the loss function, they employed a Fuzzy Dice (Kavur et al., 2021).

One of the two ensembles used in the original challenge, the team MedianCHAOS employs an ensemble of five different U-Net variant models: DualTail-Net, TernausNet (Iglovikov and Shvets, 2018), LinkNet34 (Shvets et al., 2018), ResNet-50 and SE-Resnet50 variants. Besides DualTail-Net and LinkNet34, which only used soft Dice loss, the other three models were trained with a balanced combination of soft Dice and binary cross-entropy. A modification of the 3D network based on U-Net from (Han et al., 2019) was employed by the team Mountain. They adopted pre-activation blocks, instance normalization, and convolutions with striding. Adam optimizer with an initial learning rate of 1e-3 with Dice loss and batch size = 1. IVD-Net (Dolz et al., 2018), a U-Net variant with no dense connection between the encoder path and with modality dropout, was used by the team CIR_MPerkonigg. Adam, with a learning rate of 1e-3, was used(Kavur et al., 2021).

The nnU-Net (no new U-Net) method is a NAS approach focused on 3D semantic segmentation (Isensee et al., 2019). nnU-Net applies a set of rules focused on a dataset to obtain a good model. They train each candidate model with five-fold cross-validation and select the best model. They also apply data augmentation techniques strongly for better generalization. They employed an ensemble of a 2D U-Net and a 3D U-Net at full resolution. The training was not employed for Task 3 (LIVER-MRI), using the model trained in Task 5 (ABDOMEN-MRI) for liver segmentation.

NAS-UNet is a NAS approaches employed in the U-Net space using differentiable search (like DARTS) to find a good candidate with a combination of two cells for downsampling and upsampling (Weng et al., 2019). Search is employed in the PASCAL VOC 2012 dataset (Everingham et al., 2015). The cells have a similar structure to the DARTS cells. They included poolings (average and max) and convolutions (normal, separable, and dilated) as the possible operations in a cell. The authors also employed a squeeze-and-excitation module (which they call 'cweight'). They employed the ProxylessNAS training approach, which loads in memory only one operation per edge (in the graph), reducing memory consumption. Figure 4.16 illustrated the method's search space.

BiX-NAS is another NAS approach based on the U-Net model, although it is mainly based on a variant from the same authors called BiO-Net (Wang et al., 2021). They employed a

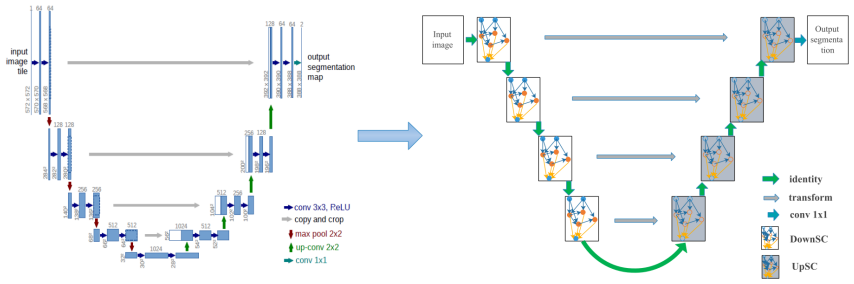


Figure 4.16: Illustration of the NAS-UNet search space. From (Weng et al., 2019).

bidirectional skip-connection approach, which reuses blocks using a recurrent approach to extract features with lower parameter usage efficiently. In Figure 4.17, an example of a generated model is present, also with a comparison between previous iterations. BiX-NAS has a two-phase search strategy. First, a trainable selection matrix with differentiable search is employed to reduce the explored search space. Then, a progressive evolutionary search is executed for exploitation. They focused on optimizing the macrostructure of its custom U-Net, mainly in how the different blocks are connected. This approach provides a dynamic multi-scale search that may be efficiently adapted to different scales and patterns images.

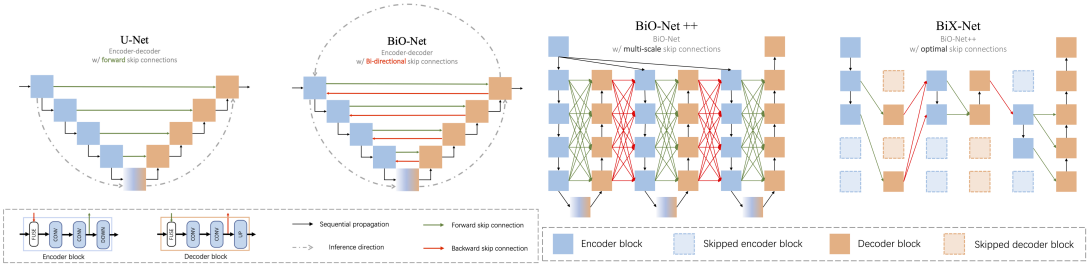


Figure 4.17: Illustration of the BiX-NAS search space. From `https://bionets.github.io/`.

AutoDeepLab (Liu et al., 2019a) is a gradient-based NAS that employs architecture search in the DeepLab search space. They employed the differential search strategy proposed in the DARTS approach and adapted it to optimize the cell structure and network architecture. Architecture search was employed in the CityScapes dataset within three GPU days on the NVIDIA P100. Figure 4.18 shows the architecture found in the CityScapes. Although no experiment was applied on the CHAOS challenge in the original paper, (Yan et al., 2020) and (Wang et al., 2021) reported results on a fine-tuned version.

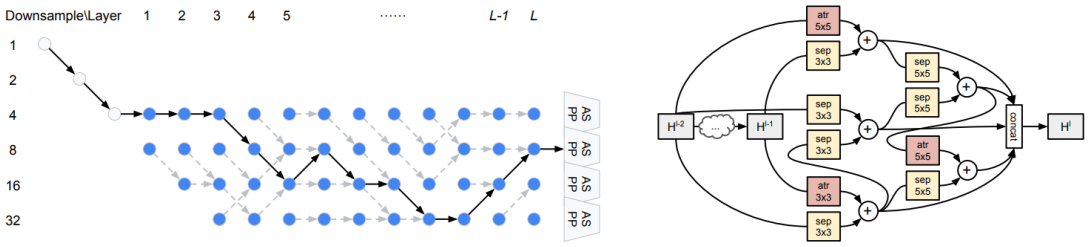


Figure 4.18: Illustration of the AutoDeepLab search space. From (Liu et al., 2019a).

Yan et al. (2020) proposed a multi-scale NAS (MS-NAS) for medical semantic segmentation in the network backbone and cell operations. They also employed multi-scale fusion, which concatenates feature maps of different scales within a network module. Authors employed

three cell types to apply downsampling, upsampling, and extract features without feature map size-changing: contracting, expanding, and non-scaling cells, respectively. Each cell is a small DAG, similar to DARTS. They also employed differential search to find the best cell configurations and network structure. Unlike other gradient-based NAS, MS-NAS returns different network configurations, which they call paths, and combines them to generate a semantic heat map. Figure 4.19 illustrates an optimized network (with five paths) with its three cell types. The search phase is executed with two GPU days in an NVIDIA GTX 1080Ti GPU. The candidate model with five paths had 11.51M parameters.

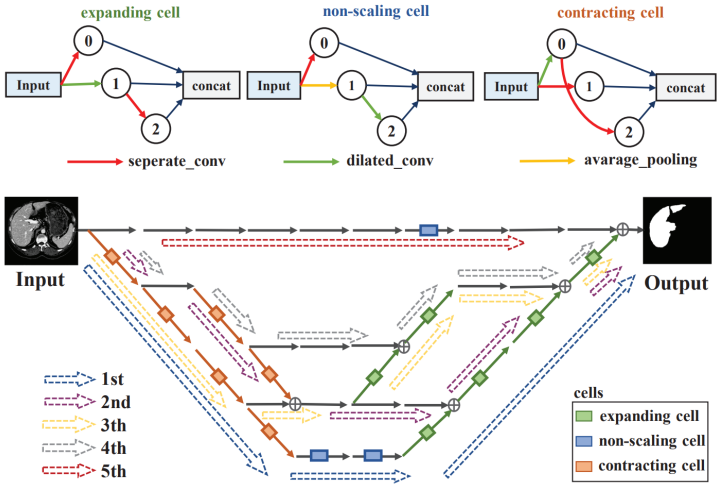


Figure 4.19: Illustration of the MS-NAS search space. From (Yan et al., 2020).

4.4.1 Overall discussion

In this section, we specifically present manually designed and NAS-generated models for the CHAOS challenge dataset. The purpose is to use this challenge as a study case of an application of NAS. Here, we presented different and robust methods applied to at least one of the proposed tasks. Table 4.3 shows some works in this challenge. Overall, the inclusion of more modalities (CT and MRI) introduce more complexity. Also, as we can expect, the inclusion of more organs reduced the DSC score.

The CHAOS challenge presents a robust challenge for medical semantic segmentation. Different modalities and tasks are provided. Thus, robust and generalized models must be crafted. Overall, most methods are manually designed. NAS implementation is seen in this challenge. Further research is encouraged in this dataset since it provides diverse data without training several models (i.e., this challenge only has five tasks).

Most works do not report model size and calculations required, neither the cost of NAS search. Thus, fair comparisons may not be viable. We encourage reporting the number of operations and number of operations of multiplication and addition for fair evaluations between different works.

Also, most works focus on optimizations to the U-Net network. This model provides efficient results in medical semantic segmentation. Thus its usage and further improvement are widespread. Since different approaches are based on U-Net, further NAS research on improving U-Net may provide significant contributions.

4.5 CONCLUSION

In this chapter, we present the works studied as the basis of our thesis. The datasets used in this works and that will be used in our thesis are described. Many NAS are also surveyed. Although they do not represent the totality of the NAS field, overall ideas can be extracted from them. Then, works employed in the CHAOS challenged are briefly described.

Table 4.3: Summary of related works surveyed for the CHAOS challenge (* denotes for reported results in a validation set only and ** denotes for an average between the results of each organ in validation sets, † denotes results reported in (Yan et al., 2020) and ‡ denotes results reported in (Wang et al., 2021)).

Approach	**Details**		**Liver**			**Abdomen**	
			CT	**MRI**	**ALL**	**MRI**	**ALL**
Manually Designed	**Proposed Model**						
OvGUMEMoRIAL	Attention 2D U-Net		.90 ± .21	.81 ± .15	.88 ± .15	.79 ± .15	.85 ± .16
ISDUE	U-Net, Auto-Encoder & Hourglass net		.91 ± .04	.85 ± .11	.87 ± .16	.83 ± .23	.85 ± .21
Lachinov	3D U-Net variant		.83 ± .20	.90 ± .05	.87 ± .13	-	-
IITKGP-KLIV	2D SUMNet & adversarial learning		-	.63 ± .07	.72 ± .31	.56 ± .06	.63 ± .36
METU_MMLAB	2D U-Net and CAN		-	.89 ± .03	.86 ± .09	.89 ± .03	-
PKDIA	U-Net & cGAN		.98 ± .00	.94 ± .01	.85 ± .26	.93 ± .02	.88 ± .21
MedianCHAOS	Ensemble of five networks		.98 ± .00	-	-	-	-
Mountain	3D U-Net variant		-	.92 ± .02	-	.90 ± .03	-
CIR_MPerkonigg	IVD-Net		-	.91 ± .07	-	-	-
NAS	**Space**	**Strategy**					
nnU-Net	U-Net ensemble	Rule-based	-	.95 ± .01	-	.95 ± .02	-
NAS-UNet*	Cell-based U-Net	Differentiable search	.974	-	-	.76	-
BiX-NAS*	Multi-scale U-Net	Differentiable & evolutionary	-	.90 ± .01	-	.8277**	-
AutoDeepLab*†	Hierarchical Dilated FCN	Differentiable search	-	.93 ± .01	-	.8752**	-
*‡			-	.88 ± .02	-	.7857**	-
MS-NAS*†	Multi-scale U-Net	Differentiable search	-	.94 ± .00	-	.9025**	-
*‡			-	.83 ± .02	-	.7990**	-

Part II

Proposed Thesis

5 ARCHITECTURE SEARCH ENVIRONMENT

In the following chapters, we present the proposed thesis concerning the theme of neural architecture search with evolutionary computation. This chapter shows how to represent convolutional neural networks with gene expression programming (GEP) for neural architecture search (NAS). Also, it shows how we can represent NAS using GEP and particularities in this approach. Characteristics of convolutional network representation are also detailed. Proposed methods for architecture search are discussed in the following chapter.

Among different functions (from a convolutional model) and inputs, **how can we find out the best combinations to solve a specific problem?** Generally, a human specialist models a convolutional network stacking an operation (or combination of operations) many times. For example, the sequence of ReLU activation, 3×3 convolution, and batch normalization is a common combination, repeated several times in a network. Different combinations can be evaluated until a good network is found. However, depending on how many operators and inputs there are, it may be too expensive to find satisfactory models.

We chose to employ evolutionary concepts to provide a more efficient search in a vast space. Moreover, it may improve the obtained models and guide the approach to a better subspace. As described below, an adapted GEP is employed to suit our problem better. Before entering in more detail, we can resume our approach in the following topics:

- We generate combinations of operations – generally called cells – automatically. We use the GEP chromosome to represent them;
- These cells are formed using operations applied to an input, like convolutions, poolings, additions, and concatenations;
- Cells are evolved (i.e., changed over generations) with mutation and recombination;
- Although we can represent the entire cell as a unique chromosome (i.e., sequence), we can also represent modules of these cells as little chromosomes – called blocks. These blocks are the ADFs of a GEP sequence;
- Each cell is composed of a combination of blocks (i.e. ADFs) with other specific functions, like additions and concatenations;
- Therefore, we aim to find out patterns that would help us attain good fitness (i.e., minimize error rate in a class-balanced classification problem).

For a better understanding of the environment, characteristics, and problems to address, in the current chapter, we show: (1) what are the primary network modules to be worked, how the network's main representation works and how can we simple represent and translate them to a deep learning model; (2) the possible operations in convnets generated by our approach; (3) the definition and explanation of the network overall structure; (4) strategies for feature map normalization; (5) weight sharing and inheritance strategies for cutting training time and reuse of the optimization already executed; and (6) replacement, to cut off weak candidates.

5.1 HOW THE NETWORK'S MAIN REPRESENTATION WORKS

In our proposal – and many others in the SOTA, networks (e.g., residual, convolutional) are constructed as a stack of cells (e.g., a unique cell repeated many times, normal and reduction cells, or a set of cells). A cell is formed with many different operations combined according to the type of network. Using the GEP representation, each element represents a possible operation to be included in a cell.

Our representation has three sequences: ops, primary index, and secondary index. The first one shows the elements (active or inactive) from this cell. The last two sequences have the inputs' positions of each element. In this proposal, a cell combines many blocks, which combines a few operations (like additions and convolutions). In Table 5.1, a example block is shown.

Table 5.1: Example of a block genotype

Pos	0	1	2	3	4
Ops	**-1**	0	**dil5x5**	**isep5x5**	**+**
Primary	-	-	**0**	**2**	**3**
Secondary	-	-	1	1	**2**

Figure 5.1 shows the phenotype for this block, which combines addition and two convolutions to generate the output h_i, including the second to last cell's output (h_{i-2}).

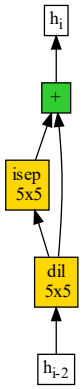


Figure 5.1: Phenotype of Table 5.1

A block can be constructed as a part of a network cell with the three sequences described before. The phenotype is formed backward. The last element is the root. Nodes are formed with the elements to the left of the root. This process follows until every active node has leaves attached, An element can have one or two inputs according to the operation type. For example, the element in position 3 is a 5x5 separable convolution in inverse order (a depth-wise convolution followed by a point-wise one), i.e. 'isep5x5'. This operation only has one input. Thus, only the information from the primary sequence is used to construct this element. In this case, the element in position 2 (5x5 dilated convolution) is used as its input.

Originally, GEP sequences were grouped in head and tail subsequences. The former has functions and terminals and the last only terminal elements. Here, sequences representing blocks have three groups: inputs, operations, and head. The **input** group contains the possible inputs for

this block (other blocks' outputs or cells' outputs). Its elements are generated according to the number of possible inputs and are immutable (i.e., reproduction is not applied in this group). The **operation** group contains operations of convolutions and poolings. This group is mutable by reproduction. Then, the **head** group behaves similar to the *operation* group plus contains binary operations (e.g., addition and concatenation).

Specifically for the **head** group in the blocks, the secondary sequence has only inputs from the operation group. This rule is applied to force the elements in this group to have elements only from the **input** group. This rule removes the possibility of a block without significant operations, like only a binary operation applied to outputs from outside the block.

In Table 5.2, we have another example of a genotype of a block. Figure 5.2 shows the phenotype of the second block.

Table 5.2: Another example of a block genotype

Pos	0	1	2	3	4
Ops	-1	**0**	**isep3x3**	**avg5x5**	**sep3x3**
Primary	-	-	**1**	**2**	**3**
Secondary	-	-	0	0	2

Figure 5.2: Phenotype of Table 5.2

As we can see, some elements are not used to construct the block. For example, the element in column 1 of Table 5.2 is not used anywhere. Although present two times in the secondary sequence (columns 2 and 3), both elements have only one input (i.e., only use the primary sequence).

Table 5.3 shows an example of cell's genotype. For better visualization, elements used to form a block will be in **bold**.

Table 5.3: Genotype representation of a cell

Pos	0	1	2	3	4	5	6	7	8	9	10
Ops	-1	0	**T3**	**T1**	**T2**	**T4**	**F1**	**cat**	cat	**cat**	**cat**
Primary	-	-	-	-	-	-	**3**	**2**	2	**6**	**9**
Secondary	-	-	-	-	-	-	**5**	**5**	3	**4**	**7**

Blocks can be terminals or functions (the first letter represents which type a cell is). In this example, the cell combines terminal and function blocks concatenated. The block genotype

dictates the inputs (only cells' outputs) in terminal blocks. Consequently, the elements in the primary and secondary sequences are never used to generate the network – thus are not present. As for the function blocks, the values in the primary and secondary sequences contain the positions of their inputs.

Similar to block sequences, cell sequences contain four groups: inputs, terminals, functions, and head. The **input** and **head** groups are similar to the block sequences, with the difference being the presence of cells' outputs only in the **input** group. As the name says, the function and terminal groups have only the function and terminal blocks, respectively.

A cell's genotype (or set of cell's genotypes) is a candidate sequence that, converting to a phenotype, represents a candidate model. The term *candidate* refers to a possible candidate automatically chosen by an architecture search approach as the most suitable model. In an evolutionary approach, a candidate model is an individual. Consequently, a population is a set of candidate models.

In Figure 5.3, we show the cell's phenotype from Table 5.3. Since only the blocks are presented but not the operations in each block, we present a normalized phenotype in Figure 5.4, using the genotype from Table 5.1 to represent every block from this cell. These blocks are combined to generate the output h_i of the current cell, generally using the previous cells' outputs h_{i-1} and h_{i-2} as its inputs. Though, in this case, only the output of the cell before the previous one was used – the input h_{i-2}.

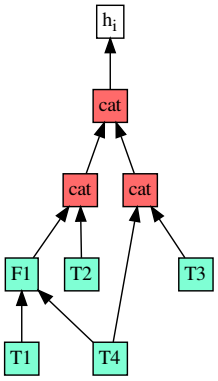


Figure 5.3: Phenotype of the cell example

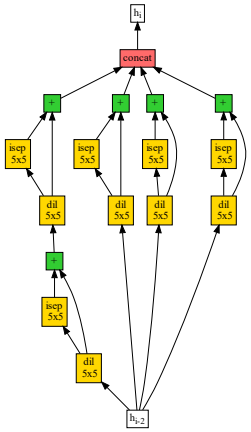


Figure 5.4: Same phenotype, but normalized

5.2 POSSIBLE OPERATIONS IN CONVNETS

In the proposed thesis, the generated convnets are composed of the following modules:

- Convolutions (these modules are composed of the operations ReLU, convolution(s) and batch normalization, stacked sequentially):
  - Point-wise and 2x2 convolutions (for feature map normalization);
  - 3x3 and 5x5 separable depth-wise convolutions (point-wise convolution, followed by a depth-wise one);
  - 3x3 and 5x5 inverted separable depth-wise convolutions (depth-wise convolution is applied then point-wise);

  - 3x3 and 5x5 separable dilated depth-wise convolutions (point-wise and then dilated depth-wise convolution with dilation = 2);
  - 3x3 and 5x5 inverted separable dilated depth-wise convolutions (dilated depth-wise convolution with dilation = 2 is applied then point-wise);

- 3x3 Max and Average Poolings (these modules are composed of ReLU -> Pooling -> Batch Norm);
- Transposed convolutions counterparts (depth-wise and dilated convolutions are replaced by it);
- Addition;
- Concatenation;

As we can see, there are many different operations in the selection pool. Automatically putting them together with different width and feature map sizes may not be straightforward. This problem occurs as some binary operations (like addition and concatenation) only works when specific dimensions are equal with both inputs. Strategies to normalize the dimensions of the inputs are discussed below.

5.3 NETWORK OVERALL STRUCTURE

Before addressing different cases of normalization, some things need to be stated. First, similar to other convnets, the overall network structure (seen in Figure 5.5) is composed of sequentially: (1) the image input (or a batch of images); (2) convolutional stem (to normalize the number of channels); (3) cell stages; (4) GAP (to reduce the feature map height and width to 1x1); and (5) a fully connected layer.

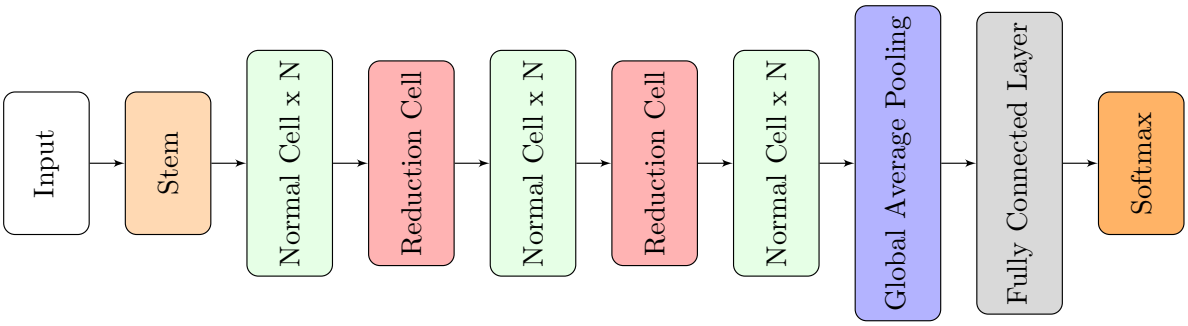


Figure 5.5: Macroview of the generated convnets.

Generally, these cell stages are composed of many normal cells repeated N times sequentially. A reduction cell (which reduces the feature map size and increases the number of channels) is found between two stages of normal cells. When addressing the depth of these convnets, we explicitly inform each stage's depth (number of cells stacked) by the N symbol. Both normal and reduction cells in our thesis have two inputs, which are outputs from the last two cells.

However, there are some exceptions: (1) since there is no cell's output before the first cell and only one in the second cell, the output of the convolutional stem is used for the two inputs in the first cell and the farthest input (h_{i-2}) in the second cell; and (2) the normal cell

after a reduction cell may have inputs of different sizes, caused by feature map reduction and channel size increasing. The reduction cell's output is used in both inputs (i.e., not generating divergences).

The number of channels is fixed to a specific size in operations from normal cells, with the first stage having the same width W (i.e., number of channels) as the convolutional stem. Also, convolutions and poolings have stride values equal to one ($s = 1$).

However, these parameters are different for reduction cells. In this cell type, operations in which the inputs contain the output of previous cells must increase the number of channels by two. Also, the feature map is divided by two (height and width) with strided operations ($s = 2$). The following normal cells use the same feature map size as the previous reduction cell.

For clarification, if the initial width is $W = 16$ and there are three stages, the input of the GAP (i.e., the last normal cell's output) will have a width of $W = 64$. If each image has dimensions of (160×160), then the feature map dimensions before GAP will be (40×40).

Of course, operations can have different dimension sizes for their outputs. However, cell operations have the same value, simplifying the network construction. For this rule to be followed, normalization is applied.

5.4 FEATURE MAP NORMALIZATION

First, convolutions can change the number of channels, height, and width – poolings only the latter two. Second, additions must have the two inputs with equal sizes; Third, concatenations can have their inputs with different widths, but their height and width (feature map size) must be the same. With this in mind, there are some cases where the dimensions must be normalized:

- When two inputs of the addition operation have divergent width, a point-wise convolution is applied to inputs with different sizes, normalizing them to the correct width;
- In reduction cells, an output of a strided operation may be added with an output of previous cells. In this case, the former would have more channels and the feature map size cut by half. The latter is normalized with a 2x2 normal convolution, fixing it to the correct size;
- When a normal cell's output is concatenated inside a reduction cell, a 2x2 normal convolution is applied to fix the feature map size (number of channels is not modified);
- Poolings do not postpone modification of channel values to convolutions when the number of channels is different, but a point-wise convolution is applied before pooling;
- One or more concatenations might be applied inside a cell. We employ a point-wise convolution to reduce the number of channels to the cell's correct width. We name this convolution as the **post-operation block**.

5.5 WEIGHT SHARING AND INHERITANCE

Two approaches might reduce the search phase time: weight sharing and weight inheritance. **Weight sharing** aims to share a block or post-operation block between candidate models. In this case, both structure and memory reference are shared. Any change to the weights reflects directly to every candidate who shares them. **Weight inheritance** is slightly different. The structure is shared, but not the memory reference. Thus, their weights would be different as back-propagation is executed.

To take advantage of the sub-optimized weights, we create a **Weights Center**. The weights from the best candidate (which uses this operation) are stored for each operation. Then, we can load the saved weights into the new candidates to cut training time. A saved weight only is changed if a candidate with better fitness is found.

5.6 REPLACEMENT

Besides preserving the best individual at each generation (i.e., elitism), a good strategy is to permanently remove the oldest individual to make room for newer individuals, encouraging novelty. If there are many candidates from the same generation, the one with the worst fitness is discarded.

Tournament selection is employed to select parents for reproduction, using validation accuracy as its metric. Reproduction is done in pairs: two parents are selected, and mutation is applied separately. The resulted children are used for recombination (i.e., crossover). These operations are applied within a probability.

One thing to note is how recombination is employed in the blocks. Since each block is a different sequence, which blocks are chosen to be recombined? For a more dynamic reproduction, at each time, blocks are randomly shuffled and then paired up (i.e., a tuple of blocks, each one from a different individual).

The reduction cell is picked from the reduction population, which can be: a new reduction cell, the same reduction cell from the parent, or another one preserved in this population. The models are included in the population if a valid sequence is generated, i.e., it has the number of blocks equal to B, and it is unique.

5.7 CONCLUSION

This chapter presented the environment in which our proposal is constructed. We show how the representation of GEP is transformed into a convolutional network. Also, we define the operations to construct a block and a cell. The cell-based convnet structure is discussed and illustrated. Then, some tweaks like feature map normalization, weight sharing, and weight inheritance are shown. They are crucial for valid networks to be generated and cut training time. Then, the metrics used for the tasks to be evaluated are chosen. The algorithm for replacement in our approach is developed. We hope that our proposals can be better understood and their concepts developed with these topics covered.

6 PROPOSED NAS APPROACHES

This chapter presents our proposed evolutionary-based NAS approaches, constructed with the environment discussed in the previous chapter as their basis. The first approach is a more simple search method and follows the common evolutionary approach of training and evaluating each candidate separately. As for the second approach, dynamic training is employed. Therefore, more explanations are included for a better understanding.

NAS consists of two phases: the search and training phase. Our approaches aim to find out the best candidate model in the search phase. In the search phase, these approaches follow the overall workflow illustrated in Figure 6.1. This workflow is summarized in the following steps: (1) an initial population is created; (2) individuals are trained and then evaluated; (3) replacement is employed if the search is not finished; (4) otherwise, the best model is returned, and the workflow is terminated; (5) reproduction is taken place; (6) go back to step 2. The best candidate model is optimized, in the training phase, for more epochs and used for prediction and other tasks.

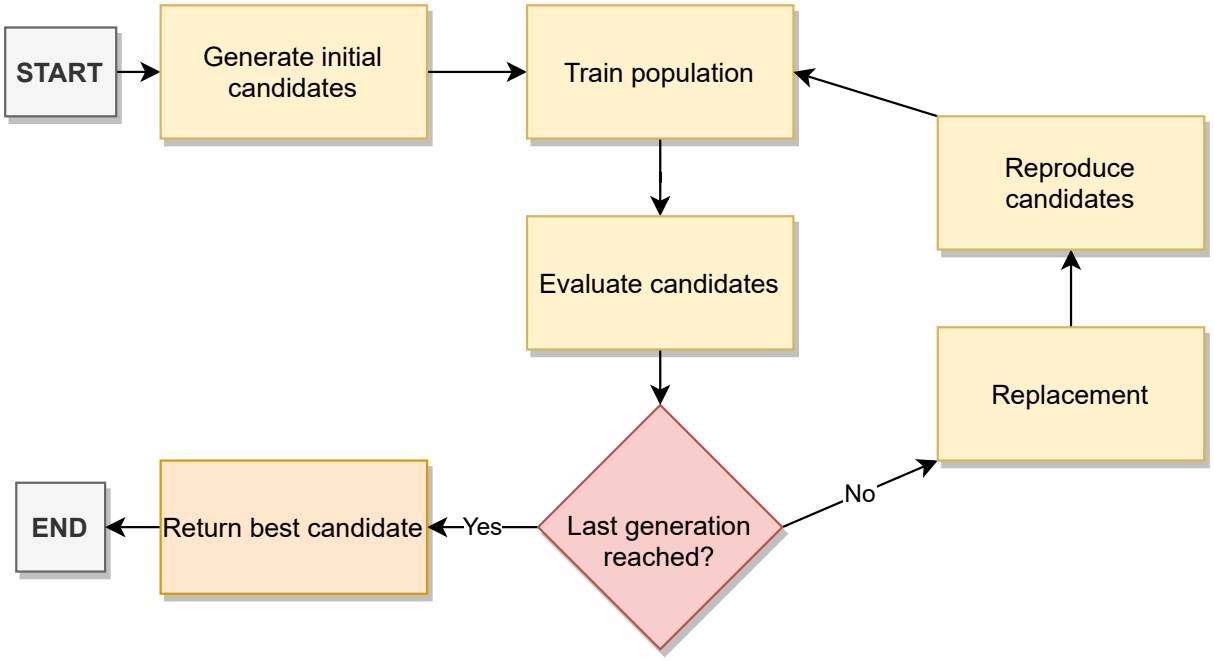


Figure 6.1: Overall approach.

The candidates of these approaches are a tuple of normal and reductions cells. Each stage of the classification is represented by a different cell. Furthermore, we propose modifications to these approaches for a five-cell representation. Finally, a modification to apply NAS to medical semantic image segmentation is proposed, where the U-Net model is used as the macro-network.

6.1 NASGEP: NEURAL ARCHITECTURE SEARCH USING GENE EXPRESSION PROGRAMMING

Based on the environment described previously, we propose a NAS method based on GEP, which we call **NASGEP (Neural Architecture Search using Gene Expression Programming)**. Like any common evolutionary approach, the candidate models are trained, evaluated, and reproduced

through generations. We aim to develop an evolutionary-based NAS approach using similar or fewer resources than gradient-based NAS approaches. In other words, to show the capability of evolutionary approaches for rapid architecture search (within twelve GPU hours in an NVIDIA RTX 2080 Ti). We want to encourage further research with this type of NAS on a low-cost budget. Moreover, reducing the search cost would encourage search in a broader search space.

The search phase in NASGEP consists mainly of a GEP workflow. Two populations of candidate sequences are created: normal and reduction populations. Before training and reproduction, these populations are initialized with random individuals ($S + Z$ individuals for each population). For each normal cell, a candidate model is created. Then, for each candidate model, a reduction cell is picked randomly.

These candidates are trained using back-propagation, one epoch per generation. Each model is evaluated with a validation subset to calculate its fitness (i.e., validation accuracy). Training for a candidate model stops when it has trained for a maximum number of epochs per model, defined as $E = 5$ to reduce finetuning and increase the number of models evaluated. If a candidate is from the current generation (i.e., it was not trained yet), saved weights are loaded into the model. If there are no weights for a specific operation, weights for this operation are randomly initialized. Then, weights from each operation are saved after a training epoch of every candidate. The training step of NASGEP is illustrated in Figure 6.2.

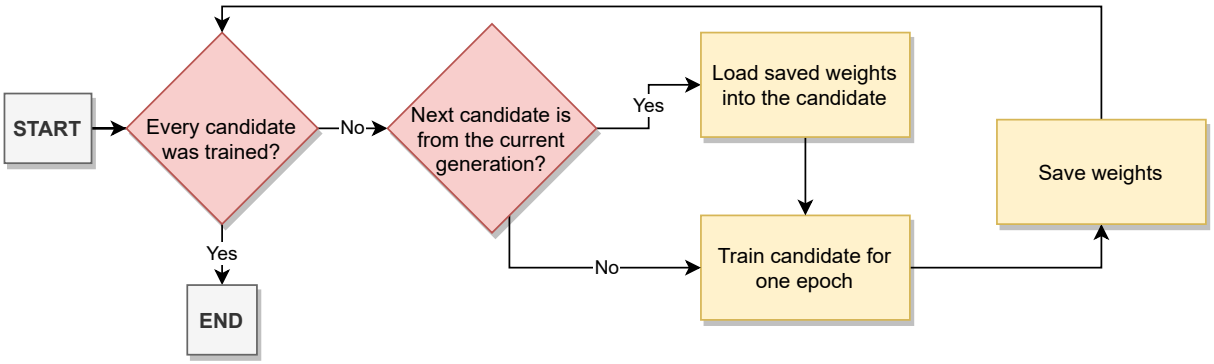


Figure 6.2: Training stage of the NASGEP approach.

Survivor selection is executed, preserving the best model (elitism), removing already converged models (trained for E epochs). The oldest model is removed if no converged model is found (excluding the best model). Finally, the worse models are removed until the population size equals S.

Reproduction is taken place, generating Z children for each population. Parents can be from both populations. After creating the required number of children, the process goes back to the training step. This process is executed until G generations. Then, the candidate model with the best fitness (a tuple of normal and reduction cells) is returned as the best candidate found. This candidate will be fully optimized in the training phase, with a large size, more epochs, and an SGD optimizer with cosine annealing. An overall algorithm of NASGEP is seen in Algorithm **??**. The parameters S, Z, E, B, G denote for, respectively, the number of preserved candidates, number of children, maximum epochs for each candidate to be trained, number of blocks in a cell, and maximum number of generations. A preliminary version of this approach was published in (Alves and Oliveira, 2020).

Algorithm 1 NASGEP's algorithm

Input: Search phase hyperparameters
Output: Best normal and reduction cells

```
1: function NASGEP(S, Z, E, B, G)
2:     normal_pop ← create_pop(S + Z, B)
3:     reduction_pop ← create_pop(S + Z, B)
4:     weights_center ← initialize()
5:     for i ← 1 to size(normal_pop) do
6:         normal_pop[i].reduction ← random(reduction_pop)

7:     for gen ← 1 to G do                          ▷ Iterating generations
8:         for i ← 1 to S + Z do
9:             if new individual then
10:                load_weights()
11:            train(normal_pop[i])
12:            validate(normal_pop[i])
13:            save_weights(normal_pop[i])
14:        replacement(S)
15:        children ← reproduce(reduction_pop, normal_pop, Z, B)
16:        normal_pop.add(children)
17:        reduction_pop.add(children)

18:    return (normal_pop[0], reduction_pop[0]) ▷ The best tuple of candidate sequences is returned
```

6.2 CSTENAS: CELL SWAPPING-BASED TRAINING FOR EVOLUTIONARY-BASED NEURAL ARCHITECTURE SEARCH

A candidate model has fixed normal and reduction cells in the first approach. Since model training is executed sequentially, changing the cells of a candidate would be infeasible. It would be necessary to generate a new model to train and evaluate a different tuple of cells. $S+Z$ candidates (S preserved candidates and Z new candidates) are evaluated for each generation. So, only a few candidate models can be evaluated in the time slot designed for the search phase. To increase the possible candidate models to be evaluated in an evolutionary approach, we propose a dynamic training approach – which we call **Cell Swapping-based Training in an Evolutionary-based Neural Architecture Search (CSTENAS)**.

As the name says, in CSTENAS, the cells are swapped at each training batch. The primary purpose is to jointly train the $(S+Z)^2$ possible combinations between the normal and reduction cells. Using the training loss returned at each batch, these many combinations can be ranked and evaluated, dynamic increasing the landscape to be analyzed since candidates are not stuck to the initial tuple combination.

Excepting the training stage, CSTENAS works similarly to NASGEP. First, $S+Z$ normal and $S+Z$ reduction candidate sequences are created. They form the initial normal and reduction populations in the search phase. Each candidate sequence must generate a cell with B blocks. Then, a supernet is created with a convolution stem, auxiliary blocks, and a fully connected layer. These components are **shared** between every candidate model, thus fixed in training and

inference. For each cell (normal and reduction), convolutional blocks are generated and loaded into the GPU. Post-operation blocks, to normalize the width of a cell's output, are created and shared between candidates.

After initialization, a loop is executed to simulate the generations of our evolutionary approach. For each generation, the population is trained for E epochs – generally $E = 1$ for CSTENAS. Algorithm 2 describes the training stage of CSTENAS.

Algorithm 2 Algorithm for dynamic training of candidates

Input: Candidate models

```
1: procedure TRAIN_POP(gen, pop)
2:     supernet ← initialize()          ▷ Initialize model with shared operations
3:     for i ← 1 to batch_size do                                  ▷ Iterating an epoch
4:         supernet.cells ← random(pop)
5:         optimizer ← Adam(supernet)
6:         supernet.forward()
7:         update_loss(supernet.normal, supernet.reduction)
```

In this stage, a supernet is trained. The normal and reductions cells are swapped at each batch by randomly picking one candidate in the respective populations. An illustration of this step is seen in Figure 6.3. The optimizer needs to restart since we do not want to create a supernet containing every cell.

Since we want to optimize only the active weights (shared blocks and the current cells), the optimizer is restarted at each iteration. This strategy enables loading the supernet into memory without batch size reduction. Moreover, since other weights are not considered, no adaptation must be made to proceed.

Then, following cell swapping, the Adam optimizer is initialized with the current active weights only. We employ two auxiliary towers (each one after a reduction cell) to improve weight flow. Then, the supernet is forwarded and back-propagated. The resulted batch loss is added to the candidate sequences. At the end of the training, candidate loss is calculated (average of the losses added for each candidate).

So, before explaining the following steps, it is essential to understand the objectives behind this approach. There are two objectives in this cell-swapping strategy: (1) training – randomly and uniformly – every active candidate model; and (2) training with different combinations of normal and reduction cells.

The first objective focuses on **training candidates equally**, at each generation, and to preserve the best ones across generations. Newer candidates use the best weights for each operation used, obtained from the current best candidates. Thus, candidates training at the same generation may compete between them.

The second objective focuses on **reducing combination bias** since they are done randomly in the training stage. It also aims to select candidate sequences that are more robust to different combinations. Furthermore, each candidate sequence – attached to a normal cell – is validated with the entire reduction cell population. This strategy selects the best reduction cell for each normal cell in the validation step. This validation approach causes the evaluation of $(S + Z)^2$ candidates (different than the previous approach, which evaluate $S + Z$ candidates), considerably increasing the search space evaluated.

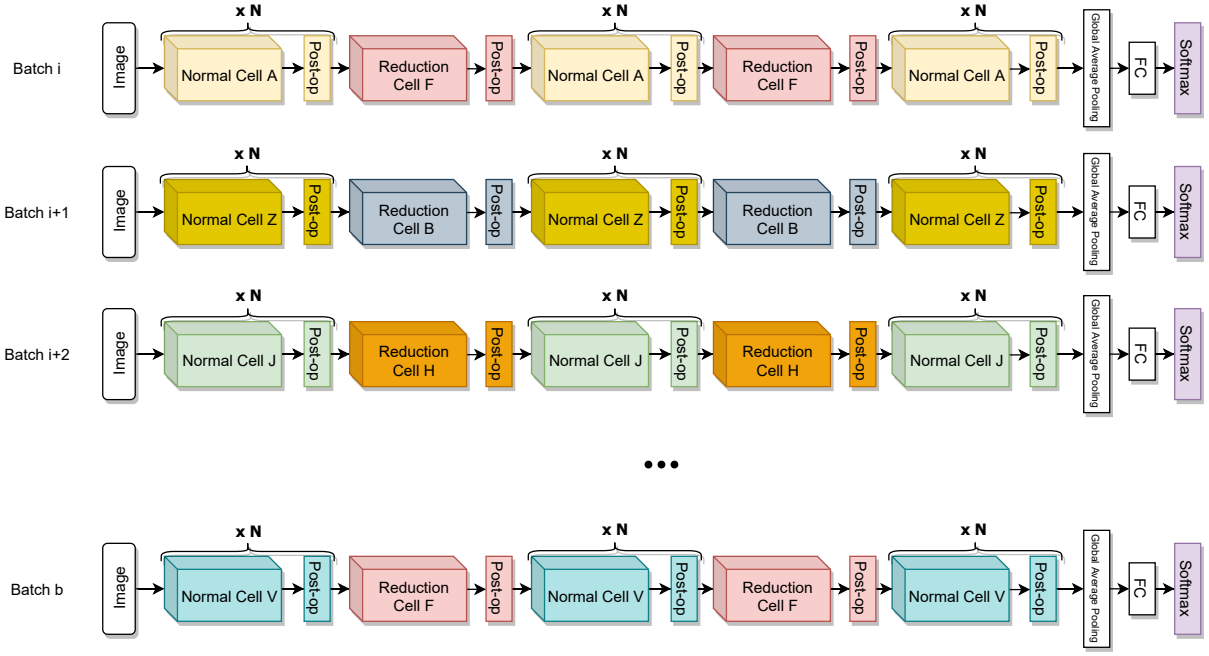


Figure 6.3: At each batch, both normal and reduction cells are randomly chosen and then forwarded.

After the training stage, normal and reduction populations are sorted by training loss in ascending order. Then, the normal cells are evaluated with the reduction population. This part is represented in Algorithm 3.

Algorithm 3 Algorithm for validating each reduction cell for a normal cell

Input: A normal cell and reduction models
Output: Best reduction

```
1:  function VALIDATE_POP(normal_cell, reduction_pop)
2:      supernet ← initialize()                    ▷ Initialize model with shared operations
3:      supernet.normal_cell ← normal_cell
4:      for each batch_block ∈ batch_blocks do      ▷ Iterating blocks of batches
5:          supernet.reduction_cell ← get_next_reduction(reduction_pop)
6:          supernet.forward()
7:          calculate_accuracy(supernet)
8:          if best reduction then
9:              reduction_cell ← supernet.reduction
10:     return reduction_cell
```

For each candidate model, the supernet is initialized with its normal cell (in this representation, the normal cell is fixed to the candidate model). The validation subset is randomly divided into blocks of batches, one block for a reduction cell. This part is executed to obtain the accuracy of each normal–reduction combination (although the accuracy is not from the entire validation subset). Then, the best combination becomes the candidate model evaluated.

Hence, the best combinations are evaluated with the entire validation subset, using the accuracy as fitness for normal and reduction cells. Weights from operations are saved if, for the specific operation, the current fitness is better than the fitness from the saved weight. After

that, replacement is employed, saving only the best S candidates from normal and reduction populations (i.e., $2S$ preserved individuals). Reproduction occurs as Z children are created for each population, loading the saved weights into the newer candidates when possible, thus beginning the next generation.

This process is executed until G generations. Finally, the best normal–reduction combination is returned as the best candidate for the search phase, then optimized in the training phase. In Algorithm 4, we show the overall idea of CSTENAS[1].

Algorithm 4 CSTENAS's algorithm

Input: Search phase hyper-parameters
Output: Best normal and reduction cells

```
1:  function CSTENAS(S, Z, B, G, P)
2:      pop[normal] ← create_pop(S + Z, B)
3:      pop[reduction] ← create_pop(S + Z, B)
4:      initialize_models()

5:      for gen ← 1 to G do                                   ▷ Iterating generations
6:          train_pop(gen, pop)
7:          sort_by_loss()          ▷ Sort populations in ascending order by training loss
8:          for i ← 1 to P do
9:              best_reduction ← validate_(pop[normal][i], pop[reduction])
10:             validate(pop[normal][i], best_reduction)
11:         for each model ∈ pop[normal] do                   ▷ Saving weights
12:             save_weights(model)
13:         replacement(S)
14:         reproduce(Z, B)
15:         load_weights()

16:     return (pop[normal][0], pop[reduction][0])          ▷ The best tuple of candidate
    sequences is returned
```

Weight inheritance is employed in the cell blocks (i.e., saving/loading best weights of children models). However, to employ weight sharing, some tweaks must be defined for this dynamic training and evaluation to work. Since each candidate sequence has different structures, it produces different outputs. If a cell has inputs with too much variation at each iteration, weights would be too difficult to be optimized. Thus, we address this pattern by using partial weight sharing.

As the name says, only part of the model is shared between candidate models. The post-operation blocks are shared to normalize the outputs to the same space. They have the objective to reduce the number of channels to a predetermined size. This normalization aims for every cell candidate output to have the same size at a specific depth (stage and block order in the stage). Sharing these post-operation blocks would force the cells' weights to be optimized to a similar space. Consequently, candidate sequences would be easily combined and optimized.

[1]Some parameters are omitted for simplicity. Please, refer to the source code for a better understanding of our approach.

6.3 DIFFERENT CANDIDATES FOR DIFFERENT CELLS

In hand-crafted and NAS-produced models, a stacking of an individual cell or the normal–reduction combination is found. Some prior studies stated that the repetition of the same block might result in higher accuracy without too much optimization. Nevertheless, models like the different versions of Inception and AmoebaNet contain different representations for specific stages. Also, when down-sampling is necessary, a different cell representation is found. These models obtained state-of-the-art results in their dates of publication. So, the question is whether good models use blocks with different representations.

A more concrete study in how and when a cell with a different representation be employed is encouraged, as it may direct to a more concise justification in their usage. A candidate model is currently defined with a tuple of candidate sequences (for normal and reduction cells) in our approaches. As a further stage in our NAS approaches, we increase the number of cells to represent a model with five cells: three of the three stages of normal cells and another two for the reduction cells between these stages.

6.3.1 Five-cell representation with NASGEP

The five-cell NASGEP approach is not much different from the original NASGEP. Each stage has a different cell instead of a tuple of normal and reduction cells. Cells are numbered from 1 to 5. Odd cells have a stride equal to one (normal cells). The two even cells have a stride equal to two (reduction cells) for their first operations (i.e., when an input is from another cell).

The most significant change is how populations are treated. There are two types of population: cell populations and candidate populations. Each stage has a different population for the first type, containing the cells used to generate reduction and normal individuals. An individual combines three (normal) or two (reduction) cells in the second type.

For each generation, in both reduction and normal populations separately, a random cell population (one of the three for normal and one of the two populations for the reduction individuals) is chosen. Every new cell is attached to a new individual based on this population. Then, for the remaining cells of this candidate, tournament selection (with $k = 2$) is employed on each remaining cell population, attaching the returned cells to the individual. Each new normal individual has a random reduction individual, forming a new candidate model with a five-cell representation.

After candidate creation, training and validation are executed like the original NASGEP version. Replacement is applied to each population (both types). Then, reproduction is employed only on the five cell populations. The children are added to the five cell populations, and a new generation begins. Only the S best candidates and cells are preserved for each population separately. The modified NASGEP's algorithm can be seen in Algorithm 5.

Algorithm 5 Modified NASGEP's algorithm for five stages

Input: Search phase hyperparameters
Output: Best normal and reduction cells

```
1: function NASGEP_MULTI_STAGE(S, Z, E, B, G)
2:     for stage ← 1 to num_stages do                ▷ Each stage has a different population
3:         pop[stage] ← create_pop(S + Z, [2..B])    ▷ Equal creation of cells with 2 to B blocks
4:     initialize_models()

5:     for gen ← 1 to G do                           ▷ Iterating generations
6:         candidates[reduction] ← generate_reduction_candidates()   ▷ Combos of two reduction cells
7:         candidates[normal] ← generate_normal_candidates(candidates[reduction])
       ▷ Combos of three normal cells and a reduction individual is attached to the candidate
8:         for i ← 1 to S + Z do
9:             if new candidate then
10:                load_weights()
11:            train(candidates[normal][i])
12:            validate(candidates[normal][i])
13:            save_weights(candidates[normal][i])
14:        for stage ← 1 to num_stages do
15:            replacement(S)
16:        children ← reproduce(pop[stage], [2..B])   ▷ Equal creation of cells with 2 to B blocks
17:        for stage ← 1 to num_stages do
18:            pop[stage].add(children)

19:    best ← (candidates[normal][0], candidates[reduction][0])
20:    return best                                   ▷ Best five-cell candidate
```

6.3.2 Five-cell representation with CSTENAS

The modified algorithm adapted to five cells is seen in Algorithm 6. One population for each stage, candidate creation, and cell attachment follows the same pattern as five-cell NASGEP. The remaining parts stay almost the same as the original CSTENAS. The differences are in the training and validation.

Algorithm 6 Modified CSTENAS's algorithm supporting a multi-cell representation

Input: Search phase hyperparameters
Output: Best five-cell candidate

1: **function** CSTENAS_MULTI_STAGE(S, Z, B, G, P)
2: **for** $stage \leftarrow 1$ to num_stages **do** ▷ Each stage has a different population
3: $pop[stage] \leftarrow create\_pop(S + Z, [2..B])$ ▷ Equal creation of cells with 2 to B blocks
4: $initialize\_models()$

5: **for** $gen \leftarrow 1$ to G **do**
6: $candidates[normal] \leftarrow generate\_normal\_candidates()$ ▷ Combos of three normal cells
7: $candidates[reduction] \leftarrow generate\_reduction\_candidates()$ ▷ Combos of two reduction cells
8: $train\_pop(gen, candidates)$
9: $sort\_by\_average\_loss()$
10: **for** $i \leftarrow 1$ to $S + Z$ **do**
11: $best\_reduction\_combo \leftarrow validate\_pop()$
12: $validate(candidates[normal][i], best\_reduction\_combo)$
13: **for each** $model \in candidates[normal] + candidates[reduction]$ **do** ▷ Saving weights
14: $save\_weights(model)$
15: $replacement(S)$
16: **for** $stage \leftarrow 1$ to num_stages **do**
17: $reproduce(pop[stage], [2..B])$ ▷ Equal creation of cells with 2 to B blocks
18: $load\_weights()$

19: $best \leftarrow (candidates[normal][0], candidates[reduction][0])$
20: **return** best ▷ Best five-cell candidate

The training follows the same pattern since we are now working with one different cell for each stage. Instead of two cells (normal and reduction) changing at each batch iteration, cell swapping occurs in each stage of the supernet. An illustration can be seen in Figure 6.4, showing the process for each batch iteration.

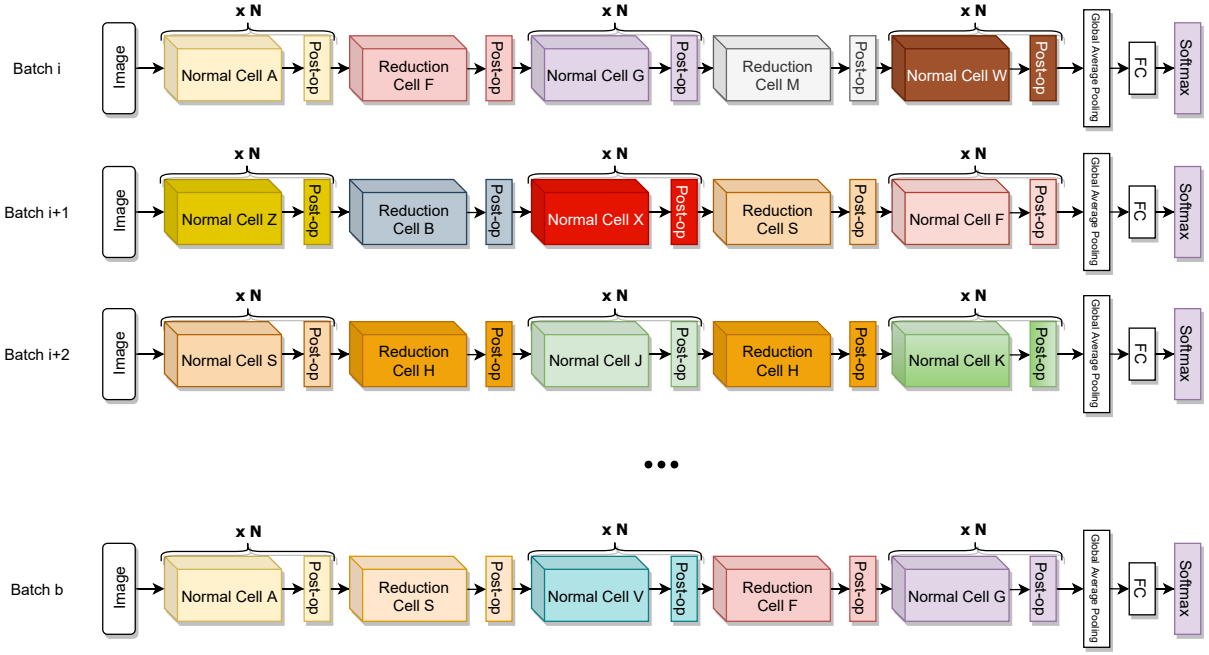


Figure 6.4: To remove bias to any candidate, cells from the five stages are randomly chosen. Like the previous version, at each batch cells from each stage population are picked up and then forwarded (and back-propagated with the same cells).

For validation, the process is the same. However, validation is now employed using the candidates, not the individuals. Algorithm 7 shows the process.

Algorithm 7 Each normal cell combo is evaluated with the entire reduction candidate group (reduction combos) to select the best reduction combo for that candidate

Input: A normal cell and reduction models
Output: Best reduction combo

```
1:  function VALIDATE_POP(candidate, candidates[reduction])
2:      supernet ← initialize()
3:      supernet.cells ← candidate.cells
4:      for each reduction_combo ∈ candidates[reduction] do
5:          for stage ← 1 to num_stages do
6:              if reduction then
7:                  supernet.cell[stage] ← reduction_combo[stage]
8:          supernet.forward()
9:          calculate_accuracy(supernet)                ▷ Accuracy of one batch
10:         if best reduction then
11:             best_reduction_combo ← reduction_combo

12:     return best_reduction_combo
```

6.4 MEDICAL SEMANTIC SEGMENTATION IN NAS

To evaluate tasks beyond image classification on natural images, we implement NAS search on medical images for semantic segmentation. Since semantic segmentation aims to realize a

pixel-wise segmentation of the entire image without instance detection, we use the U-Net model as our basis. The U-Net is a popular semantic segmentation model applied to medical images. The original and many variations are very successful in organ and lesion segmentation.

For the segmentation task, we also use the five cells approach for the five stages of the network – defined as levels. U-Net has two main parts: encoder and decoder. Each part has four levels, and each level is represented with a different cell. These cells are replicated in both parts – encoder and decoder. A fifth level is used to connect both paths. In the decoder part, depthwise convolutions are switched for transposed depthwise convolutions.

For each level of the encoder part, the outputs of the two previous cells serve as the input. Level 1 encoder cell has the output of a 3×3 convolution (applied to the image input) as its input. Level 2 encoder cell has the 3×3 convolution output and the level 1 encoder cell output as its inputs, and so on. For the decoder part, the output of a previous cell (normal arrows) is added to the output of the encoder part of the current cell (gray arrows), being the first input of the cell. In the end, the output of the level 1 decoder cell is added to the output of the first 3×3 convolution and inputted to a second 3×3 convolution. Then the resulted output is used to generate the segmentation heat-map. Figure 6.5 illustrates the custom U-Net.

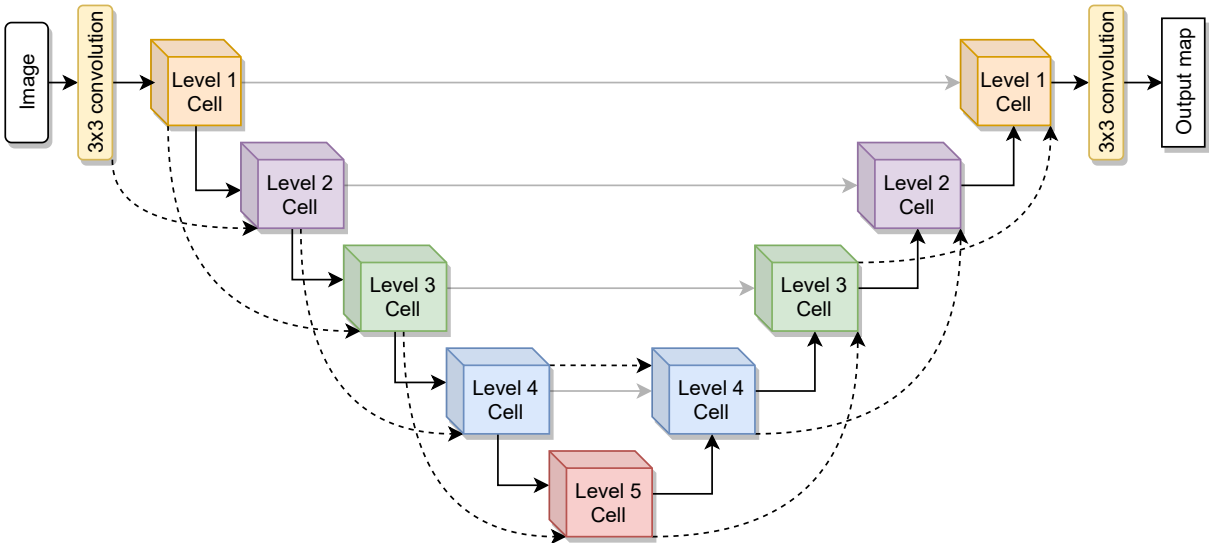


Figure 6.5: The custom U-Net to be generated with our NAS proposals (post-operation are omitted for clarity). The first operations of the levels 2 to 4 have stride $s = 2$. Solid arrows are the first input, and dashed arrows are the second input. Gray arrows are the output of the encoder part of a level (when applied, this output is added to the first input).

Also, each cell is present only two times (encoder and decoder parts, except level 5). So, the levels have almost equal representation in the model. Separation of normal and reduction combos is not intuitive here. Therefore, only one candidate combo (containing five cells) is used here. A slight modification in CSTENAS is employed for this task where, for each generation, a random level is picked up to generate the new candidates. The modified CSTENAS for the segmentation task can be seen in Algorithm 8.

Algorithm 8 Modified CSTENAS's algorithm supporting a multi-cell representation for the custom U-Net model

Input: Search phase hyperparameters
Output: Best five-cell candidate

function CSTENAS_MULTI_UNET(S, Z, B, G, P)
  for $stage \leftarrow 1$ to num_stages **do** ▷ Each stage has a different population
    $pop[stage] \leftarrow create\_pop(S + Z, [2..B])$ ▷ Equal creation of cells with 2 to B blocks
  $initialize\_models()$

  for $gen \leftarrow 1$ to G **do**
    $candidates \leftarrow generate\_candidates()$ ▷ Combos of five cells
    $train_pop(gen, candidates)$
    $sort\_by\_average\_loss()$
    for $i \leftarrow 1$ to P **do**
      $validate(candidates[i])$
    for each $model \in candidates$ **do** ▷ Saving weights
      $save\_weights(model)$
    $replacement(S)$
    for $stage \leftarrow 1$ to num_stages **do**
      $reproduce(pop[stage], [2..B])$ ▷ Equal creation of cells with 2 to B blocks
    $load\_weights()$

  $best \leftarrow candidates[0]$
  return best ▷ Best five-cell candidate

6.5 CONCLUSION

Our proposed NAS approaches are presented in this chapter: NASGEP and CSTENAS. Their workflow, modifications to employ a five-cell combo, and custom U-Net for semantic segmentation are discussed and detailed. In the following chapter, the experiments employed to evaluate these approaches and the results obtained are shown.

7 EXPERIMENTS

This chapter discusses the experiments and results obtained in our study. First, we briefly describe the computational resources used for our experiments. Then, the experiment workflow is detailed to understand the experiments realized and the characteristics of our approach. Metrics for candidate evaluation are briefly described. Furthermore, we present an ablation study in the CIFAR-10 dataset, aiming to evaluate the robustness of our approaches regarding the modifications of hyper-parameters, optimization in different search spaces, and slight strategy changes. Then, we present the experiments realized in the CIFAR-100 dataset to evaluate the employment of our approaches in a more challenging image classification task. Finally, we employ evaluation on a different task, medical semantic segmentation, to show the generalist capability of our approaches.

7.1 COMPUTATIONAL RESOURCES

This section briefly describes the computational resources used to develop and evaluate our approach. This information may aid the reproducibility of our experiments and further research based on the proposed thesis.

7.1.1 PyTorch

PyTorch is an open-source imperative, and Pythonic machine learning and deep learning framework (Paszke et al., 2019). It is focused on supporting the development of research-focused works. It contains automatic differentiation (i.e., the responsibility of computing derivatives is switched from the programmer to the framework). Also, its datatype core (i.e., Tensors) is very similar to the NumPy arrays, providing fast learning of the framework for those already familiar with it.

One feature of PyTorch is the eager execution. Unlike graph-based deep learning frameworks, it is not necessary to create a fixed network (or graph) before executing some set of operations. If an operation is already initialized in memory, it can be easily executed using the *forward()* and *backward()* functions, if necessary.

Eager execution was necessary for developing our approach because of the necessity of changing the current cells for each batch (in the supernet). It would be too complex to develop an approach in a graph-based deep learning framework since cells are swapped at each batch, and new cells are created at each generation. Eager execution simplifies cell swapping and the supernet's optimization. Since PyTorch was the more suitable framework at the beginning of the proposed thesis, it was chosen for its full development.

Our experiments run on an NVIDIA RTX 2080 Ti model with 11GB, 418.197.02 driver version, CUDA 10.1, and PyTorch 1.7.1.

7.1.2 Automatic Mixed Precision

In recent years, further development of graphic processing units (GPU) introduced the usage of the 16-bit floating-point type (FP16) in several operations of deep learning models. FP16 significantly reduces the memory used in GPUs by deep learning models and, sometimes, reduces processing time. The problem is that a deep learning model with only FP16 operations has several convergence problems caused by the lower precision.

Automatic mixed-precision approaches were developed to address convergence problems of FP16 operations. Some operations can be in FP16, but others do not. The idea of automatic mixed-precision is to train a model with operations with both types (FP16 and FP32) without instability. In cases when back-propagation has some error, for example, a NaN (Not A Number) error, the batch is discarded, and training follows. In PyTorch, we use the default automatic mixed precision[1].

7.2 EXPERIMENTAL WORKFLOW

We first execute the search phase in both approaches to find a good candidate. Then, initialize this model with random weights (using normal distribution Kaiming initialization) in the training phase and optimize it. The hyperparameters used in both phases are described in Table 7.1. The G parameter (generations) for both CSTENAS and NASGEP were chosen to be inside a 9 GPU-hour limit in a single NVIDIA RTX 2080 Ti. The exceptions were the experiment FINETUNING (executed within twelve GPU hours) and the experiment SAMPLING (the search phase was not employed), which will be described below. Moreover, the fitness average was relatively stable in these generations (i.e., only a slight improvement would be found with more generations).

Table 7.1: Hyper-parameters of the search phase and training phase for image classification.

	Search phase	**Training phase**
Epochs	5 (NASGEP only)	600
CSTENAS Generations	300	-
NASGEP Generations	20	-
Optimizer	Adam	SGD
Scheduler	-	Cosine annealing
Learning Rate	0.001	0.025
Weight decay	0.0005	0.0005
Momentum	0.9	0.9
Drop-path	-	0.2
Batch size	128	128
Initial width	$W^s = 16$	$W^t = 50$
Stage depth	$N^s = 3$	$N^t = 6$
# blocks	5	-
Head size	4	-
Terminal size	5	-
Block head size	1	-
Block terminal size	2	-
# of inputs	2	-
Mutation rate	0.1	-
1-point recombination rate	0.5	-
2-point recombination rate	0.2	-

Furthermore, some details need to be highlighted regarding the default search space (i.e., possible combinations in the cell). First, the normal and reductions cells are within the same

[1]Examples can be seen in `https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html`.

search space, with the difference being the modifications to the stride value and the number of channels. Except for specific experiments (which are explicitly stated), our default search space is defined as follows:

- Always five terminal blocks ($B = 5$) in a cell and no function blocks;
- These blocks are combined with concatenation only;
- Blocks can have the addition operation and the following convolutions (with 3×3 and 5×5 kernels):
  - Separable depthwise convolutions (pointwise followed by depthwise) and its inversion (depthwise followed by pointwise);
  - Separable dilated convolutions (pointwise followed by dilated) and its inversion (dilated followed by pointwise);

The search phase always uses training and validation subsets. Models are trained in the entire training subset and then evaluated in the test subset, where metrics are applied to assess model quality. This strategy would enable improvements without "seeing" the data to be evaluated (i.e., testing subsets).

Data augmentations are used in the training phase for image classification tasks. We employ online augmentation, where at each training batch, random augmentations are applied. The default augmentations are: (1) random cropping (padding of four pixels to the image than a random crop of 32×32); (2) random horizontal flipping; (3) AutoAugment's CIFAR-10 policy; (4) mean and standard deviation normalization; and (5) Cutout (one hole of 16×16). In Table 7.2, we show the sizes of a full model for CIFAR-10 dataset.

Table 7.2: Size of inputs and outputs of a full model trained in CIFAR-10 dataset with default parameters (five blocks per cell and an initial width of 50 channels)

		Input size	Output size
Stem	Normal convolution 3x3	$(3 \times 32 \times 32)$	$(50 \times 32 \times 32)$
Stage 1	Normal cell $\times$ N	$(50 \times 32 \times 32)$	$(50 \times 32 \times 32)$
Stage 2	Reduction cell	$(50 \times 32 \times 32)$	$(100 \times 16 \times 16)$
Stage 3	Normal cell $\times$ N	$(100 \times 16 \times 16)$	$(100 \times 16 \times 16)$
Stage 4	Reduction cell	$(100 \times 16 \times 16)$	$(200 \times 8 \times 8)$
Stage 5	Normal cell $\times$ N	$(200 \times 8 \times 8)$	$(200 \times 8 \times 8)$
GAP	Global average pooling	$(200 \times 8 \times 8)$	$(200 \times 1 \times 1)$
FC	Fully connected layer	$(200 \times 1 \times 1)$	10 (# of classes)

7.3 EVALUATING A CANDIDATE

We use some metrics for direct and indirect evaluation of candidates in image classification and medical semantic segmentation tasks. The direct evaluations are the performance estimation strategies, which will be used to remove, preserve or use candidates for reproduction. These metrics are:

- Accuracy: The most used method for model evaluation is the accuracy (the number of correct predictions divided by the total number of samples) in an unseen dataset;
- Dice: Used for medical image segmentation. Discards the evaluation of true negatives (since its inclusion may show an optimistic score) and increases the weight of importance to true positives;
- Cross-entropy or Soft-Dice Loss: Besides accuracy, one can also analyze the loss values from each trained batch. In cases when validating an entire subset is costly, one may use the training loss to discard bad candidates;

The number of parameters and the number of operations of multiplication and addition (# MAdds) are indirect evaluation metrics. Through not being used for reproduction and ranking of individuals, they describe the model size and computational cost for inference. Also, the Wilcoxon rank-sum test (Mann and Whitney, 1947) is employed to evaluate the statistically relevant difference between the accuracies of different experiments, where a value of $p < 0.05$ demonstrates a statistically significant result.

7.4 ABLATION STUDY WITH CSTENAS IN THE CIFAR-10 DATASET

Not every tweak can lead to improvements. Some changes may even hinder obtaining good models. For these cases, we propose an ablation study to appropriately evaluate the different components of an approach and how they influence its results. With this in mind, several experiments were executed to analyze different hyper-parameters employed in our approach. Given its dynamism, we aim to check what CSTENAS improves NAS (also compared with NASGEP), using the CIFAR-10 dataset for evaluation. Then, we transfer the best hyperparameters and search space found in CIFAR-10 to applied NAS for other tasks. We execute the following experiments executed in this ablation study (detailed in the following subsections):

- **DYNAMIC**: This experiment is the first one and the baseline for other evaluations. This first experiment block evaluates CSTENAS and NASGEP with their default parameters. Also, we evaluate CSTENAS with a fixed combination of normal and reduction cells for each candidate (similar to NASGEP);
- **REPRODUCTION**: In this experiment, we evaluate CSTENAS with a population size of 50 and CSTENAS with distinct reproductions in the normal and reduction populations (i.e., an individual from one population does not reproduce with an individual from another population);
- **WEIGHT**: The third experiment focuses on the differences between weight sharing and weight inheritance. We also employ partial weight sharing, where post-operation blocks are shared between candidates;
- **OPS**: Aiming to check the influence of different search spaces, we evaluate different operations and combinations. We evaluate the following search spaces with CSTENAS: baseline with pooling operations, 3×3 separable convolution and poolings, and concatenations plus additions to combine blocks;
- **BLOCK**: This experiment is another one that evaluates the search space. It is common to use four or five blocks in cell-based NAS approaches (like in AmoebaNet (Real et al., 2018), NASNet (Zoph et al., 2017) and DARTS (Liu et al., 2018)). We aim to evaluate

the performance of our NAS approaches using fewer blocks. Different widths according to the number of blocks (B) are used for the models to have similar parameters and MAdds;

- **STAGES**: Here, we aim to evaluate NAS when each stage has a different cell structure using cells with a different range of blocks (between two and five). Different from previous experiments represented by two cells (normal and reduction), each candidate has five cells;

- **FINETUNING**: The search space increases considerably with different cells for each stage (total of five) and different blocks for each cell. Thus, we use this experiment to finetune our approaches to this search space (i.e., from experiment STAGES);

- **SAMPLING**: Our last experiment in the ablation study is to evaluate the search space of the experiment STAGES with random sampling and manually designed models. The purpose of this experiment is to assess the usefulness of our NAS approaches.

For each experiment, we show plots containing the relation between test error, number of parameters, and # MAdds. Each experiment is executed in five runs, and the best candidate of each run is evaluated. The runs of each experiment are linked with a line in the plots to evaluate its overall performance better. Also, they are sorted in ascending order by the x-axis for a better analysis. Therefore, we can also verify improvements as the parameters and MAdds increase.

We aim to present the different combinations, find out by our approaches and discuss them. Also, for each experiment, the best cell combination is illustrated. Finally, a table containing the overall results is shown to better analyze the entire ablation study.

7.4.1 Experiment DYNAMIC: the capability of the dynamic search

In this experiment, we aim to analyze the influence of our dynamic search compared to a standard evolutionary search, where each candidate is trained separately (for at most $E = 5$ epochs). Comparison with NASGEP would show the importance of optimizing candidates simultaneously using shared weights since CSTENAS changes the current cells at each batch. Common evolutionary searches may introduce bias caused by the order and how much optimization is done with the same network structure.

We check the difference (e.g., computationally, accuracy) between swapping cells for each batch randomly or always swap jointly (i.e., the cells are always a combination from some candidate model). Since, for the latter, each batch contains a fixed normal and reduction combination, validation follows this pattern (i.e., no reduction selection at each validation).

In Figure 7.1, we show the plots containing the relations between different metrics of the trained models. These results are summarized in Table 7.3.

Neither result had a statistically significant difference between each other. For the normal–reduction combo, either approach would obtain similar results. Although CSTENAS with dynamic combo strategy had only a slight improvement compared to the fixed combo strategy($p = 0.1719$), we use it in the following experiments, enabling the evaluation of more combinations.

Figures 7.2, 7.3 and 7.4 are the best candidates of the runs of NASGEP, CSTENAS and CSTENAS with fixed combination, respectively. Overall, the best cells had many additions inside their blocks (in the default search space, each block is joined by a concatenation). Skip connections from previous cells (i.e., white blocks added to other nodes) are seen in every cell. Also, the usage of h_{i-1} and h_{i-2}, as the inputs of convolutions, is balanced.

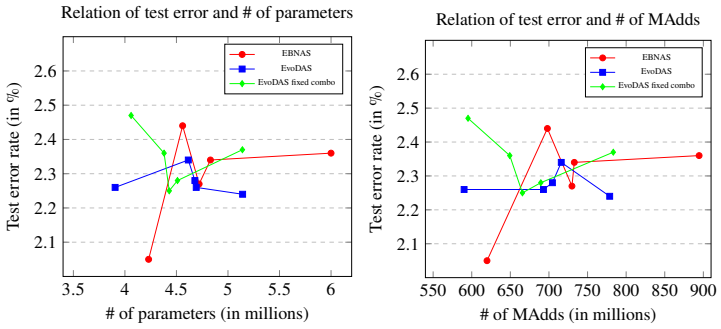


Figure 7.1: Relation of the test error, number of parameters and number of MAdds in the experiment DYNAMIC

Table 7.3: Results from the experiment DYNAMIC

Strategy	Error (in %)	# Param (in M)	# MAdds (in M)
NASGEP with $N = 20$, $B = 5, G = 300$	2.29 ± .15	4.871 ± .671	734.99 ± 100.15
CSTENAS with $N = 20$, $B = 5, G = 20$ (baseline)	2.28 ± .04	4.607 ± .445	696.43 ± 68.06
CSTENAS with fixed normal–reduction combo	2.35 ± .09	4.505 ± .395	676.59 ± 69.01

7.4.2 Experiment REPRODUCTION: the influence of number of candidates

This experiment focuses on the influence of the population size and the independence of the two populations. The experiment **REPRODUCTION** is divided into two parts: (1) we use the default search space and change the population size from 20 to 50; and (2) the normal and reduction populations are evolved separately (also using the default search space). Figure 7.5 shows the relation between the three metrics in the experiment REPRODUCTION. Also, we use the CSTENAS in experiment DYNAMIC as a baseline for further comparisons.

In Table 7.4 we show the results from this experiment, also including the baseline CSTENAS. Applying the Wilcoxon rank-sum test, the baseline CSTENAS is statistically significantly different than CSTENAS with two distinct reproductions ($p = 0.03615$). Thus, jointly reproducing distinct populations (e.g., normal and reduction populations) is better. This

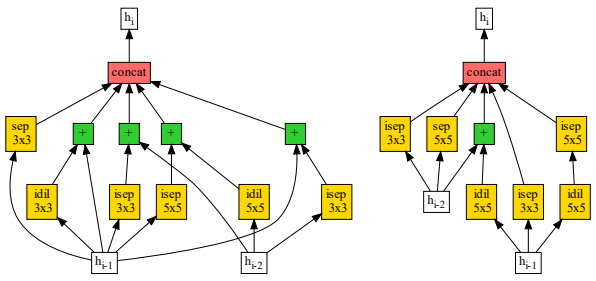


Figure 7.2: Normal (left) and reduction (right) cells of the best NASGEP run in the experiment DYNAMIC

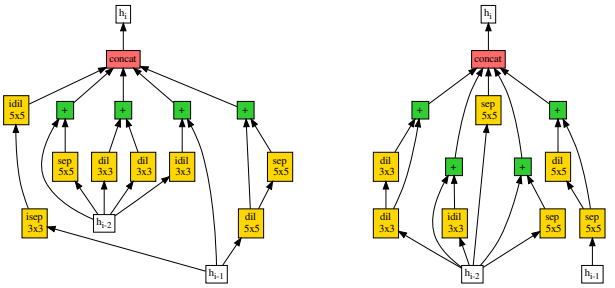


Figure 7.3: Normal (left) and reduction (right) cells of the best CSTENAS run in the experiment DYNAMIC

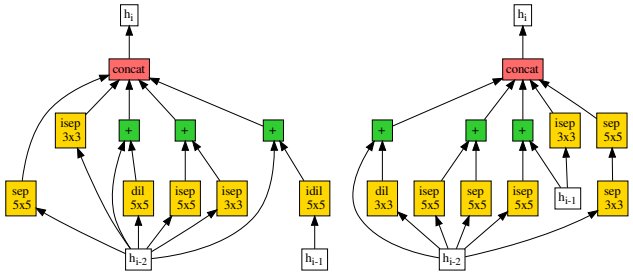


Figure 7.4: Normal (left) and reduction (right) cells of the best CSTENAS (with fixed combo) run in the experiment DYNAMIC

strategy may remove sub-optimum candidates in a population, which may be stuck in local minima.

For example, the reduction cell in Figure 7.7 does not have skip connections, which generally are used to achieve better performance. Although this specific candidate had similar results to other good candidates (which can be seen in Figure 7.5), other runs may have suffered from local minima, thus reproducing cells with bad combinations. Figure 7.6 has similar patterns as the other experiments since reproduction is executed jointly.

7.4.3 Experiment WEIGHT: analyzing weight sharing and inheritance

As stated previously, weight sharing and inheritance are defined differently here. In the former, candidates share the same memory address of an operation. As for the latter, best weights from

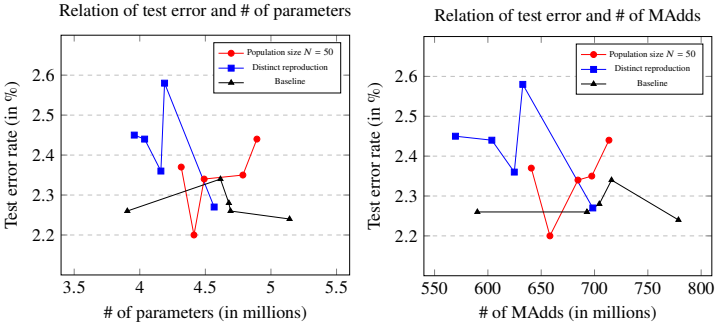


Figure 7.5: Relation of the test error, number of parameters and number of MAdds in the experiment REPRODUCTION

Table 7.4: Results from the experiment REPRODUCTION

Strategy	Error (in %)	# Param (in M)	# MAdds (in M)
CSTENAS (baseline)	2.28 ± .04	4.607 ± .445	696.43 ± 68.06
CSTENAS with $N = 50$	2.34 ± .09	4.580 ± .248	678.86 ± 29.42
CSTENAS with two distinct reproductions	2.42 ± .12	4.182 ± .234	625.84 ± 47.35

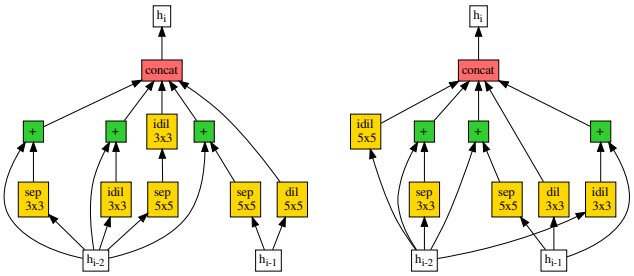


Figure 7.6: Normal (left) and reduction (right) cells of the best run with $N = 50$ in the experiment REPRODUCTION

older candidates are loaded in newer candidates, and changes in one candidate are not reflected in another.

In this experiment, we aim to evaluate the results using different approaches for weight sharing and inheritance: (1) **total sharing approach** – as the name says, every operation will be shared. Then, changes in one candidate that uses some operation will reflect in the entire population; (2) **total inheritance approach** – every operation will be initialized with saved weights (if any), but it will not have memory sharing between other candidates; and (3) **hybrid weight approach** – weight from cells will be inherited, but post-operation blocks will be shared between candidates. Figure 7.8 plots the metrics' relationship of these three experiments.

Table 7.5 shows the results for the experiment WEIGHT. Using partial weight sharing (baseline) is slightly better than only applying weight inheritance in CSTENAS ($p = 0.04587$). Partial or total weight sharing are similar ($p = 0.6752$). One thing to note is that candidate models from the Total Sharing strategy have higher parameters and MAdds averages. Figures 7.9 and 7.10 show the best candidates from full weight sharing and full weight inheritance, respectively. Like previous candidates, diverse combinations are seen, like additions of two

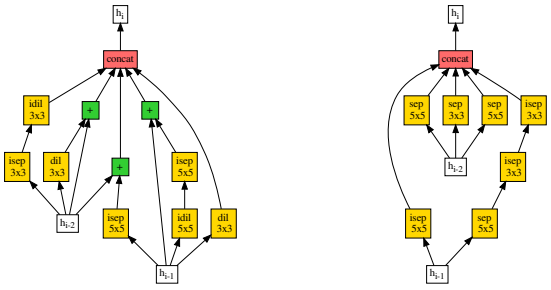


Figure 7.7: Normal (left) and reduction (right) cells of the best run in the experiment REPRODUCTION when reproduction is employed separately in the normal and reduction populations

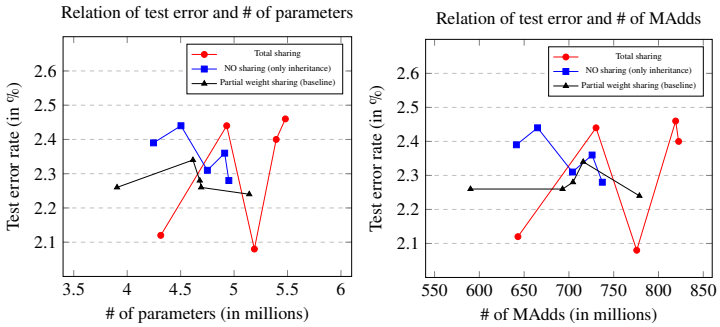


Figure 7.8: Relation of the test error, number of parameters and number of MAdds in the experiment WEIGHT

Table 7.5: Results from the experiment WEIGHT using CSTENAS

Strategy	Error (in %)	# Param (in M)	# MAdds (in M)
Partial weight sharing (baseline)	2.28 ± .04	4.607 ± .445	696.43 ± 68.06
Total sharing	2.30 ± .18	5.062 ± .468	758.15 ± 74.49
NO sharing (only inheritance)	2.36 ± .06	4.672 ± .296	694.69 ± 40.62

convolutions and additions with skip-connections. Two stacked convolutions and a convoluted result are used as the input of the other two operations (non-tree structure) in less proportion.

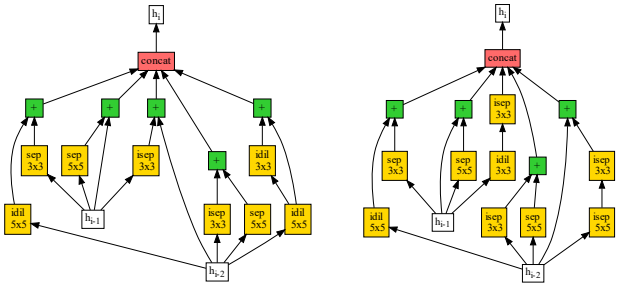


Figure 7.9: Normal (left) and reduction (right) cells of the best run with TOTAL WEIGHT SHARING

7.4.4 Experiment OPS: understanding the influence of operations in the search space

Here, we focus on evaluating search spaces with different operations. Both blocks and cells may have different combinations. How a search space is defined can lead to promising candidates. Alternatively, bad definitions can hinder the search for good models since the search space may contain bad combinations caused by inadequate operations. With this in mind, we conduct some experiments with different search spaces: (1) blocks are combined with concatenations and **additions** (cell outputs with different channel sizes); (2) inclusion of 3x3 **average and max poolings** to the default search space (evaluating the inclusion of poolings); (3) blocks are formed only with 3x3 separable convolutions, max and average poolings **(limited search space)**; and (4) inclusion of **squeeze-and-excitation** counterparts of the default operations.

Overall, this experiment had the most divergent results. Here, the search strategy was not evaluated. Figure 7.11 shows the relation of the three metrics for each search space, where

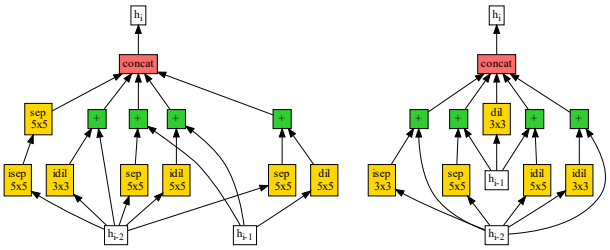


Figure 7.10: Normal (left) and reduction (right) cells of the best run with NO WEIGHT SHARING (only inheritance)

baseline+poolings and concatenations+additions (for block combination) had more variations in model sizes and error rates. Table 7.6 shows the results of this experiment.

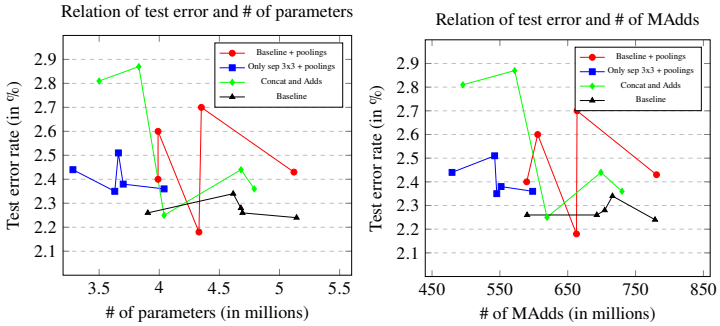


Figure 7.11: Relation of the test error, number of parameters and number of MAdds in the experiment OPS

Table 7.6: Results from the experiment OPS using CSTENAS

Strategy	**Error (in %)**	**# Param (in M)**	**# MAdds (in M)**
Baseline (normal and inverted separable and dilated convolutions)	2.28 ± .04	4.607 ± .445	696.43 ± 68.06
Baseline+poolings	2.46 ± .20	4.356 ± .461	660.57 ± 75.09
Only sep+poolings	2.41 ± .07	3.665 ± .270	543.47 ± 42.37
Concatenation+addition (for block combination)	2.55 ± .28	4.167 ± .550	623.15 ± 95.01

The addition of poolings to the default search space increased the mean error by almost 6%. However, the best candidate from *baseline+pooling* had a similar error than the candidates from baseline. As we can see in Figure 7.12, this candidate has only one pooling in its reduction cell (i.e., two in the entire model).

Figure 7.13 shows the best candidate from the experiment "only sep 3×3+poolings". This search space (only one convolution and two pooling operations) had worse models than the baseline ($p = 0.01193$). However, the average number of parameters is one million less than the baseline – also, 150 million MAdds lesser. A search space with fewer possible operations to be combined reduced the standard deviation between runs, although worse models were found.

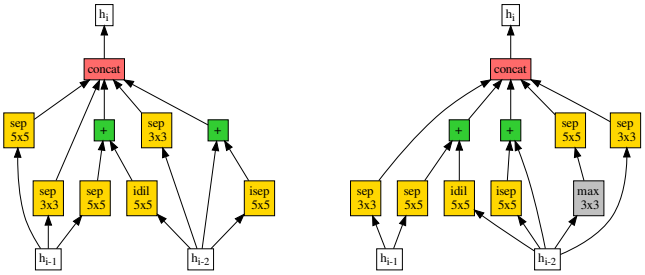


Figure 7.12: Normal (left) and reduction (right) cells of the best run with poolings (experiments OPS)

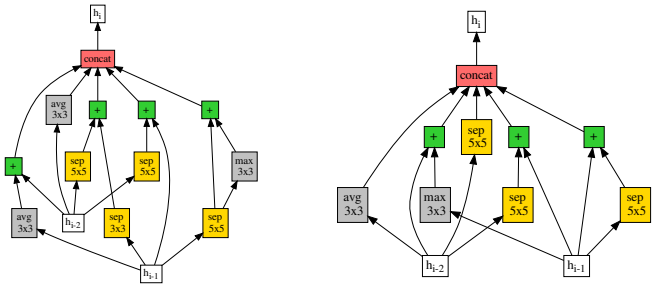


Figure 7.13: Normal (left) and reduction (right) cells of the best run with only 3x3 separable convolutions and poolings (experiment OPS)

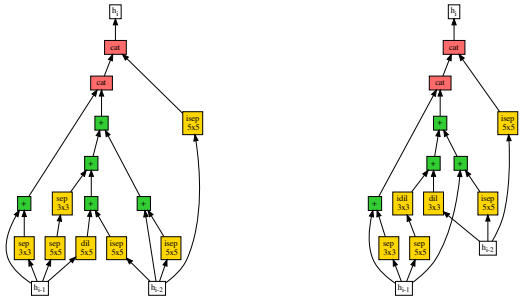


Figure 7.14: Normal (left) and reduction (right) cells of the best run with concatenations and additions joining the blocks (experiment OPS)

Finally, the usage of concatenation+addition (for block combination) had the worst results in the experiment using CSTENAS (the candidate from the best run is seen in Figure 7.14). There was no pattern in how the number of concatenations and additions to combine the blocks impacted the negative results (although the two smallest models had relatively higher errors).

Overall, the search space strongly impacts the results obtained. Although one may apply this approach in an already efficient search space to find better candidates, unknown search spaces may be more challenging.

7.4.5 Experiment BLOCK: evaluation of the number of blocks

In many cell-based NAS, four or five blocks are present in their cells. The number of blocks is fixed, and the search process is based on this configuration. Manual tuning may hinder the search

process of good models or slightly introduce bias. A specific value may introduce an overall good and similar search space for some tasks. In this case, the search strategy may have little influence on picking the best model. The question is: **there is a specific number of blocks that is adequate for some tasks?**

For this evaluation, the experiment **BLOCK** aims to evaluate search with different number of blocks ($B \subset \{2, 3, 4, 5\}$). For each value, different width values will be used. Based on the number of parameters generated from the initial configuration ($B = 5, W^s = 16, W^t = 50$), the following values will be used to generate models with similar size:

Table 7.7: Configurations for cells with different number of blocks

# blocks	W^s	W^t
$B = 2$	26	80
$B = 3$	21	65
$B = 4$	18	56
$B = 5$	16	50

With each configuration in Table 7.7, we execute a different search to obtain optimum models in that configuration. Then, we evaluate the influence of this hyper-parameter in CSTENAS and cell-based NAS. We can see in Figure 7.15 that the number of parameters and number of MAdds are similar between experiments with different B values. Thus, no block configuration is harmed by reducing operations in a cell. Table 7.8 shows the results compared with the baseline.

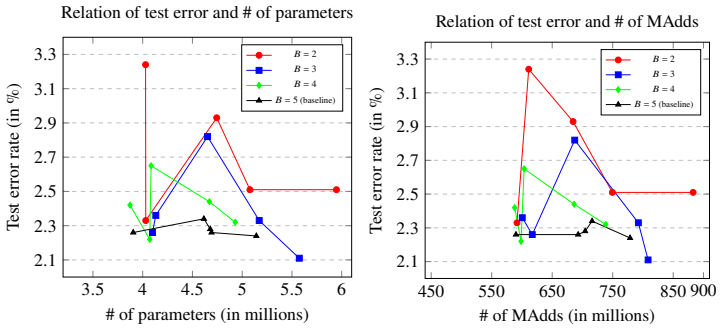


Figure 7.15: Relation of the test error, number of parameters and number of MAdds in the experiment BLOCK

Table 7.8: Results from the experiment BLOCK using CSTENAS

Strategy	Error (in %)	# Param (in M)	# MAdds (in M)
Two blocks per cell	2.70 ± .37	4.766 ± .802	703.98 ± 117.96
Three blocks per cell	2.38 ± .27	4.725 ± .646	701.16 ± 96.48
Four blocks per cell	2.41 ± .16	4.326 ± .451	642.96 ± 66.24
Five blocks per cell (baseline)	2.28 ± .04	4.607 ± .445	696.43 ± 68.06

Figure 7.16 shows the best candidate when $B = 2$. A simple cell is generated, although the initial width is greater than other models $W^t = 80$. In Figures 7.17 and 7.18, we can see more

complex cell structures. Using only two blocks per cell had the worst results in this experiment, having a statistically significant difference with the baseline ($p = 0.02078$). As we can see, using more blocks reduced the average test error and the standard deviation.

Two points can be noted here for the CIFAR-10 dataset. First, the search space when using fewer blocks may have less good candidates if compared with search spaces with more blocks. So, using a cell-based search space with more blocks is ideal for finding better models. Second, our approach may not have a robust search, which finds average models. Therefore, our approach did not suffer too much since search spaces with more blocks may have overall good models.

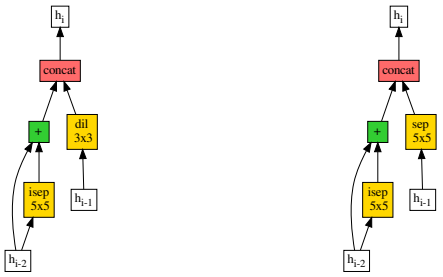


Figure 7.16: Normal (left) and reduction (right) cells of the best run for cells with two BLOCKs

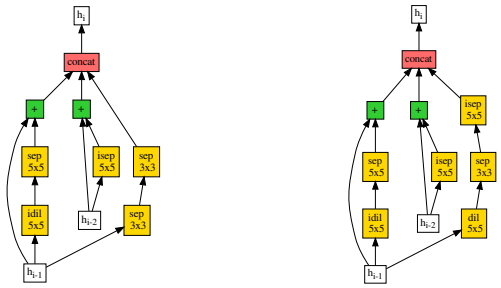


Figure 7.17: Normal (left) and reduction (right) cells of the best run for cells with three BLOCKs

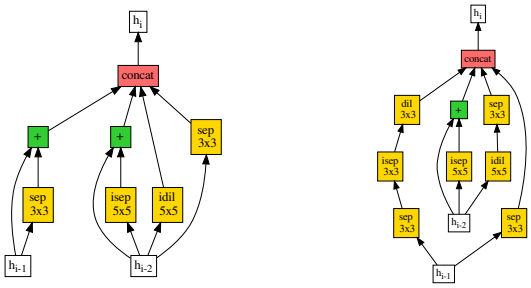


Figure 7.18: Normal (left) and reduction (right) cells of the best run for cells with four blocks

With these experiments, we can see why more blocks are employed. Further experiments are necessary to evaluate higher B values. Not only based on the error rate but also the inference cost. Although the number of parameters and MAdds may be similar, the inference cost can

differ. Besides, a better evaluation is necessary on the influence of block concatenation. Fewer operations per block, but more concatenated blocks may be reducing the standard deviation.

7.4.6 Experiment STAGES: each stage with a different cell structure

In previous experiments, we optimized the normal and reduction cells for a task and then replicated them to create the convolutional model. We limit the search space to be represented with only two cells. Generally, manually designed and NAS-based approaches also replicate a specific combination (or combinations) across the model. Although this pattern presents good performance, **is there any improvement in a model with different structures through the network?**

In experiment **STAGES**, each stage will have a different cell candidate. CIFAR-10 and other image classification datasets are common to have three stages with a reduction cell between stages. This experiment will search for candidates represented by five cells to form the convolutional model. Since the stage depth N from the search phase and training may be different ($N^s \neq N^t$), it would be necessary to change N^t. For simplification, a stage will contain only one cell structure.

This experiment will use the same search space from the experiment **BLOCK**. Cell candidates may have different B values in the same candidate model. Then, an ample search space is evaluated, containing suitable cells for specific stages. The population will have the same number of candidates for each B value for a balanced search. Using the possible values presented in Table 7.7 and a population size of 20, each B configuration will have four candidates.

For the width of each candidate, there will be two different experiments. The first experiment will have widths equal to Table 7.1 for every candidate. As for the second experiment, width values will be according to Table 7.7 when changing the number of blocks in a cell. The convolutional stem and post-operation convolutions will follow the same values described in Table 7.1. So, convolutions inside a cell will have distinct widths (according to the value of B). However, the cell's inputs and outputs will not change regarding the number of blocks in a cell.

The first, third, and fifth cells are normal cells (which are repeated N times), and the second and fourth cells are reduction ones (although they are not repeated, feature map is reduced by half and width is doubled). For example, Figure 7.20 shows a candidate with five different cells. Top-left is the first stage (normal), top-middle is the second stage (reduction), top-right is the third stage (normal), bottom-left is the fourth stage (reduction), and bottom-right is the fifth stage (normal).

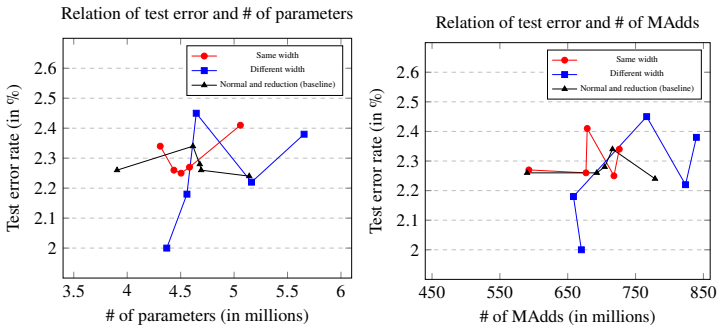


Figure 7.19: Relation of the test error, number of parameters and number of MAdds in the experiment STAGES

Table 7.9: Results from the experiment STAGES using CSTENAS

Strategy	Error (in %)	# Param (in M)	# MAdds (in M)
Normal and reduction tuple with $B = 5$ (baseline)	2.28 ± .04	4.607 ± .445	696.43 ± 68.06
Different blocks with same width	2.31 ± .07	4.578 ± .287	678.49 ± 52.90
Different blocks with different width	2.25 ± .18	4.879 ± .524	751.65 ± 84.37

Table 7.9 shows the results of experiment STAGES. This experiment shows that CSTENAS obtained similar results (see a comparison in Figure 7.19) to the baseline, even with a bigger search space. Analyzing the best candidates in Figures 7.20 and 7.21, heterogeneous models were find out. Although most cells had five blocks (in this search space, we can see the number of blocks checking the inputs to the concatenation), some cells had other quantities.

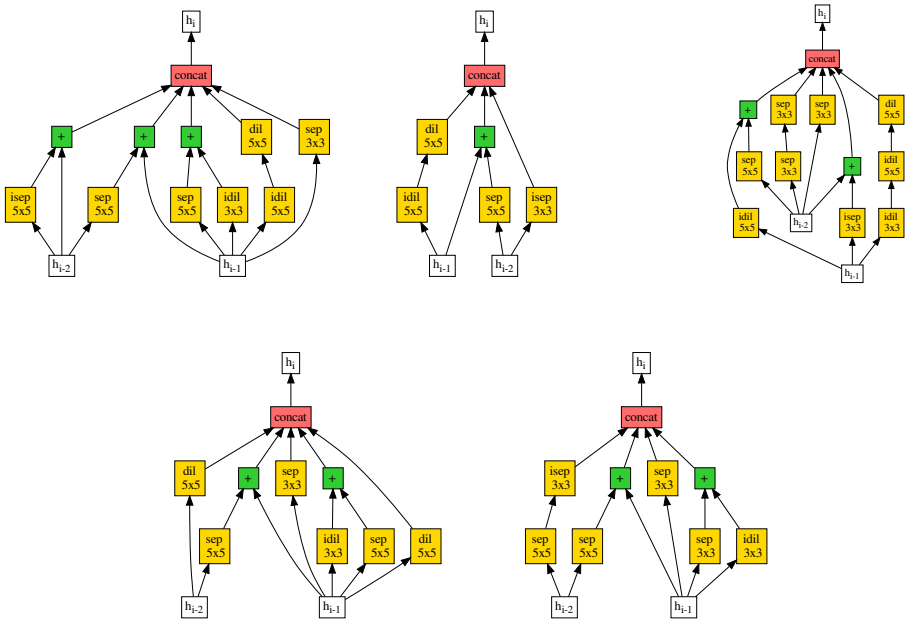


Figure 7.20: Cells of the five stages (in order) from the best run of CSTENAS with five stages and same width

Even when an optimum number of blocks is unknown, we can search different configurations in a unique search. Good models similar to suitable search spaces (e.g., the baseline) are found. In this experiment, we find out the best model of our approaches in the CIFAR-10 dataset, with an error rate of 2% (seen in Figure 7.21). Thus, the necessity of setting a fixed value can be thrown out.

7.4.7 Experiment FINETUNING: finetuning the best candidates

In this simple experiment, we check the influence of finetuning the best candidates obtained in the search phase from experiment STAGES (in the five runs). Since the number of cells significantly increases (from two populations to five distinct populations of cells), optimization of the candidates may be slower in the search phase.

After the default search phase, the best ten candidates are preserved and individually trained for 20 epochs. The candidate with the best average fitness between these 20 epochs is

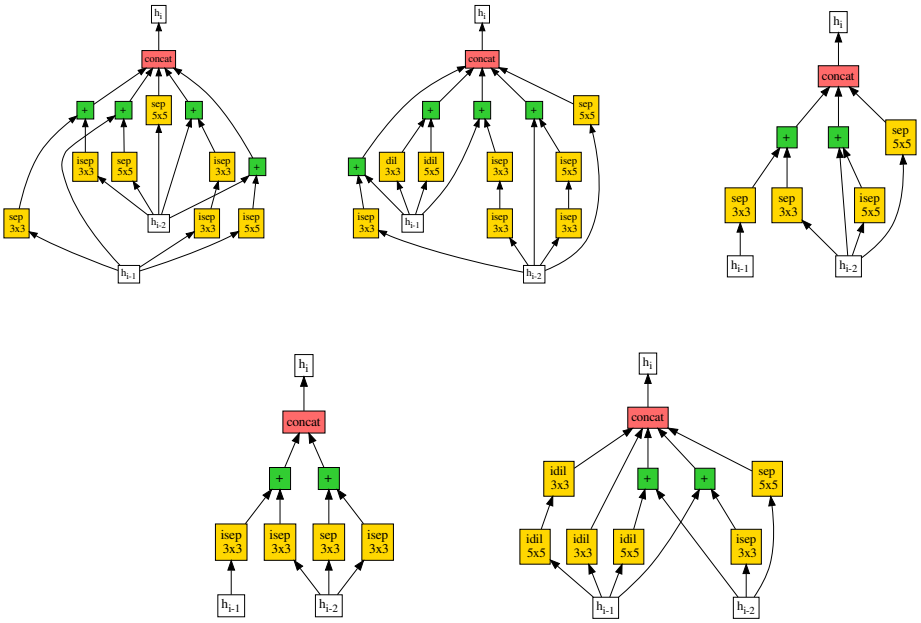


Figure 7.21: Cells with five stages (in order) from the best run of CSTENAS with five stages but different widths based on the number of blocks (also the best model in the entire CIFAR-10 study)

selected for the training phase. If the selected candidate was the previous best candidate, the model is not retrained, but the previous result is used.

For a better comparison, we apply NASGEP for 30 generations in the STAGES search space (to match the search cost of CSTENAS with finetuning). The motivation is to check which approach is better in this case. The same runs from the previous experiment were used for this finetuning.

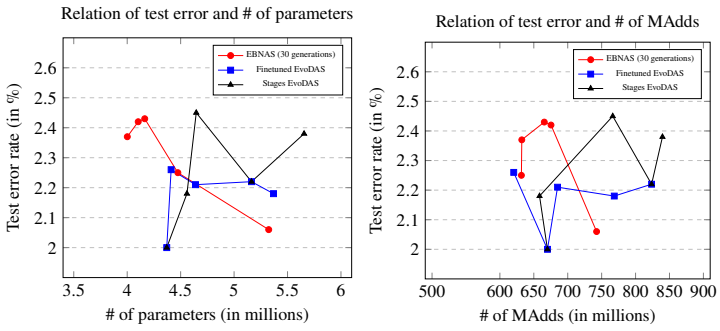


Figure 7.22: Relation of the test error, number of parameters and number of MAdds in the experiment FINETUNING

Table 7.10 shows the results of this experiment. NASGEP had similar results to the original approach (the best run had the candidate seen in Figure 7.23). As for CSTENAS, two runs had superior results (without finetuning, two runs had 2.38% and 2.45%, then finetuning in the search phase improved results to 2.18% and 2.26%, respectively), one had a similar error (2.18% without finetuning and 2.21% with finetuning). The other two did not change the candidate (one of which is already seen in Figure 7.21). These differences can be better seen in Figure 7.22.

Table 7.10: Results from the experiment FINETUNING

Strategy	Error (in %)	# Param (in M)	# MAdds (in M)
NASGEP / different blocks with different width + 10 generations	2.31 ± .16	4.413 ± .539	669.48 ± 45.28
CSTENAS / different blocks with different width + 20 epochs	**2.17 ± .10**	4.790 ± .452	713.56 ± 81.43

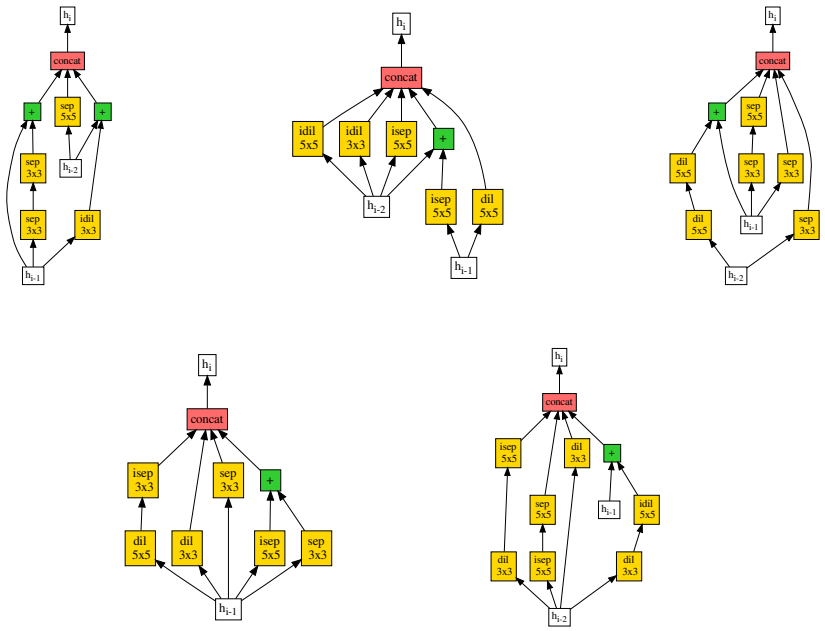


Figure 7.23: Cells of the five stages (in order) from the best run of NASGEP with 30 generations

7.4.8 Experiment SAMPLING: is NAS necessary?

One thing to think about is how well a random sample from a specific search space behaves in a task. There is a vast amount of effort to research and apply different NAS approaches. However, **is it necessary to apply a specific NAS approach, or is it better to pick a random sample from the search space and train it?**

In this experiment, our motivation is to verify if CSTENAS performs well enough to justify its usage compared to random sampling. For this, the idea is to apply random sampling to select a few valid models, i.e., that CSTENAS can pick in its search phase and train them. Besides a Wilcoxon rank-sum test evaluation, we apply the evaluation formula proposed in (Yang et al., 2019). NAS methods present promising results mainly due to a robust (or biased) search space and efficient training configurations (e.g., data augmentation, strong regularization). One of the problems in NAS research stated by them is the inefficiency of different search methods. The relative improvement (RI) metric is calculated as follows:

$$RI = 100 \times \frac{Acc_m - Acc_r}{Acc_r}, \tag{7.1}$$

which Acc_m and Acc_m are the accuracies (in %) of a specific NAS approach and random sampling, respectively. With this, we can evaluate the influence of CSTENAS in obtaining promising models in an automated way.

The search space evaluated is the same as the experiment **STAGES**, as it has the bigger search space in our proposal. With this in mind, we can check the difference between random sampling and our proposal with a search space that may contain sub-optimum models. Moreover, five manually designed models were also trained and evaluated to check the improvement of the employed of NAS in contrast with expert-designed models.

As we can see in Figure 7.24, there is a high variation in the error rates of the sampling runs.

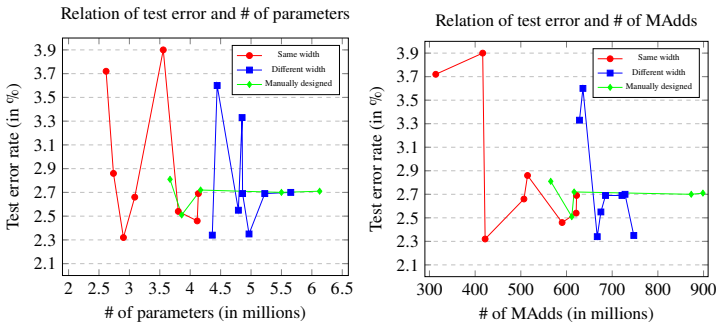


Figure 7.24: Relation of the test error, number of parameters and number of MAdds in the experiment SAMPLING

Table 7.11 shows the results of the sampling runs and the manually designed runs.

Table 7.11: Results from the experiment SAMPLING

Strategy	Error (in %)	# Param (in M)	# MAdds (in M)
Random sampling / different blocks with same width (8 runs)	2.89 ± .59	3.369 ± .613	500.92 ± 110.69
Random sampling / different blocks with different width (8 runs)	2.75 ± .45	4.895 ± .412	686.14 ± 43.21
Manually designed	2.69 ± .11	4.665 ± 1.087	713.10 ± 159.21

In comparison to our best result (CSTENAS with a test error rate of 2.17% ± 0.10), there is a statistically significant difference between it and the two random sampling experiments ($p = 0.001554$ for same width and $p = 0.004257$ for different widths). Regarding the relative improvement (RI metric), there was a relative improvement of 0.5964 in CSTENAS with cells where widths are configured based on the number of blocks. As for NASGEP in the experiment FINETUNING, there was a relative improvement of 0.4524. For comparison, DARTS had a $RI = 0.32$, showing how we had better improvements than one of the most popular NAS approaches. We also had higher accuracy in the CIFAR-10 dataset (DARTS had an error rate of 2.76 ± 0.09).

Compared to manual design, CSTENAS and NASGEP also had statistically significant differences ($p = 0.007937$ for both). We can see the improvement using NAS, even with carefully manually designed models. In Figure 7.25, we show the five designed models. As we can see, they use the same search space with two main differences: combinations are uniform, and there is only one cell representing the entire model. These two features were employed since they are closer to what an expert would develop.

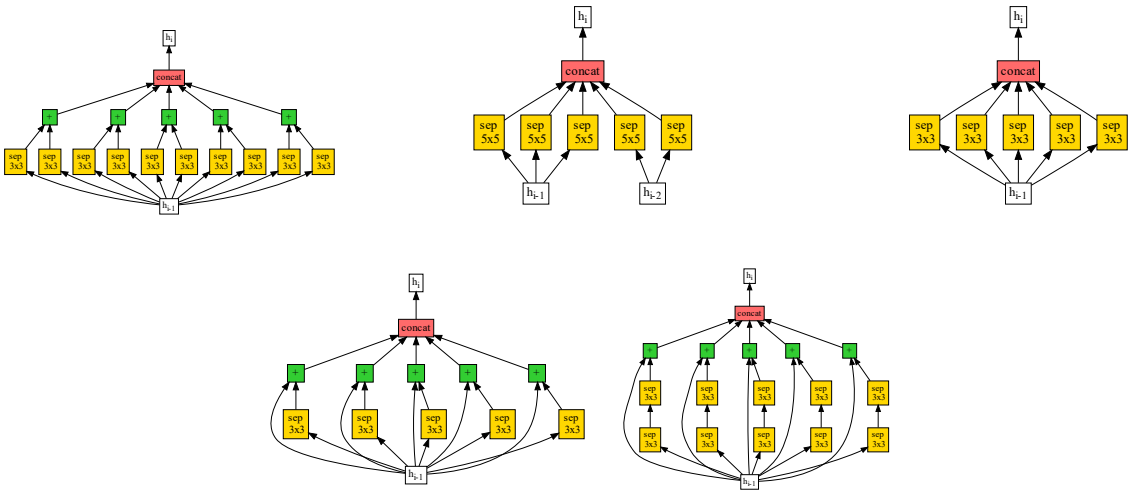


Figure 7.25: Cells of each model manually designed and evaluated

The design was aimed to use many combinations of operations and the employing of skip-connections, more depth, large kernels, and usage of outputs of the two previous cells. The best model was the second one, with an error of 2.51% (which is a slightly weaker model if compared with our NAS approaches). This model had less than 4 million parameters and used a few 5×5 separable convolutions. Also, it was the only cell that used the output of the two previous cells. The worst model (third cell) had similar parameters but an error rate of 2.81%. For each block in this model, only 3×3 separable convolutions and the output of the previous cell were used.

In both random sampling and manually designed models, we can see that the number of parameters does not correlate with improved results (see Figure 7.24). Especially in the manually designed, the other models had close results with the worst model (one model had more than 6 million parameters). This simple experiment can show how an intelligent design can obtain promising results even with a smaller size.

7.4.9 Overall results and next steps

In Table 7.12, we show the results obtained in the entire ablation study. Overall, modifications in the search strategy introduced little difference. Although this may appear as a negative point, this stability would aid in applying our approaches to other tasks without further hyper-parameter tuning. Analyzing the results concerning random sampling and manually designed models, we obtained satisfactory improvements.

The main problem is the search space design, which impacted the final results considerably when changed. However, even with worse search spaces, our approaches had superior results if compared with the models obtained in the experiment SAMPLING. Therefore, we would be encouraged to use our approaches instead of manual and straightforward ones.

Most NAS works involve convnets generation for image classification in the CIFAR10 dataset. However, they are not necessarily specific for this type. Some are focused on other datasets, and others focus on another task, like segmentation of medical images. Experiments in other domains would assert the versatility of techniques for automatic network modeling.

Table 7.12: Overall results obtained in the ablation study employed in the CIFAR-10 dataset

Experiment	Strategy	Configuration	Error (in %)	# Param (in M)	# MAdds (in M)
DYNAMIC	NASGEP	$N = 20, B = 5, G = 300$	2.29 ± .15	4.871 ± .671	734.99 ± 100.15
	CSTENAS	$N = 20, B = 5, G = 20$ (baseline)	2.28 ± .04	4.607 ± .445	696.43 ± 68.06
		Fixed normal and reduction combo	2.35 ± .09	4.505 ± .395	676.59 ± 69.01
REPRODUCTION	CSTENAS	$N = 50$	2.34 ± .09	4.580 ± .248	678.86 ± 29.42
		Two distinct reproductions	2.42 ± .12	4.182 ± .234	625.84 ± 47.35
WEIGHT	CSTENAS	Total sharing	2.30 ± .18	5.062 ± .468	758.15 ± 74.49
		NO sharing	2.36 ± .06	4.672 ± .296	694.69 ± 40.62
OPS	CSTENAS	Baseline + poolings	2.46 ± .20	4.356 ± .461	660.57 ± 75.09
		Only sep 3x3 + poolings	2.41 ± .07	3.665 ± .270	543.47 ± 42.37
		Concatenation and addition to combine blocks	2.55 ± .28	4.167 ± .550	623.15 ± 95.01
BLOCK	CSTENAS	Two blocks per cell	2.42 ± .12	4.766 ± .802	703.98 ± 117.96
		Three blocks per cell	2.38 ± .27	4.725 ± .646	701.16 ± 96.48
		Four blocks per cell	2.41 ± .16	4.326 ± .451	642.96 ± 66.24
STAGES	CSTENAS	Different blocks with same width	2.31 ± .07	4.578 ± .287	678.49 ± 52.90
		Different blocks with different width	2.25 ± .18	4.879 ± .524	751.65 ± 84.37
FINETUNING	NASGEP	Different blocks with different width + 10 generations	2.31 ± .16	4.413 ± .539	669.48 ± 45.28
	CSTENAS	Different blocks with different width + 20 epochs	**2.17 ± .10**	4.790 ± .452	713.56 ± 81.43
SAMPLING	Random	Different blocks with same width (8 runs)	2.89 ± .59	3.369 ± .613	500.92 ± 110.69
		Different blocks with different width (8 runs)	2.75 ± .45	4.895 ± .412	686.14 ± 43.21
	Manual	Manually designed	2.69 ± .11	4.665 ± 1.087	713.10 ± 159.21

We aim to verify the generality of our NAS approach in comparison with other NAS works and hand-crafted models. Thus, for this work, we evaluate our NAS approach basically with two tasks: more general image classification and medical image segmentation. The idea is to provide an overall understanding of each task to be addressed.

7.5 IMAGE CLASSIFICATION IN AN DIFFERENT DATASET: CIFAR-100

To evaluate a more complex image classification task in our approaches, we employ experiments in the CIFAR-100 dataset. Although this dataset has the same image size and number of samples as the CIFAR-10 dataset, one hundred classes can be found. Thus, each class has only 600 samples for each class, where CIFAR-10 dataset has 6000 samples per class. Although image classification is more straightforward than other tasks, its difficulty can be seen as the number of classes increases (and its complexity). Thus, evaluation of different datasets can aid in a better understanding of the capabilities of CSTENAS for this task.

Since there are few differences between the CIFAR-10 and CIFAR-100 datasets, the same experiment workflow was used here. To better evaluate the proposed approaches, the following experiments were executed in this dataset: (1) NASGEP with five stages (between two and five blocks for each cell) for 30 generations; (2) CSTENAS with five stages (between two and five blocks for each cell) for 300 generations; and (3) a finetuning CSTENAS (resuming the search phase of the previous experiment) with more 20 epochs for each candidate, separately.

In Table 7.13, we show the overall results obtained in this dataset. Also, we plot in Figure 7.27 the individual runs to evaluate the scenario.

Table 7.13: Results obtained in the CIFAR-100 dataset

Strategy	**Configuration**	**Error (in %)**	**# Param (in M)**	**# MAdds (in M)**
NASGEP	Different blocks with different width (30 generations)	15.47 ± .51	4.780 ± .288	709.82 ± 54.52
CSTENAS	Different blocks with different width (STAGES)	16.07 ± .42	4.486 ± .241	685.30 ± 11.18
	Different blocks with different width + 20 epochs (FINETUNING)	15.84 ± .15	4.865 ± .591	701.22 ± 72.51

Both NASGEP and CSTENAS had a high standard deviation, but the employment of the finetuning in CSTENAS reduced its standard deviation and improved the overall results. Although NASGEP has a higher standard deviation, its worse results are comparable with finetuned CSTENAS. However, its best candidate had the best accuracy in the overall CIFAR-100 experiment. The best candidate had a test error rate of 14.94% (seen in Figure 7.26). Figures 7.28 and 7.29 show the best candidates of CSTENAS without and with finetuning in the search phase, respectively (the candidates are from different runs).

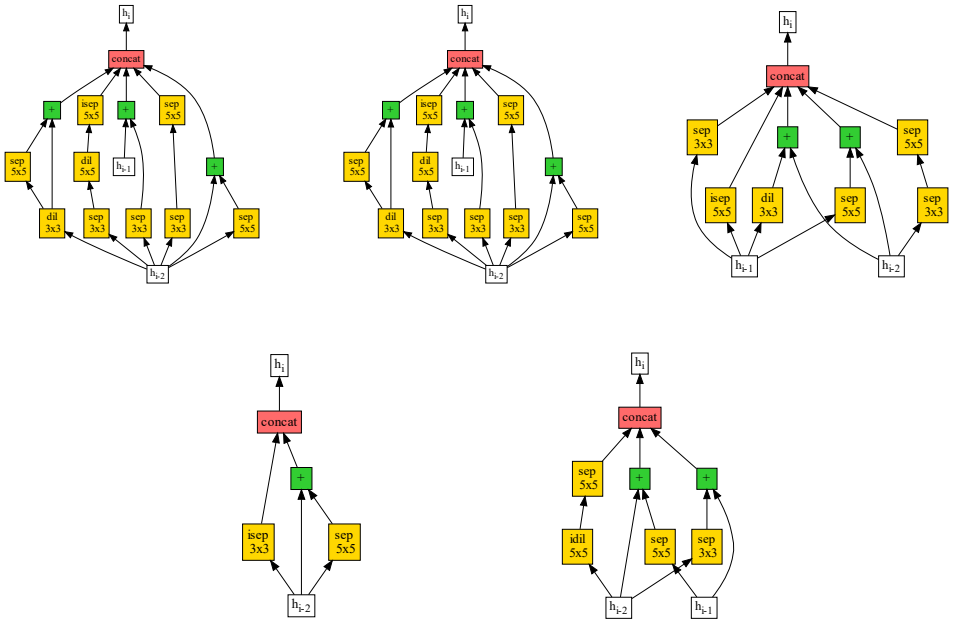


Figure 7.26: Best candidate from NASGEP in CIFAR-100 dataset

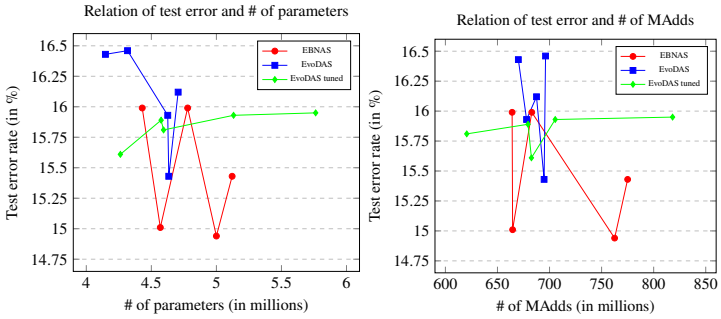


Figure 7.27: Relation of the test error, number of parameters and number of MAdds in the CIFAR-100 dataset

7.6 MEDICAL IMAGE SEMANTIC SEGMENTATION: THE CHAOS DATASET

As we previously stated, our NAS approach is not limited only to convolutional networks for classification. The core ideas are the same when using convnets for detection and segmentation, also with recurrent nets or other types of deep networks. The differences are mainly in the combination of functions and terminals, with some tweaks (e.g., heatmaps must be generated in the end). We adapted the search phase to be executed with a similar GPU cost to the experiment FINETUNING for the CHAOS challenge – a semantic segmentation task applied to medical images.

We use the U-Net model as the basis of our NAS. This model is famous for the medical image semantic segmentation task. Unlike the classification task, we do not use the normal and reduction cell pattern. However, we follow the structure of five levels from U-Net. Each level has different receptive fields (controlled by the stride parameter). Since medical semantic segmentation with a custom U-Net has a different structure than our image classification custom networks, changes to the width values were made. Based on Table 7.7, we cut by half the widths, generating custom models with a similar number of parameters (to the image classification task). In Table 7.14, we show the initial widths for cells with different number of blocks.

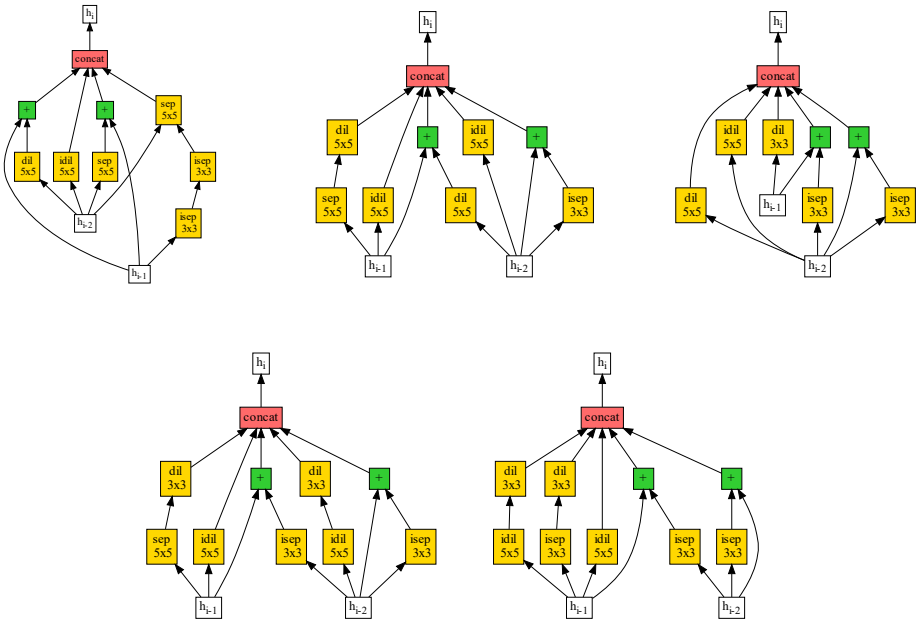


Figure 7.28: Best candidate from CSTENAS in CIFAR-100 dataset

Table 7.14: Configurations for cells with different number of blocks for our custom U-Net

# blocks	W^s	W^t
$B = 2$	13	40
$B = 3$	12	32
$B = 4$	9	28
$B = 5$	8	25

For each level of the custom U-Net model (in encoder and decoder parts), we have different cell representations and strides = 2 except for level 5. These values were chosen according to match the original U-Net different levels. Similar to the default search space, separable and dilated convolutions are included.

Other hyperparameters were changed according to this task (and to match similar works applied to the datasets evaluated), which are summarized in Table 7.15. First, the AdamW optimizer is used for 100 epochs with a learning rate of 3e-4 and a one-cycle cosine learning rate scheduler. We use this scheduler to stabilize the model in the first epochs. After the 30º epoch, the scheduler works similar to the cosine annealing used in the image classification tasks. Also, the soft-Dice loss function is combined with the cross-entropy loss. Soft-Dice loss is used since the true negative is not essential in this task and may introduce optimistic bias. However, we combine both losses for better model optimization since gradient optimization is better and more stable with cross-entropy.

In the preprocessing part, auto-contrast is applied to the data. Also, resize is applied in tasks with MRI to normalize the height and weight of the inputs. The online data augmentation uses random resize, crop, curvature flow, shear, translation, rotate, and elastic transform.

We report the results obtained from the CHAOS challenge evaluation server. Usage of an evaluation server may present more reliable results since the testing ground truth is not accessible. This challenge proposes five different tasks to evaluate the semantic segmentation of

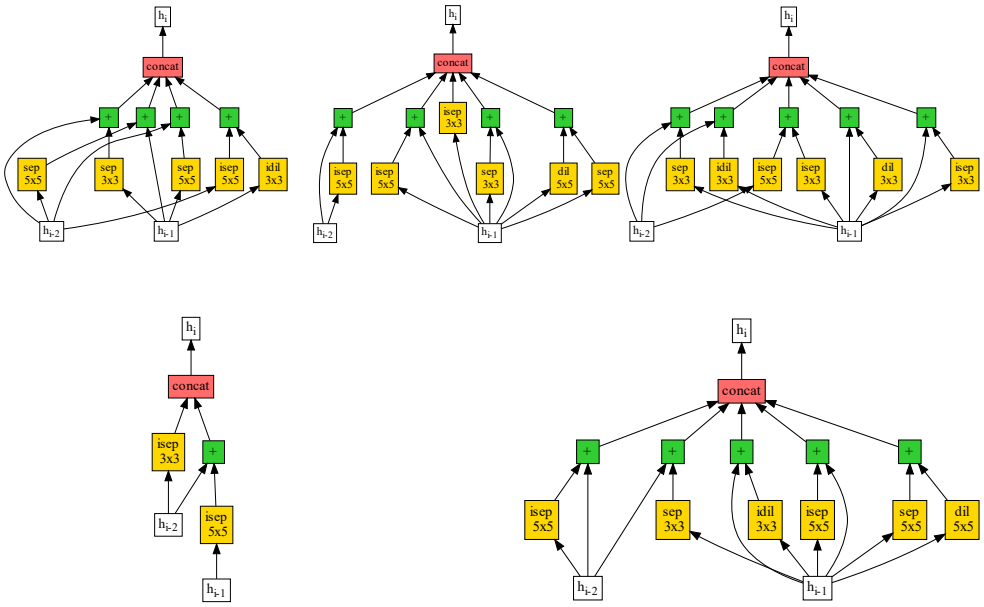


Figure 7.29: Best candidate from CSTENAS with finetuning in CIFAR-100 dataset

abdominal organs. These tasks are evaluated in our work, being briefly described below along with the results obtained:

7.6.1 LIVER-CT: Liver segmentation in CT scans (Task 2)

The most simple task, LIVER-CT, focuses on the segmentation task in CT images. The CT images contain only the ground truth of the liver. Thus a CT-only task cannot evaluate the segmentation of other organs. LIVER-CT is the less demanding task in this challenge and with the most significant number of participants. Also, the overall results obtained in this task were higher (detailed evaluations on the tasks can be seen in (Kavur et al., 2021)).

Table 7.16 shows the average Dice score for NASGEP and CSTENAS. Also, Table 7.17 details the distribution of the twenty cases.

Table 7.16: Results obtained in the LIVER-CT task from the CHAOS challenge

Strategy	**DSC (in %)**	**# Param (in M)**	**# MAdds (in G)**
NASGEP	95.39 ± 2.86	5.040	27.61
CSTENAS	95.82 ± 1.87	7.223	50.24

Table 7.17: Distribution of the LIVER-CT task from the CHAOS challenge

Strategy	**Median**	**Q1**	**Q3**	**Min**	**Max**	**Mean**	**STD**
NASGEP	96.36%	95.09%	97.12%	86.66%	97.91%	95.39%	2.86%
CSTENAS	96.16%	95.33%	96.92%	89.51%	97.85%	95.82%	1.87%

Table 7.15: Changes to the hyper-parameters for the medical semantic segmentation task

Parameter	New value
Stride	levels = [2, 2, 2, 2, 1]
Optimizer	AdamW
Scheduler	Cosine one-cycle
Epochs	100
Learning rate	3e-4
Loss function	Soft-Dice and cross-entropy
Preprocessing	Auto-contrast Resize to 512×512 for tasks with MRI
Data augmentation	Random resize @ [0.5×, 1.5×] Random crop @ *original size* (after resizing) Curvature flow @ *time step*= 0.125, *iterations*= 5 Random shear @ $[-10°, +10°]$ Random translation @ *offset*= 0.1 Random rotate @ $[-30°, +30°]$ Random elastic deformation @ $\alpha = 1.5$, $\sigma = 0.07$

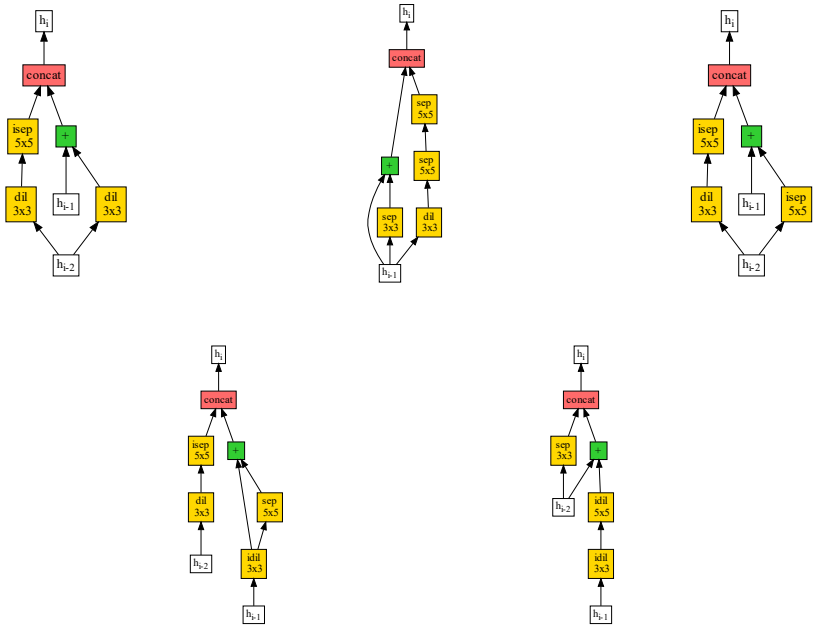


Figure 7.30: Candidate generated by the NASGEP approach for the liver segmentation task with CT

Although little difference is seen between the two approaches, CSTENAS had a slightly better DSC in mean and standard deviation (more stable). One thing to note is that NASGEP generated a model with only two-block cells (illustrated in Figure 7.30). As for CSTENAS (Figure 7.31), cells with different blocks were generated. Although in the CIFAR-10 experiments, two-block cells had inferior results, NASGEP returned this type of model with comparable results to a model with more diversity.

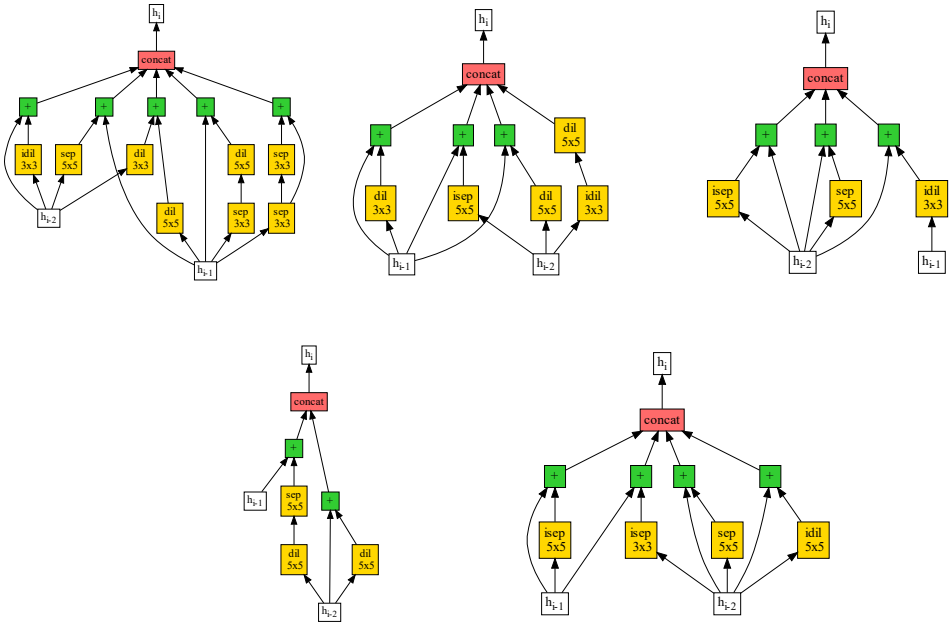


Figure 7.31: Candidate generated by the CSTENAS approach for the liver segmentation task with CT

7.6.2 LIVER-MRI: Multi-modal liver segmentation in MR image (Task 3)

The MR liver segmentation task is more challenging than LIVER-CT (Task 2, in the original paper), mainly because of the absence of standardization like the Hounsfield Units (HU). Although the ground truth of other organs is present, it is crucial to evaluate liver segmentation in a different image modality. One thing to note is that the patients of the CT scans are different from those who had taken the MRI. Although direct comparison is not possible, this would enable a more robust evaluation in latter tasks which use both modalities (i.e., cross-modality). Also, it is expected of the approach to work on T1-DUAL (in-phase and oppose-phase) and T2-SPIR images.

T1 has in-phase and out-phase as its two channels, with T2 replicating the slice for both channels. Table 7.18 shows the results for both proposed NAS approaches, with Table 7.19 detailing the distribution of the forty cases (20 T1 cases plus their respective T2 exams). Overall, we had lower DSC in this task than the LIVER-CT task.

Table 7.18: Results obtained in the LIVER-MRI task from the CHAOS challenge

Strategy	**DSC (in %)**	**# Param (in M)**	**# MAdds (in G)**
NASGEP	91.11 ± 10.36	6.733	36.56
CSTENAS	93.13 ± 3.04	6.735	37.45

As we can see, both approaches found similar models in terms of parameters and MAdds. NASGEP (Figure 7.32) performed badly in relation to CSTENAS (Figure 7.33). Although T1 for NASGEP has a slightly superior mean DSC, its standard deviation increased considerably because of an exam with very low DSC (lowest with 37.5%).

Table 7.19: Distribution of the LIVER-MRI task from the CHAOS challenge

Strategy	Modality	Median	Q1	Q3	Min	Max	Mean	STD
NASGEP	ALL	93.83%	92.33%	95.29%	37.50%	96.39%	91.11%	10.63%
	T1	94.37%	93.25%	95.36%	37.50%	95.94%	91.36%	12.76%
	T2	92.97%	91.34%	94.97%	67.94%	96.39%	90.87%	7.59%
CSTENAS	ALL	93.87%	92.45%	94.93%	80.06%	95.88%	93.13%	3.05%
	T1	93.95%	92.66%	95.06%	84.75%	95.70%	93.31%	2.70%
	T2	93.76%	92.11%	94.80%	80.06%	95.88%	92.94%	3.60%

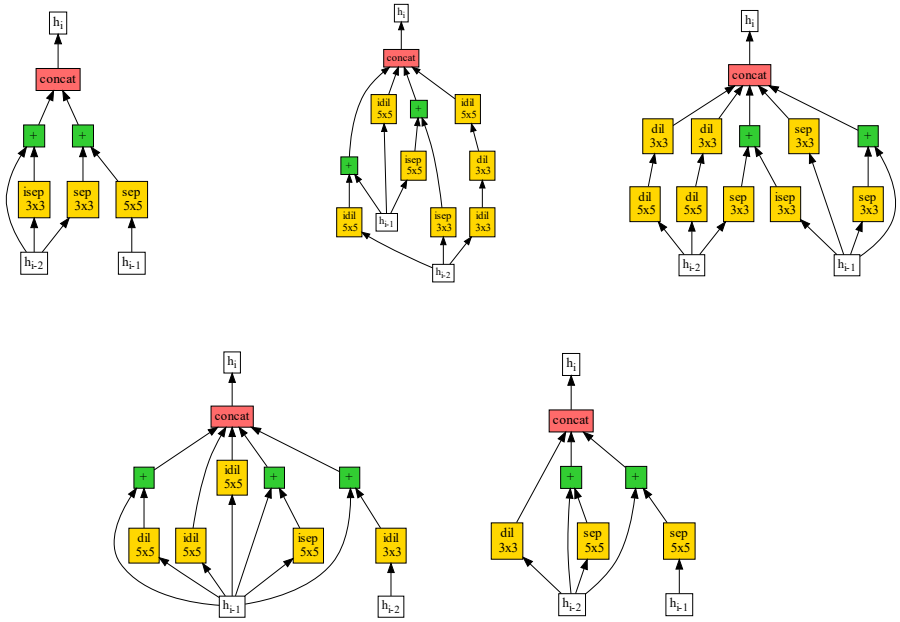


Figure 7.32: Candidate generated by the NASGEP approach for the liver segmentation task with MRI (T1 and T2)

7.6.3 LIVER-CROSS: One model for liver segmentation in CT and MR images (Task 1)

This task evaluates a liver segmentation approach that works both in CT and MRI without switching approaches based on the modality. Specifically for our approach, the objective is to discover a segmentation model which works for both modalities (CT and MRI) without any modification in the structure. Thus, this task focus on the performance of cross-modality approaches.

Table 7.20: Results obtained in the LIVER-CROSS task from the CHAOS challenge

Strategy	DSC (in %)	# Param (in M)	# MAdds (in G)
NASGEP	94.96 ± 2.26	5.614	26.73
CSTENAS	94.44 ± 4.49	5.400	31.37

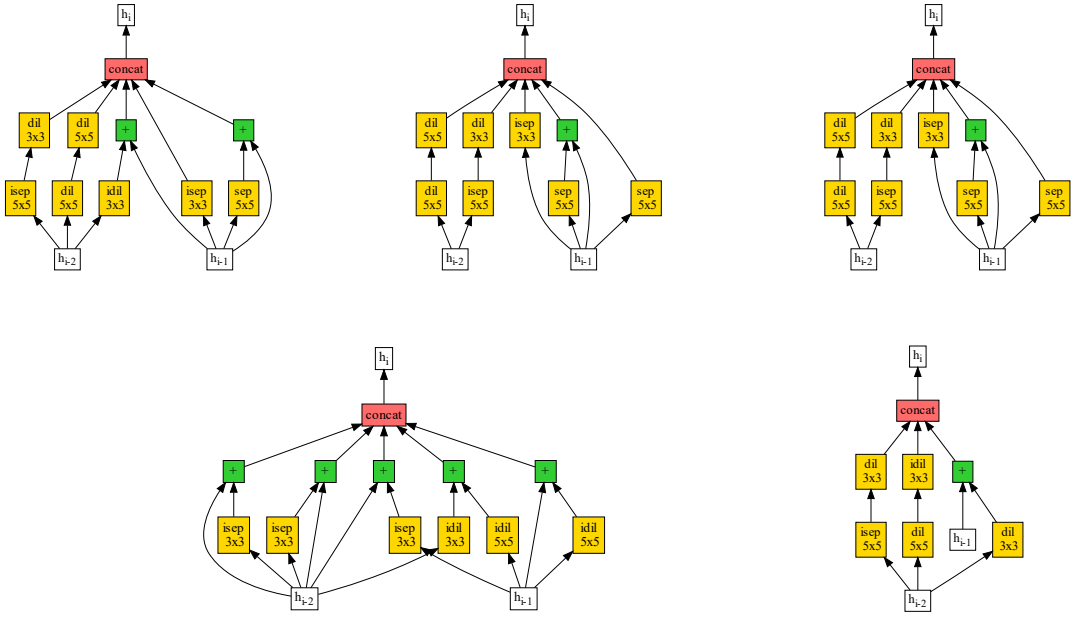


Figure 7.33: Candidate generated by the CSTENAS approach for the liver segmentation task with MRI (T1 and T2)

Table 7.21: Distribution of the LIVER-CROSS task from the CHAOS challenge

Strategy	Modality	Median	Q1	Q3	Min	Max	Mean	STD
NASGEP	ALL	95.25%	94.14%	96.36%	85.39%	98.06%	94.96%	2.27%
	T1	94.70%	93.75%	95.09%	89.29%	96.36%	94.17%	1.69%
	T2	94.65%	92.76%	96.20%	85.39%	96.64%	94.14%	2.69%
	CT	97.11%	96.22%	97.51%	93.63%	98.06%	96.83%	1.06%
CSTENAS	ALL	95.28%	93.79%	96.17%	64.10%	98.04%	94.44%	4.49%
	T1	94.13%	93.06%	95.04%	64.10%	96.04%	92.61%	6.81%
	T2	94.88%	92.76%	95.60%	87.37%	96.21%	93.92%	2.39%
	CT	97.16%	96.11%	97.46%	94.40%	98.04%	96.80%	0.96%

In Table 7.20, we can see that NASGEP had a model slightly superior to CSTENAS, with the latter having a higher standard deviation. In both approaches, the CT modality had better results. CSTENAS had worse results mainly caused by the same exam from the previous task. In this case, CSTENAS had a DSC of 64.10% for this exam.

Analyzing Tables 7.17, 7.19 and 7.21, we can see a small improvement in the usage of cross-modality training. Since MRI had worse results than CT, the overall DSC for the cross-modality was lower than LIVER-CT.

7.6.4 ABDOMEN-MRI: Abdominal organ segmentation in MRI (Task 5)

This task is similar to LIVER-MRI (Task 3). However, segmentation is employed in the four organs. It may be more challenging than other tasks because the regions of interest have different structures and patterns. The impact of liver segmentation with another three classes – left kidney, right kidney, and spleen – is evaluated in this task.

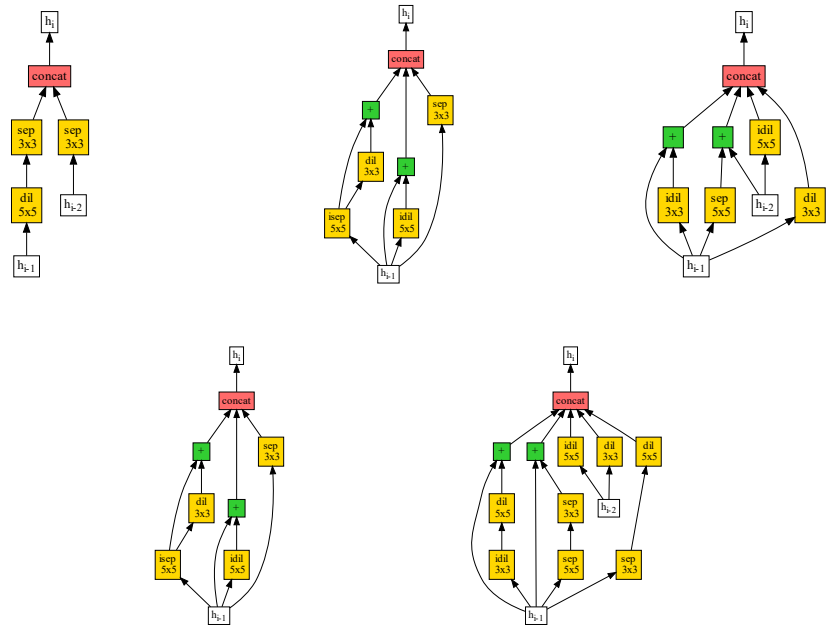


Figure 7.34: Candidate generated by the NASGEP approach for the liver segmentation task with cross-modality (CT and MRI)

Table 7.22 shows the results obtained. Overall, this task had the worst scores for both approaches. Furthermore, both approaches performed similarly.

Table 7.22: Results obtained in the ABDOMEN-MRI task from the CHAOS challenge

Strategy	**DSC (in %)**	**# Param (in M)**	**# MAdds (in G)**
NASGEP	89.62 ± 6.73	5.557	32.27
CSTENAS	89.57 ± 6.20	4.090	26.14

As we can see in Table 7.23, both NASGEP (Figure 7.36) and CSTENAS (Figure 7.37) performed better in T2 than T1. The combination of T1 in-phase and out-phase may have hindered the overall results in this modality. One alternative is to provide a different initial convolutional block for T1 to combine the in-phase and out-phase data. Overall, our models performed worse in MRI than CT. It is also possible that the MRI cases may have more complexity than the CT scans.

Table 7.23: Distribution of the ABDOMEN-MRI task from the CHAOS challenge

Strategy	**Modality**	**Median**	**Q1**	**Q3**	**Min**	**Max**	**Mean**	**STD**
NASGEP	ALL	91.95%	89.13%	93.53%	65.20%	95.78%	89.63%	6.91%
	T1	90.42%	87.60%	93.04%	69.22%	94.76%	88.45%	6.82%
	T2	92.71%	89.61%	94.15%	65.20%	95.78%	90.80%	6.61%
CSTENAS	ALL	91.51%	88.53%	93.47%	62.02%	95.70%	89.58%	7.07%
	T1	90.18%	88.50%	92.57%	62.02%	94.89%	88.14%	8.59%
	T2	92.53%	88.78%	94.13%	74.23%	95.70%	91.02%	5.10%

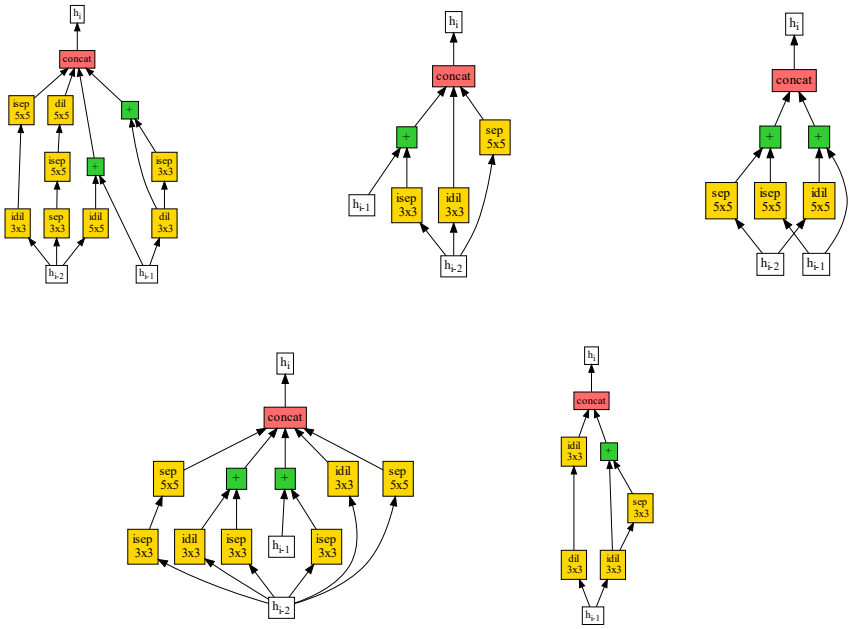


Figure 7.35: Candidate generated by the CSTENAS approach for the liver segmentation task with cross-modality (CT and MRI)

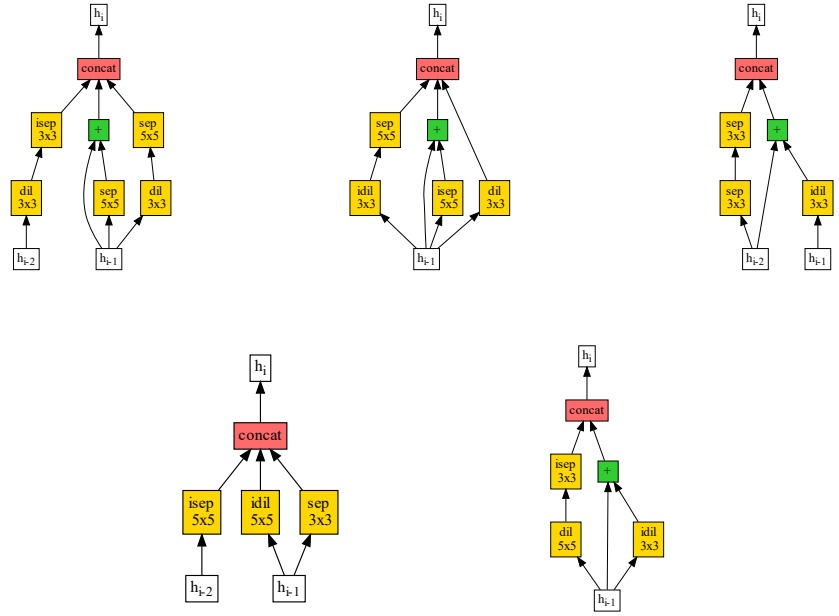


Figure 7.37: Candidate generated by the CSTENAS approach for the abdomen segmentation task with multi-modality MRI (T1-DUAL and T2-SPIR)

7.6.5 ABDOMEN-CROSS: Abdominal organ segmentation in both CT and MRI (Task 4)

The last and most challenging task is segmentation of four organs in MR images and liver segmentation in CT images with the same model. Unlike LIVER-CROSS (Task 1), adding three more classes in MRI may introduce a much more challenging task to be addressed. Model fusion with shared preprocessing block the modality type of MRI can be implemented here. However,

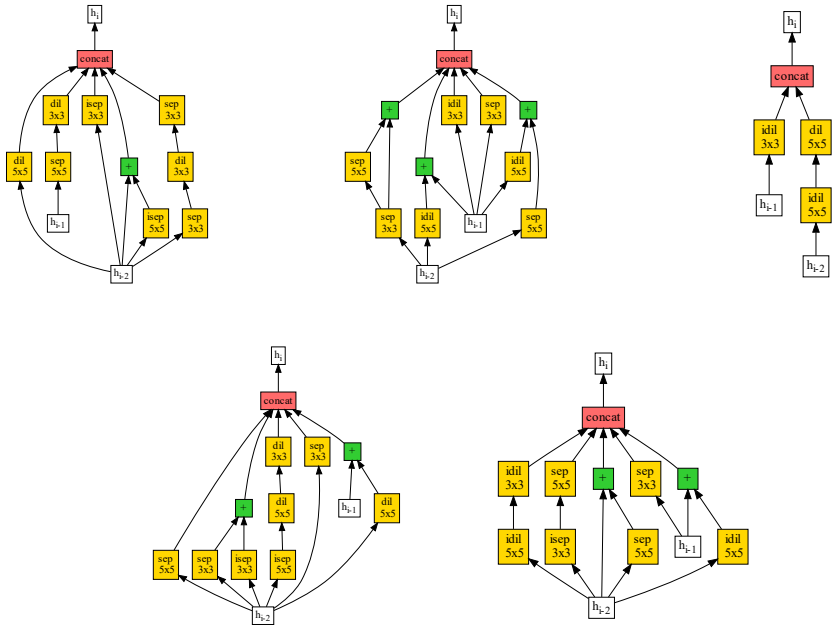


Figure 7.36: Candidate generated by the NASGEP approach for the abdomen segmentation task with multi-modality MRI (T1-DUAL and T2-SPIR)

we followed the same pattern in obtaining a single model with NAS to segment the different modalities and organs.

The results can be seen in Table 7.24. Overall, no improvement was found for MRI in relation to the ABDOMEN-MRI task (see Tables 7.23 and 7.25). Also, DSC for CT images is similar to the LIVER-CROSS task.

Table 7.24: Results obtained in the ABDOMEN-CROSS task from the CHAOS challenge

Strategy	**DSC (in %)**	**# Param (in M)**	**# MAdds (in G)**
NASGEP	91.51 ± 7.42	5.593	33.89
CSTENAS	91.83 ± 6.69	5.837	35.67

Both NASGEP (Figure 7.38) and CSTENAS (Figure 7.39) performed better in CT than MRI. Overall, we had better averages and standard deviations in CT. Segmentation of many organs in MRI provided a more complex task for our approaches.

7.6.6 Overall results

In general, both approaches had similar results. The main difference is in the LIVER-MRI task, in which CSTENAS had a superior score. In this small experiment, it is clear that MRI is more sensitive in segmentation. A more detailed evaluation of the results obtained in MRI is necessary to assess the higher deviation found.

Another thing to note is that, overall, the cells in the CHAOS challenge had more stacked convolutions. In this case, cells are more profound than those found in the CIFAR datasets. Our custom U-Net is more shallow than the image classification networks – nine against 20 sequential cells. Also, the macro-structure has skip connections (decoder part receiving outputs from the encoder). Since we have a more shallow network that also benefits from residual connections, our NAS approaches focused on cells with more depth.

Table 7.25: Distribution of the ABDOMEN-CROSS task from the CHAOS challenge

Strategy	Modality	Median	Q1	Q3	Min	Max	Mean	STD
NASGEP	ALL	93.21%	90.46%	96.01%	61.70%	98.08%	91.32%	7.48%
	T1	90.76%	87.18%	92.93%	63.31%	94.83%	88.06%	8.22%
	T2	92.48%	89.99%	93.89%	61.70%	95.89%	89.76%	8.21%
	CT	96.72%	96.17%	97.15%	93.56%	98.08%	96.61%	0.96%
CSTENAS	ALL	93.55%	89.96%	96.04%	65.55%	97.94%	91.84%	6.70%
	T1	90.18%	88.55%	92.88%	65.55%	94.74%	88.65%	6.54%
	T2	92.74%	88.82%	94.38%	67.94%	95.73%	90.12%	7.52%
	CT	97.08%	96.04%	97.19%	93.38%	97.94%	96.74%	1.00%

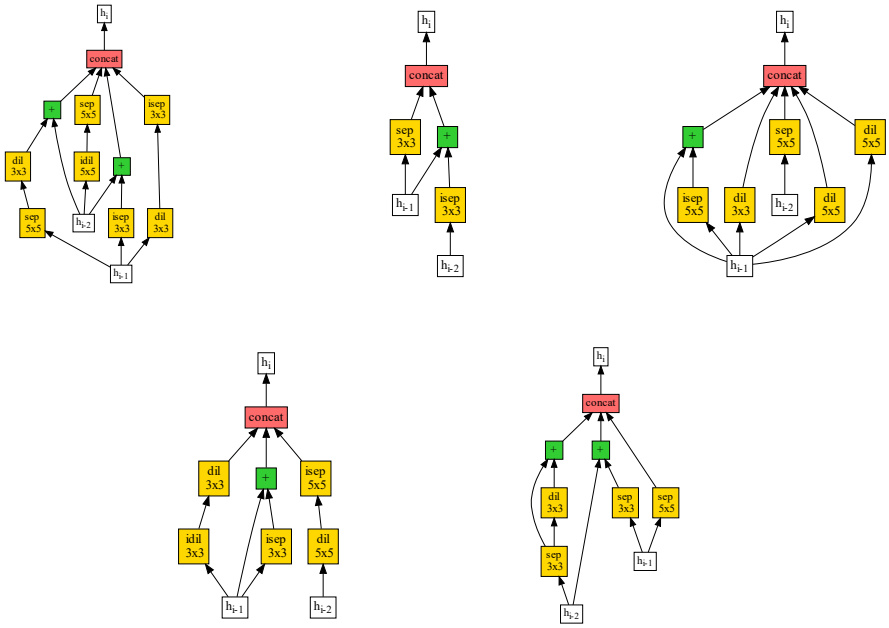


Figure 7.38: Candidate generated by the NASGEP approach for the abdomen segmentation task with cross-modality (CT and MRI)

7.7 CONCLUSION

In this chapter, we describe the experiments conducted in this work. The hyperparameters, computational resources, and the motivation behind the ablation study were discussed. Next, we have shown the results obtained and some comparisons to understand them better.

This chapter presents the results obtained in our approaches for image classification and medical semantic segmentation. For CIFAR-10, our best experiment had an error rate of 2.17% ± .10 with CSTENAS. For CIFAR-100, an error rate of 15.47% ± .51 with NASGEP. As for the CHAOS challenge, CSTENAS was slightly better than NASGEP, mainly because of the LIVER-MRI task. In this task, CSTENAS had a DSC of 93.13% ± 3.04 against 91.11% ± 10.36. Overall, CSTENAS showed to be more stable (lower standard deviation) than NASGEP.

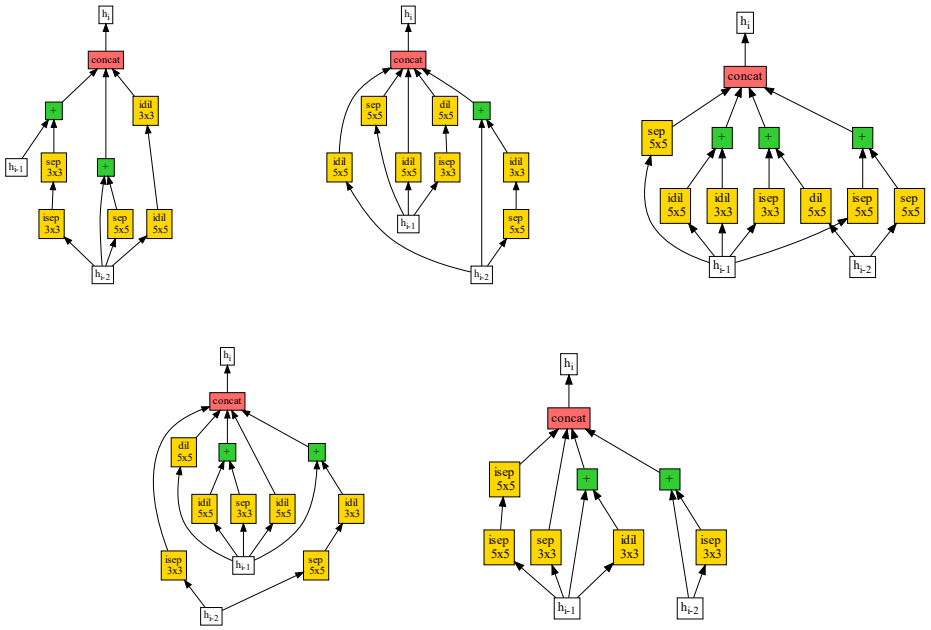


Figure 7.39: Candidate generated by the CSTENAS approach for the abdomen segmentation task with cross-modality (CT and MRI)

8 OVERALL DISCUSSION

This chapter presents the overall discussion of our results and related works regarding the CIFAR datasets and the CHAOS challenge. We discuss the pros and cons of our approaches compared with other NAS works. We also provide some discussions about the employment of NAS and the further directions to be addressed.

8.1 CIFAR DATASETS

In Table 8.1 we present the results from reviewed NAS works compared with our two methods in the CIFAR-10 dataset. We present three metrics to evaluate the proposed NAS works: test error (in %), search cost (in GPU days), and the number of parameters (in millions). With these three metrics, a better comparison can be made.

Regarding the test error in CIFAR-10, our approaches had good numbers, being competitive concerning the state-of-the-art NAS methods. The best NAS methods were NAT (Lu et al., 2021) (error rate of 1.6%), NoisyDARTS (Chu et al., 2020) (error rate of 1.72%) and FairNAS (Chu et al., 2019) (error rate of 1.9 ± 0.1). These approaches combined NAS with transfer learning. Although a direct comparison with other approaches can not be made, since these methods used transfer learning for error reduction, they are the SOTA on generated models with less than 10M parameters. They encourage the combination of NAS with transfer learning to obtain better performance.

Even without transfer learning, other works had competitive results. For example, sharpDARTS (Hundt et al., 2019) had an error rate of 1.98 ± 0.07, similar to FairNAS. NoisyDARTS without transfer learning had an error rate of 2.47%. Even if transfer learning had a big influence on NoisyDARTS performance, a competitive model could be found with this approach. ProxylessNAS (Cai et al., 2018c), a NAS approach with a similar search space than NAT and FairNAS, had an error rate of 2.08. AmoebaNet (Real et al., 2018) had an error rate of 2.13 ± 0.04, and EvNas (Sinha and Chen, 2020) had an error rate of 2.57 ± 0.09, showing the capability of evolutionary approaches in obtaining good and stable generated models. Then, PLNT (Cai et al., 2018b) had an error rate of 2.30%.

NAT and FairNAS had impressive results (although they also used transfer learning). These approaches had a search space based on the MobileNetV2 – a sequential network with inverted residual blocks (using different kernel sizes and inclusion of expansion ratio to increase the number of channels in depth-wise convolutions) and residual connections. Also, images are resized to a similar size to ImageNet-based models.

In terms of error rate, NASGEP and CSTENAS are on pair to state-of-the-art NAS methods. Although the best approaches had significant error reduction compared with NASGEP and CSTENAS, our approaches do not use transfer learning. Comparing them with NoisyDARTS without transfer learning, NASGEP and CSTENAS had better accuracy. Regardless, transfer learning can be employed in the best-searched models to reduce the overall error. Although it is better to report results without transfer learning (for a better evaluation of the search strategy and space), we can use any NAS approach combined with transfer learning (e.g., training in ImageNet and finetuning in another specific dataset for image classification). The only problem with this approach is the additional cost of training using a large dataset.

Table 8.1: Comparison of NAS works in the CIFAR-10 dataset with our approaches, reporting error (in %), search phase time (in GPU days), and the number of parameters (in millions) of the searched fully-trained model(s). * denotes results with a model using transfer learning (trained on ImageNet and finetuned on CIFAR-10). NR denotes for Not Reported.

Approach	**Error**	**Cost**	**# Param**
RNN & REINFORCE (Zoph and Le, 2016)	3.65	1800	37.4
MetaQNN (Baker et al., 2016)	9.09 ± 1.68	80~100	3.71 ± 4.21
CNF (Saxena and Verbeek, 2016)	7.43	NR	21.2
Hierarchical Efficient AS (Liu et al., 2017b)	3.63 ± 0.10	300	NR
NAS (Evolutionary) (Real et al., 2017)	5.4	NR	5.4
CoDeepNEAT (Miikkulainen et al., 2017)	7.3	NR	NR
NASH (Elsken et al., 2017)	4.4	1	88
NASNet (Zoph et al., 2017)	2.40	83	27.6
PNAS (Liu et al., 2017a)	3.41 ± 0.09	17	3.2
CGP (Suganuma et al., 2017)	5.66	20~24	1.50~2.84
AmoebaNet (Real et al., 2018)	2.13 ± 0.04	3150	34.9
PPP-Net (Dong et al., 2018)	4.36	NR	0.52
EAS (Cai et al., 2018a)	3.44	< 10	10.7
BlockQNN (Zhong et al., 2018)	3.54	96	39.8
Efficient NAS (Pham et al., 2018)	2.89	0.5	4.6
DARTS (Liu et al., 2018)	2.76 ± 0.09	4	3.3
NASBOT (Kandasamy et al., 2018)	12.3 ± 0.3	1.17	NR
BOHB (Zela et al., 2018)	3.18 ± 0.16	32	27.6
Memetic evolution (Lorenzo and Nalepa, 2018)	27.73	0.08	NR
PLNT (Cai et al., 2018b)	2.30	8.33	14.3
One-shot AS (Bender et al., 2018)	3.9 ± 0.02	3.33	19.3 ± 0.6
ProxylessNAS (Cai et al., 2018c)	2.08	8	5.7
SNAS (Xie et al., 2018)	2.85 ± 0.02	3	2.8
sharpDARTS (Hundt et al., 2019)	1.98 ± 0.07	1.2	3.6
LEMONADE (Elsken et al., 2019)	3.6	56	3.4
GDAS (Dong and Yang, 2019)	2.82	0.17	2.5
FairNAS (Chu et al., 2019)	1.9 ± 0.1*	10	5.73 ± 0.15
SGAS (Li et al., 2020)	2.66 ± 0.24	0.25	3.7
NoisyDARTS (Chu et al., 2020)	2.47 & 1.72*	0.29	3.01 & 4.3
EvNas (Sinha and Chen, 2020)	2.57 ± 0.09	4.4	3.6 ± 0.2
NAT (Lu et al., 2021)	**1.6***	65.62	6.9
Proposed approaches			
NASGEP	2.31 ± 0.16	0.5	4.41 ± 0.54
CSTENAS	2.17 ± 0.10	0.5	4.79 ± 0.45

Search cost in GPU days is one of the essential things for NAS evaluation. Practical use of NAS approaches with high cost is not viable. In the first years of NAS, many methods consumed weeks or even years of GPU (in some cases, the search cost was not reported or clearly stated). It is crucial for NAS approaches to employ the search phase in a viable time. The requirement of several GPU days restricts NAS employment to only big companies and labs. Thus, we evaluate NAS approaches based on the search cost, encouraging robust approaches in

a few days of search. We specifically discuss works within one GPU day and one GPU week before an overall discussion for a better comparison.

In the reviewed methods, the works that employ the search phase within one GPU-day are NASH (Elsken et al., 2017), Efficient NAS (Pham et al., 2018), Memetic evolution (Lorenzo and Nalepa, 2018), GDAS (Dong and Yang, 2019), SGAS (Li et al., 2020) and NoisyDARTS (Chu et al., 2020). Only a few works aim for a fast search phase. Although we may think that search time is highly correlated with high accuracy, this is not always true. For example, except for NASH and Memetic evolution, the remaining methods had error rates less than 3%. Also, NASGEP and CSTENAS had impressive results within 12 GPU hours. The two exceptions (NASH and Memetic evolution) are relatively old approaches that may not be well adapted regarding the challenges of the NAS field. Well-planned search strategies and a good search space may reduce the requirement of several GPU days of searching.

Regarding NAS approaches within one GPU-week, besides the ones that were already discussed, there are DARTS (Liu et al., 2018), NASBOT (Kandasamy et al., 2018), One-shot AS (Bender et al., 2018), SNAS (Xie et al., 2018), sharpDARTS (Hundt et al., 2019) and EvNas (Sinha and Chen, 2020). Overall, the results are similar. Some methods performed poorly, but others had state-of-the-art results like sharpDARTS.

We highlight AmoebaNet with more than three thousand GPU days spent for the remaining methods. AmoebaNet is an evolutionary-based NAS that generated impressive models with a high computational cost. In their days, evolutionary-based NAS was highly correlated with excessive cost (Ren et al., 2021). However, we can see that evolutionary-based approaches like EvNAS and the proposed methods (i.e., NASGEP and CSTENAS) can return competitive models within a few GPU days with efficient strategies and reduced search space. Furthermore, other robust approaches like FairNAS and NAT used more than one GPU week. Specifically, NAT consumes two months' worth of GPU to employ a search phase based on transfer learning.

Gradient-based methods employ architecture search in a specific DAG search space (DARTS-like). They generally involve training a supernet and robust weight sharing/inheritance, enabling search time reduction without hindering the final results. Although evolutionary-based, our approaches focus on weight sharing/inheritance, reduced individual training (five epochs in NASGEP), and dynamic training to reduce over-fitting (in the case of CSTENAS).

Finally, we evaluate the number of parameters from the fully-trained searched models. Overall, NAS methods focus on fully training models with less than 10M parameters. It is important to evaluate models with similar sizes since larger models generally have better accuracy (though more slowly). The best models have parameters between three and seven million parameters. In some cases, we report models with more than 20 million parameters (e.g., NASNet (Zoph et al., 2017) and AmoebaNet (Real et al., 2018)).

To clarify the influence in the error rate regarding the model size, we can see the results of AmoebaNet with different initial widths. For models sizes of 2.8M, 13.7M and 26.7M, AmoebaNet had the error rates of 2.55% ± 0.05, 2.31% ± 0.05 and 2.21% ± 0.04, respectively. Although model size dramatically impacts the performance, significant increases in its size may be necessary for a relevant improvement.

In Table 8.2 we show the NAS works employed in CIFAR-100. This dataset is more complex since there are more classes than CIFAR-10 with the same number of samples for the entire dataset. Furthermore, fewer NAS works are applied in this dataset.

NAT (Lu et al., 2021) and FairNAS (Chu et al., 2019) are the top NAS work in CIFAR-100, employing transfer learning to increase the accuracy. BlockQNN (Zhong et al., 2018), NASH (Elsken et al., 2017) and GDAS (Dong and Yang, 2019), where transfer learning

Table 8.2: Comparison with the SOTA in CIFAR-100 with error (in %), search phase time (in GPU days) and number of parameters (in millions) of the fully-trained searched model(s). * denotes results with a model using transfer learning (trained on ImageNet and finetuned on CIFAR-100).

Approach	Error	Cost	# Param
MetaQNN (Baker et al., 2016)	27.14	80~100	11.18
NAS (Evolutionary) (Real et al., 2017)	23	NR	5.4
NASH (Elsken et al., 2017)	19.6	1	88
BlockQNN (Zhong et al., 2018)	18.06	96	39.8
GDAS (Dong and Yang, 2019)	18.13	0.17	2.5
FairNAS (Chu et al., 2019)	13 ± 0.3*	10	5.73 ± 0.15
NAT (Lu et al., 2021)	**11.7***	65.62	6.9
Proposed approaches			
NASGEP	15.47 ± 0.51	0.5	4.78 ± 0.29
CSTENAS	15.84 ± 0.15	0.5	4.86 ± 0.59

was not used, had similar results. Our approaches, NASGEP and CSTENAS, had better results than these, although usage of AutoAugment may have increased the difference.

In these experiments, we evaluate the performance of NAS works in the CIFAR datasets. There are distinct variables that may improve the overall results of NAS approaches. Robust weight-sharing policies, reduced individual model training (in the search phase), heavy data augmentation, and transfer learning are some strategies that may aid in obtaining efficient models. Although recent works focus on gradient-based methods for a fast and efficient search, these methods are not required to obtain robust models within a few GPU days.

8.2 CHAOS CHALLENGE

We evaluate different methods employed in the CHAOS challenge, manually designed or automatically generated models. Although few NAS works were employed in this challenge, we can evaluate their performance concerning manually designed models. Almost every model is based on U-Net. Minimal optimization is employed in NASGEP and CSTENAS to adapt to this challenge. The idea is to employ NAS methods developed based on image classification without tweaking the medical semantic segmentation problem. Table 8.3 shows the overall results from the studied approaches and our proposed methods.

More complexity is found in the cross-modality and multi-organ tasks regarding the CHAOS challenge tasks. However, our approaches had results comparable with other robust approaches applied in this challenge. Also, we had superior results in the cross-modality tasks compared with the approaches employed in the initial challenge.

One problem in the surveyed methods is the absence of model size and computation reporting. For example, nnU-Net employs an ensemble of U-Net models. This approach may present a large model if compared with the others. Our approaches, for example, generate models with less than 10M parameters. BiX-NAS employs a model with 0.38M parameters since they use a recurrent approach to reduce parameter size, but it has similar MAdds to our approaches (28 billion). The authors of this approach (Wang et al., 2021) reported large model sizes and computations for NAS-UNet (2.42M parameters and 67.31G MAdds), AutoDeepLab (27.13M parameters and 60.33G MAdds), and MS-NAS (14.08M parameters and 72.71G MAdds). For reference, they reported 8.64M parameters and 65.83G MAdds for the vanilla U-Net model.

Table 8.3: Summary of related works surveyed for the CHAOS challenge compared with our proposed NAS approaches (* denotes for reported results in validation sets only, ** denotes for an average between the results of each organ in validation sets, † denotes results reported in (Yan et al., 2020), and ‡ denotes results reported in (Wang et al., 2021)).

Approach	**Liver**			**Abdomen**	
	CT	**MRI**	**ALL**	**MRI**	**ALL**
Manually Designed					
OvGUMEMoRIAL	.90 ± .21	.81 ± .15	.88 ± .15	.79 ± .15	.85 ± .16
ISDUE	.91 ± .04	.85 ± .11	.87 ± .16	.83 ± .23	.85 ± .21
Lachinov	.83 ± .20	.90 ± .05	.87 ± .13	-	-
IITKGP-KLIV	-	.63 ± .07	.72 ± .31	.56 ± .06	.63 ± .36
METU_MMLAB	-	.89 ± .03	.86 ± .09	.89 ± .03	-
PKDIA	**.98 ± .00**	.94 ± .01	.85 ± .26	.93 ± .02	.88 ± .21
MedianCHAOS	**.98 ± .00**	-	-	-	-
Mountain	-	.92 ± .02	-	.90 ± .03	-
CIR_MPerkonigg	-	.91 ± .07	-	-	-
NAS					
nnU-Net	-	**.95 ± .01**	-	**.95 ± .02**	-
NAS-UNet*	.974	-	-	.76	-
BiX-NAS*	-	.90 ± .01	-	.8277**	-
AutoDeepLab*†	-	.93 ± .01	-	.8752**	-
*‡	-	.88 ± .02	-	.7857**	-
MS-NAS*†	-	.94 ± .00	-	.9025**	-
*‡	-	.83 ± .02	-	.7990**	-
Proposed NAS					
NASGEP	.95 ± .03	.91 ± .10	**.95 ± .02**	.90 ± .07	**.92 ± .07**
CSTENAS	.96 ± .02	.93 ± .03	.94 ± .04	.90 ± .06	**.92 ± .07**

In the MS-NAS paper (Yan et al., 2020), there were some differences regarding these models. The reported parameters were: 12.45M for NAS-UNet, 14.45M for AutoDeepLab, and 11.51M for MS-NAS (the proposed approach with five paths). As we can see in Table 8.3, experiments employed in the (Yan et al., 2020) paper had superior Dice scores. It was unclear why this difference between the two papers, although on how the cross-validation was executed, model size, data augmentations, and other configurations may have impacted these results. Also, it is encouraged to report results from the evaluation test server since it would provide better comparison and reduce sample peeking. The remaining approaches were not clear regarding these metrics.

In comparison with these models, both NASGEP and CSTENAS obtained superior results. However, our approaches had inferior results compared to nnU-Net. Employment of ensemble or even a focus on MRI may have contributed to nnU-Net attaining a better Dice score. Further investigation in the employment of ensemble or even large models may improve our proposed NAS methods.

Also, nnU-Net trains only one model in the ABDOMEN-MRI task and uses liver prediction for the other task. Since we focused on evaluating the generated models in the

evaluation test server, there is little information about individual organ results besides the other tasks focused on the liver. Furthermore, each task has a different searched model. Further experiments employing K-fold cross-validation for organ segmentation evaluation and evaluating each search model on all tasks may clarify the influence of different networks and tasks in our methods.

Although extensive optimizations and evaluations in the CHAOS and other medical challenges are required to better assess our NAS approaches, these works can be employed as an initial step in medical semantic segmentation problems. Then, further improvements can be applied to obtain more efficient models.

8.3 USAGE OF NAS AND FOLLOWING DIRECTIONS

Several GPU days spent to search good models may not be necessary. Overall, good results can be found with a few GPU days. How a search space is designed and effective search strategies may be more helpful than to spent excessive computational resources. For example, cell-based search space (Inception-like) or MobileNetV2 search space may provide good candidate models. Searched models combined with heavy data augmentation and transfer learning have significant improvements.

It is unnecessary to fully train candidate models in evolutionary-based NAS or spend considerable resources on each candidate. Weight sharing/inheritance provides a fast way to obtain optimized models (although not entirely). NAT, EvNAS, and the proposed approaches show that hundreds of days are not required to employ evolutionary-based NAS with efficient searched models.

Furthermore, NAS is not a replacement for transfer learning. The idea of NAS is to obtain better structures based on a specific problem. However, it is possible to further optimize a network by training it in a large dataset and finetune it in the specific dataset of interest. Thus, more efficient models can be obtained. Although this process is not obligatory to make NAS useful, it is an excellent choice to reduce model error (e.g., NAT, NoisyDARTS, and FairNAS) if resources are available.

We employed two different evolutionary-based NAS approaches. NASGEP uses the standard workflow of evolutionary search, training one epoch of each candidate at a time. Newer candidates benefit from the pre-optimized weights from other candidates to optimize their weights faster. CSTENAS is a dynamic approach that trains the population jointly, employing a dynamic training similar to NAT. However, CSTENAS executes the search in a cell-based search space and combines candidates, with each training batch having cells from different candidates. The idea is to reduce the bias from combinations of cells. Also, NAT employs a search strategy based on transfer learning. Thus, NAT needs several weeks to optimize the weights for application in different datasets. Both NASGEP and CSTENAS do not follow this strategy, reducing the search cost within twelve GPU hours. Both approaches can easily be employed in different datasets and tasks with a fast and low-cost search phase. Also, we can apply transfer learning in the training phase to reduce model error.

Another thing to note is that little difference is found in NASGEP and CSTENAS results. At first, exotic training approaches may present little difference concerning a more standard training approach. Further research may be necessary to understand better the influence of training candidate models in evolutionary-based NAS. Also, better candidate evaluation methods besides the validation error can be used. For example, NAT employs surrogate models to predict the error of candidate models if they were fully trained. Further research in efficient surrogate

models applied to NAS and how the training strategy influences their prediction can help to improve the NAS field.

With fewer resources required in the search phase, we can develop different strategies to obtain networks less like manually designed ones but more effective. We employed a different cell for each of the five stages in our experiments. Each stage has a different channel size, cell repetition, and feature map size. Reducing the cost in the search phase with fixed structures may open the way to increase the complexity of the designed search spaces. For example, each stage may have dynamic configurations optimized in the search phase. This approach would increase the probability of finding better models that in common search spaces would not be found.

Some directions can be followed for further research and improvements in the NAS field and, especially, in the proposed approaches:

- Further research in the training loss as the fitness function, reducing the requirement of validation steps (consequently reducing search cost);
- Expanding our approaches to work with other network types like recurrent and convolutional networks focused on object detection and instance segmentation to increase the different problems that our approaches can be applied;
- Usage of ImageNet, COCO, CityScapes, Medical Decathlon and other robust datasets, evaluating the generative power of our approaches;
- Multi-objective implementation of our approaches in different environments (e.g., GPU, CPU, specific smartphones) to reduce the number of parameters, number of MAdds, and latency without losing prediction performance;
- Further research in the influence of data augmentation and transfer learning combined with NAS;
- Employ dynamic configuration of the number of channels according to the cell structure and overall network, minimizing the requirement of manual configuration of the network width;
- Research of robust methods which pre-evaluates candidate models (like NAT), increasing the probability of finding the best models in a search space;
- Evaluating policies for training optimization and reduction of biased weights;
- Further studies in macro architecture search for image classification and semantic segmentation, like a multi-scale U-Net search.

Some points are highlighted for the NAS field, which may contribute to better developments of search strategies, search spaces, and evaluation methods:

- Present a more reliable execution metric (e.g., FLOPs) than GPU hours/days for the search phase, since a specific time in a GPU may not reflect the same number of iterations as in another GPU;
- NAS evaluation in different few GPU time (e.g., within one day and one week), to evaluate NAS with low search cost;

- Evaluations with and without transfer learning, with different data augmentations and different model sizes and computations. These studies may provide a better understanding of the variables which positively contributes to better scores;
- Evaluations of different search strategies (e.g., gradient-based against reinforcement learning and evolutionary-based NAS) in the same search space and configurations to evaluate the relevance of specific search strategies.

8.4 CONCLUSION

In this chapter, an overall discussion of NAS in the CIFAR datasets and the CHAOS challenge is made. We evaluate many approaches with the NAS methods proposed in this thesis. With this in mind, we explained the positive points in employing NAS and the proposed methods. Also, we discuss following directions in the NAS field and further work to be developed regarding our approaches.

9 CONCLUSION

In this thesis, efficient and low-cost neural architecture search based on evolutionary computation was researched. We discussed and validated the hypothesis of practical employment of evolutionary-based NAS to achieve competitive models without the requirement of hundreds of GPU power. For this purpose, we proposed two evolutionary-based NAS: one more conventional and another with more dynamic training. We employed several studies on the CIFAR datasets and the CHAOS challenge to evaluate the proposed methods to assess their effectiveness in different tasks.

We employed an ablation study with several experiments on the CIFAR-10 dataset, one of the studied datasets, for NAS evaluation. Different configurations were evaluated to assess the robustness of our approaches to hyper-parameter alteration. Overall, little difference was found from the experiments employed when small policies in our search strategies were changed. As for modifications in the search space, our methods suffered accuracy drops when bad combinations were employed in the search space. However, even when widening the search space with five-cell models with variable blocks, our proposals had competitive results with similar search times. Regarding random sampling and manually designed models, we had superior accuracy and significant differences. The search cost of twelve GPU hours is minimal for the gains found, showing the advantage of automating network design instead of manual adjustments. This pattern encourages a more profound study in diverse search spaces and further research outside the normal/reduction cell combination. Migration to other domains also had promising results. In the CIFAR-100 dataset, test errors similar to the state-of-the-art approaches with similar parameters were found without hyper-parameter tuning. In the CHAOS challenge, we had superior performance compared to many methods developed in the challenge, and some NAS approaches were applied to this dataset.

As our final considerations, the proposed thesis supports the evidence that using evolutionary-based NAS methods with only a few GPU hours can achieve state-of-the-art results in different tasks. Further research is encouraged to evaluate more complex problems and improve candidate model evaluation to reduce overall error (and standard deviation). The usage of robust and low-cost NAS may encourage the pursuit of more robust and efficient deep learning models.

REFERENCES

Abraham, N. and Khan, N. M. (2019). A novel focal tversky loss function with improved attention u-net for lesion segmentation. In *2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019)*, pages 683–687. IEEE.

Alves, J. H., Neto, P. M. M., and Oliveira, L. F. (2018). Extracting Lungs from CT Images Using Fully Convolutional Networks. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.

Alves, J. H. and Oliveira, L. F. (2020). Optimizing Neural Architecture Search using Limited GPU Time in a Dynamic Search Space: A Gene Expression Programming Approach. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8.

Baker, B., Gupta, O., Naik, N., and Raskar, R. (2016). Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*.

Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., and Le, Q. (2018). Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 549–558.

Bengio, Y., Lecun, Y., and Hinton, G. (2021). Deep learning for AI. *Communications of the ACM*, 64(7):58–65.

Bertels, J., Robben, D., Vandermeulen, D., and Suetens, P. (2019). Optimization with soft dice can lead to a volumetric bias. In *International MICCAI Brainlesion Workshop*, pages 89–97. Springer.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2017). SMASH: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*.

Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. (2018a). Efficient architecture search by network transformation. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Cai, H., Yang, J., Zhang, W., Han, S., and Yu, Y. (2018b). Path-Level Network Transformation for Efficient Architecture Search. *arXiv preprint arXiv:1806.02639*.

Cai, H., Zhu, L., and Han, S. (2018c). Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*.

Chen, Y., Yang, T., Zhang, X., Meng, G., Pan, C., and Sun, J. (2019). DetNAS: Neural Architecture Search on Object Detection. *arXiv preprint arXiv:1903.10979*.

Chollet, F. (2017). Xception: Deep Learning With Depthwise Separable Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1251–1258.

Chu, X., Zhang, B., and Li, X. (2020). Noisy differentiable architecture search. *arXiv preprint arXiv:2005.03566*.

Chu, X., Zhang, B., Xu, R., and Li, J. (2019). Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. *arXiv preprint arXiv:1907.01845*.

Çiçek, Ö., Abdulkadir, A., Lienkamp, S. S., Brox, T., and Ronneberger, O. (2016). 3d u-net: learning dense volumetric segmentation from sparse annotation. In *International conference on medical image computing and computer-assisted intervention*, pages 424–432. Springer.

Conze, P.-H., Kavur, A. E., Cornec-Le Gall, E., Gezer, N. S., Le Meur, Y., Selver, M. A., and Rousseau, F. (2021). Abdominal multi-organ segmentation with cascaded convolutional and adversarial deep networks. *Artificial Intelligence in Medicine*, 117:102109.

Cordeiro, C. Q., Ioshii, S. O., Alves, J. H., and Oliveira, L. F. (2018). An Automatic Patch-based Approach for HER-2 Scoring in Immunohistochemical Breast Cancer Images Using Color Features. *arXiv preprint arXiv:1805.05392*.

Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., and Le, Q. V. (2018). AutoAugment: Learning Augmentation Policies from Data. *arXiv preprint arXiv:1805.09501*.

DeVries, T. and Taylor, G. W. (2017). Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*.

Dice, L. R. (1945). Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302.

Dolz, J., Desrosiers, C., and Ayed, I. B. (2018). Ivd-net: Intervertebral disc localization and segmentation in mri with a multi-modal unet. In *International workshop and challenge on computational methods and clinical applications for spine imaging*, pages 130–143. Springer.

Dong, J.-D., Cheng, A.-C., Juan, D.-C., Wei, W., and Sun, M. (2018). PPP-Net: Platform-aware Progressive Search for Pareto-optimal Neural Architectures. In *ICML Workshop*.

Dong, X. and Yang, Y. (2019). Searching for a robust neural architecture in four gpu hours. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1761–1770.

Eiben, A. E., Smith, J. E., et al. (2003). *Introduction to evolutionary computing*, volume 53. Springer.

Elsken, T., Metzen, J.-H., and Hutter, F. (2017). Simple and efficient architecture search for Convolutional Neural Networks. *arXiv preprint arXiv:1711.04528*.

Elsken, T., Metzen, J. H., and Hutter, F. (2019). Efficient Multi-objective Neural Architecture Search via Lamarckian Evolution.

Everingham, M., Eslami, S. A., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. (2015). The pascal visual object classes challenge: A retrospective. *International journal of computer vision*, 111(1):98–136.

Ferreira, C. (2001). Gene Expression Programming: A New Adaptive Algorithm for Solving Problems. *Complex Systems*, 13(2):87–129.

Ferreira, C. (2006a). Automatically defined functions in gene expression programming. In *Genetic Systems Programming*, pages 21–56. Springer.

Ferreira, C. (2006b). *Gene expression programming: mathematical modeling by an artificial intelligence*, volume 21. Springer.

Floreano, D., Dürr, P., and Mattiussi, C. (2008). Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

Han, S., He, Y., Carass, A., Ying, S. H., and Prince, J. L. (2019). Cerebellum parcellation with convolutional neural networks. In *Medical Imaging 2019: Image Processing*, volume 10949, page 109490K. International Society for Optics and Photonics.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., et al. (2019). Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324.

Hundt, A., Jain, V., and Hager, G. D. (2019). sharpdarts: Faster and more accurate differentiable architecture search. *arXiv preprint arXiv:1903.09900*.

Hutter, F., Kotthoff, L., and Vanschoren, J., editors (2018). *Automatic Machine Learning: Methods, Systems, Challenges*. Springer. In press, available at http://automl.org/book.

Iglovikov, V. and Shvets, A. (2018). Ternausnet: U-net with vgg11 encoder pre-trained on imagenet for image segmentation. *arXiv preprint arXiv:1801.05746*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR.

Isensee, F., Jäger, P. F., Kohl, S. A., Petersen, J., and Maier-Hein, K. H. (2019). Automated design of deep learning methods for biomedical image segmentation. *arXiv preprint arXiv:1904.08128*.

Jiang, W., Zhang, X., Sha, E. H.-M., Yang, L., Zhuge, Q., Shi, Y., and Hu, J. (2019). Accuracy vs. Efficiency: Achieving Both through FPGA-Implementation Aware Neural Architecture Search. *arXiv preprint arXiv:1901.11211*.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. (2018). Neural architecture search with bayesian optimisation and optimal transport. *arXiv preprint arXiv:1802.07191*.

Kavur, A. E., Gezer, N. S., Barış, M., Aslan, S., Conze, P.-H., Groza, V., Pham, D. D., Chatterjee, S., Ernst, P., Özkan, S., et al. (2021). CHAOS challenge-combined (CT-MR) healthy abdominal organ segmentation. *Medical Image Analysis*, 69:101950.

Kavur, A. E., Selver, M. A., Dicle, O., Barış, M., and Gezer, N. S. (2019). CHAOS - Combined (CT-MR) Healthy Abdominal Organ Segmentation Challenge Data.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press.

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, Citeseer.

Laroca, R., Araujo, A. B., Zanlorensi, L. A., de Almeida, E. C., and Menotti, D. (2021a). Towards Image-Based Automatic Meter Reading in Unconstrained Scenarios: A Robust and Efficient Approach. *IEEE Access*, 9:67569–67584.

Laroca, R., Zanlorensi, L. A., Gonçalves, G. R., Todt, E., Schwartz, W. R., and Menotti, D. (2021b). An Efficient and Layout-Independent Automatic License Plate Recognition System Based on the YOLO Detector. *IET Intelligent Transport Systems*, 15(4):483–503.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436.

Li, G., Qian, G., Delgadillo, I. C., Muller, M., Thabet, A., and Ghanem, B. (2020). Sgas: Sequential greedy architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1620–1630.

Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

Liu, C., Chen, L.-C., Schroff, F., Adam, H., Hua, W., Yuille, A., and Fei-Fei, L. (2019a). Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. *arXiv preprint arXiv:1901.02985*.

Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2017a). Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. (2017b). Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*.

Liu, H., Simonyan, K., and Yang, Y. (2018). DARTS: Differentiable Architecture Search. *arXiv preprint arXiv:1806.09055*.

Liu, P., El Basha, M. D., Li, Y., Xiao, Y., Sanelli, P. C., and Fang, R. (2019b). Deep Evolutionary Networks with Expedited Genetic Algorithm for Medical Image Denoising. *Medical Image Analysis*.

Lorenzo, P. R. and Nalepa, J. (2018). Memetic evolution of deep neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 505–512. ACM.

Loshchilov, I. and Hutter, F. (2016). Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*.

Loshchilov, I. and Hutter, F. (2018). Fixing weight decay regularization in adam.

Lu, Z., Sreekumar, G., Goodman, E., Banzhaf, W., Deb, K., and Boddeti, V. N. (2021). Neural architecture transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E., and Banzhaf, W. (2018). NSGA-NET: A Multi-Objective Genetic Algorithm for Neural Architecture Search. *arXiv preprint arXiv:1810.03522*.

Lucio, D. R., Laroca, R., Zanlorensi, L. A., Moreira, G., and Menotti, D. (2019). Simultaneous iris and periocular region detection using coarse annotations. In *Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 178–185.

Luz, E., Moreira, G., Junior, L. A. Z., and Menotti, D. (2018). Deep periocular representation aiming video surveillance. *Pattern Recognition Letters*.

Malladi, R. and Sethian, J. A. (1996). Image processing: Flows under min/max curvature and mean curvature. *Graphical models and image processing*, 58(2):127–141.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60.

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., et al. (2017). Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*.

Miller, G. F., Todd, P. M., and Hegde, S. U. (1989). Designing Neural Networks using Genetic Algorithms. In *ICGA*, volume 89, pages 379–384.

Mortazi, A. and Bagci, U. (2018). Automatically designing CNN architectures for medical image segmentation. In *International Workshop on Machine Learning in Medical Imaging*, pages 98–106. Springer.

Mower, J. P. (2005). PREP-Mt: predictive RNA editor for plant mitochondrial genes. *BMC bioinformatics*, 6(1):1–15.

Nalepa, J., Marcinkiewicz, M., and Kawulok, M. (2019). Data augmentation for brain-tumor segmentation: a review. *Frontiers in computational neuroscience*, 13:83.

Nandamuri, S., China, D., Mitra, P., and Sheet, D. (2019). Sumnet: Fully convolutional model for fast segmentation of anatomical structures in ultrasound volumes. In *2019 IEEE 16th international symposium on biomedical imaging (ISBI 2019)*, pages 1729–1732. IEEE.

Nekrasov, V., Chen, H., Shen, C., and Reid, I. (2019a). Architecture Search of Dynamic Cells for Semantic Video Segmentation. *arXiv preprint arXiv:1904.02371*.

Nekrasov, V., Shen, C., and Reid, I. (2019b). Template-Based Automatic Search of Compact Semantic Segmentation Architectures. *arXiv preprint arXiv:1904.02365*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*.

Perone, C. S., Calabrese, E., and Cohen-Adad, J. (2018). Spinal cord gray matter segmentation using deep dilated convolutions. *Scientific reports*, 8(1):1–13.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018). Efficient Neural Architecture Search via Parameter Sharing. In *ICML*.

Prellberg, J. and Kramer, O. (2018). Lamarckian Evolution of Convolutional Neural Networks. *arXiv preprint arXiv:1806.08099*.

Quan, H.-y. and Yang, G. (2007). Gene expression programming with dag chromosome. In *International Symposium on Intelligence Computation and Applications*, pages 271–275. Springer.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. (2017). Large-Scale Evolution of Image Classifiers. In *International Conference on Machine Learning*, pages 2902–2911.

Ren, P., Xiao, Y., Chang, X., Huang, P.-Y., Li, Z., Chen, X., and Wang, X. (2021). A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)*, 54(4):1–34.

Ronneberger, O., Fischer, P., and Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer.

Ruiz, D. V., Krinski, B. A., and Todt, E. (2019). ANDA: A Novel Data Augmentation Technique Applied to Salient Object Detection. In *2019 19th International Conference on Advanced Robotics (ICAR)*, pages 487–492.

Ruiz, D. V., Krinski, B. A., and Todt, E. (2020). IDA: Improved Data Augmentation Applied to Salient Object Detection. In *2020 33rd SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 210–217.

Ruiz, D. V. and Todt, E. (2021). BEyond observation: an approach for ObjectNav. *2th Embodied AI Workshop at IEEE Conference on Computer Vision and Pattern Recognition*.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

Salomon, G., Britto, A., Vareto, R. H., Schwartz, W. R., and Menotti, D. (2020). Open-set Face Recognition for Small Galleries Using Siamese Networks. In *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*, pages 161–166. IEEE.

Saxena, S. and Verbeek, J. (2016). Convolutional neural fabrics. In *Advances in Neural Information Processing Systems*, pages 4053–4061.

Shvets, A. A., Rakhlin, A., Kalinin, A. A., and Iglovikov, V. I. (2018). Automatic instrument segmentation in robot-assisted surgery using deep learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 624–628. IEEE.

Silva, P. H., Luz, E., Zanlorensi, L. A., Menotti, D., and Moreira, G. (2018). Multimodal feature level fusion based on particle swarm optimization with deep transfer learning. In *2018 Congress on Evolutionary Computation (CEC)*, pages 1–8.

Simard, P. Y., Steinkraus, D., Platt, J. C., et al. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, volume 3.

Sinha, N. and Chen, K.-W. (2020). Evolving Neural Architecture Using One Shot Model.

Smith, L. N. (2018). A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*.

Sorensen, T. A. (1948). A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons. *Biol. Skar.*, 5:1–34.

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Highway networks. *arXiv preprint arXiv:1505.00387*.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

Suganuma, M., Shirakawa, S., and Nagao, T. (2017). A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 497–504. ACM.

Tan, M. and Le, Q. V. (2021). Efficientnetv2: Smaller models and faster training. *arXiv preprint arXiv:2104.00298*.

Wang, X., Xiang, T., Zhang, C., Song, Y., Liu, D., Huang, H., and Cai, W. (2021). Bix-nas: Searching efficient bi-directional architecture for medical image segmentation. *arXiv preprint arXiv:2106.14033*.

Weng, Y., Zhou, T., Li, Y., and Qiu, X. (2019). Nas-unet: Neural architecture search for medical image segmentation. *IEEE Access*, 7:44247–44257.

Wu, Y. and He, K. (2018). Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19.

Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. (2017). Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 5987–5995. IEEE.

Xie, S., Zheng, H., Liu, C., and Lin, L. (2018). SNAS: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*.

Yan, X., Jiang, W., Shi, Y., and Zhuo, C. (2020). Ms-nas: Multi-scale neural architecture search for medical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 388–397. Springer.

Yang, A., Esperança, P. M., and Carlucci, F. M. (2019). NAS evaluation is frustratingly hard. *arXiv preprint arXiv:1912.12522*.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.

Yu, F. and Koltun, V. (2015). Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*.

Zanlorensi, L. A., Laroca, R., Lucio, D. R., Santos, L. R., Britto Jr., A. S., and Menotti, D. (2020a). UFPR-Periocular: A periocular dataset collected by mobile devices in unconstrained scenarios. *arXiv preprint*, arXiv:2011.12427:1–12.

Zanlorensi, L. A., Laroca, R., Luz, E., Britto Jr., A. S., Oliveira, L. S., and Menotti, D. (2021). Ocular recognition databases and competitions: A survey. *Artificial Intelligence Review*, pages 1–52.

Zanlorensi, L. A., Lucio, D. R., Britto Jr., A. S., Proença, H., and Menotti, D. (2020b). Deep representations for cross-spectral ocular biometrics. *IET Biometrics*, 9:68–77.

Zanlorensi, L. A., Luz, E., Laroca, R., Britto Jr., A. S., Oliveira, L. S., and Menotti, D. (2018). The impact of preprocessing on deep representations for iris recognition on unconstrained environments. In *Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 289–296.

Zanlorensi, L. A., Proença, H., and Menotti, D. (2020c). Unconstrained periocular recognition: Using generative deep learning frameworks for attribute normalization. In *2020 International Conference on Image Processing (ICIP)*, pages 1361–1365.

Zela, A., Klein, A., Falkner, S., and Hutter, F. (2018). Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *arXiv preprint arXiv:1807.06906*.

Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2020). *Dive into Deep Learning*. `https://d2l.ai`.

Zhong, J., Ong, Y.-S., and Cai, W. (2016). Self-learning gene expression programming. *IEEE Transactions on Evolutionary Computation*, 20(1):65–80.

Zhong, Z., Yan, J., Wu, W., Shao, J., and Liu, C.-L. (2018). Practical Block-wise Neural Network Architecture Generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2423–2432.

Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2017). Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2(6).